

# Drops of knowledge of C++

Yan Zhao



# Contents

<b>Preface</b>	<b>11</b>
<b>How to</b>	<b>13</b>
<b>1 C++ Introduction</b>	<b>15</b>
1.1 Overview . . . . .	15
1.1.1 Multi-paradigm . . . . .	15
1.1.2 Statement and expression . . . . .	15
1.2 Compilation and link . . . . .	17
1.2.1 Separate compilation and header file . . . . .	17
1.2.2 Declaration and definition . . . . .	18
1.2.2.1 Forward declaration . . . . .	19
1.2.3 Linkage . . . . .	21
1.2.3.1 ODR . . . . .	21
1.2.3.2 external and internal linkage . . . . .	21
1.2.3.3 Inline function . . . . .	22
1.2.4 Combine C and C++ . . . . .	24
<b>2 type, operator and expression</b>	<b>27</b>
2.1 Numerical . . . . .	27
2.1.1 Numerical . . . . .	27
2.1.1.1 Numerical Overflow . . . . .	27
2.1.2 Numerical conversions . . . . .	28
2.1.2.1 implicit promotion . . . . .	29
2.1.2.2 Explicit numerical conversion . . . . .	30
2.2 type . . . . .	30
2.2.1 Type cast in c++ . . . . .	30
2.2.1.1 Implicit conversion in C++ . . . . .	30
2.2.1.2 type cast operator . . . . .	32
2.2.1.3 RTTI . . . . .	34
2.2.2 cv-qualifier . . . . .	34
2.2.2.1 const in function . . . . .	34
2.2.2.2 const in class . . . . .	36
2.2.2.3 const_interator . . . . .	39
2.2.3 constexpr . . . . .	40
2.2.3.1 constant expression . . . . .	40
2.2.3.2 constexpr variable . . . . .	41
2.2.3.3 constexpr function . . . . .	42
2.2.4 static . . . . .	45

2.2.5	size_t and ptrdiff_t . . . . .	46
2.2.5.1	unsigned int . . . . .	46
2.2.5.2	size_t . . . . .	47
2.2.6	Aggregate and POD . . . . .	47
2.2.6.1	Aggregate . . . . .	47
2.2.6.2	POD . . . . .	49
2.2.6.3	trivial class . . . . .	50
2.3	Name . . . . .	51
2.3.1	Namespace . . . . .	51
2.3.1.1	namespace basic knowledge . . . . .	51
2.3.2	Name lookup . . . . .	53
2.3.2.1	Name hiding . . . . .	54
2.3.2.2	ADL . . . . .	55
2.4	overload . . . . .	58
<b>3</b>	<b>initialization</b>	<b>61</b>
3.1	Basic . . . . .	61
3.1.1	principle rule . . . . .	61
3.1.2	initialization order . . . . .	61
3.1.3	Init method . . . . .	62
3.1.3.1	Six init methods . . . . .	62
3.1.3.2	value init . . . . .	64
3.1.3.3	copy init and direct init . . . . .	65
3.2	Brace Init . . . . .	68
3.2.1	Basic syntax . . . . .	68
3.2.2	Brace init advantage . . . . .	69
3.2.3	copy-list-initialization and traditional copy-initialization? . . . . .	71
3.2.4	vexing parsing . . . . .	73
3.3	initiliaer_list . . . . .	75
3.3.1	why initiliaer_list . . . . .	75
3.3.2	difference with brace init . . . . .	75
3.3.3	problem of initializer_list . . . . .	76
3.4	auto init(AAA) . . . . .	77
3.4.1	auto declaration . . . . .	77
3.4.1.1	Basic knowledge . . . . .	77
3.4.1.2	pitfall of auto . . . . .	79
3.4.2	auto and function . . . . .	80
3.4.2.1	parameter . . . . .	80
3.4.2.2	Generalized return type deduction . . . . .	81
3.4.2.3	atuo in lambda and template . . . . .	82
3.5	summary . . . . .	82
3.5.1	Syntax demo . . . . .	82
3.5.2	How to understand init . . . . .	83
3.5.3	How to init by youself . . . . .	84
<b>4</b>	<b>Memory</b>	<b>85</b>
4.1	malloc and remalloc . . . . .	85
4.1.1	malloc . . . . .	85
4.1.2	realloc . . . . .	86
4.2	new operator . . . . .	86

4.2.1	Basic . . . . .	86
4.2.2	Inside of new operator . . . . .	87
4.2.3	new_handler . . . . .	88
4.2.4	placement new . . . . .	89
4.2.5	array new . . . . .	90
4.2.6	Customize operator new . . . . .	91
<b>5</b>	<b>pointer and smart pointer</b>	<b>93</b>
5.1	pointer and new . . . . .	93
5.1.1	function pointer . . . . .	93
5.1.2	When to use new? . . . . .	93
5.2	basic smart pointer knowledge . . . . .	94
5.2.1	Smart pointer Basic knowledge . . . . .	94
5.2.2	unique_ptr . . . . .	96
5.2.2.1	basic . . . . .	96
5.2.2.2	unique_ptr and container . . . . .	98
5.2.3	shared_ptr . . . . .	99
5.2.4	weak_ptr . . . . .	102
5.2.5	make function . . . . .	103
5.2.6	wrapping resource handler in smart pointer . . . . .	104
5.2.6.1	basic idea . . . . .	104
5.2.6.2	Examples . . . . .	105
5.3	smart pointer and polymorphism . . . . .	108
5.3.1	pointer_cast function . . . . .	108
5.3.2	Usage . . . . .	108
5.4	smart pointer and class . . . . .	110
5.4.1	RAII . . . . .	110
5.5	smart pointer Summary . . . . .	113
5.5.1	principle . . . . .	113
5.5.2	Function interface . . . . .	115
<b>6</b>	<b>reference and rvalue reference</b>	<b>117</b>
6.1	reference basic . . . . .	117
6.2	lvalue, rvalue and xvalue . . . . .	118
6.2.1	Definition . . . . .	119
6.2.2	Example of xvalue . . . . .	120
6.2.3	xvalue and rvalue reference . . . . .	121
6.2.4	Why need xvalue . . . . .	122
6.3	rvalue reference and move scenario . . . . .	123
6.3.1	basic of rvalue reference . . . . .	123
6.3.2	move ctor and assignment . . . . .	124
6.3.3	move semantic . . . . .	126
6.4	universal(forwarding) reference . . . . .	126
6.4.1	definition . . . . .	126
6.4.2	pros . . . . .	128
6.4.3	cons . . . . .	128
6.4.4	Usage of forwarding reference . . . . .	129
6.4.5	std::move and std::forward implementation . . . . .	130
6.5	function interface-parameter . . . . .	132
6.5.1	Generic function parameter design . . . . .	132

6.5.2	read-copy function parameter design . . . . .	132
6.5.2.1	overload solution . . . . .	133
6.5.2.2	only value solution . . . . .	133
6.5.2.3	only rvalue reference solution . . . . .	135
6.5.2.4	forwarding reference . . . . .	136
6.6	funciton interface-return . . . . .	136
6.6.1	return plain reference . . . . .	137
6.6.2	return rvalue reference . . . . .	138
6.6.3	return value-RVO . . . . .	140
6.6.3.1	common RVO case . . . . .	140
6.6.3.2	RVO limitations . . . . .	141
6.6.3.3	Some practical demos ans analysis . . . . .	142
6.6.3.4	RVO and move . . . . .	143
6.6.3.5	RVO summary . . . . .	144
<b>7</b>	<b>OOP</b> . . . . .	<b>145</b>
7.1	Object based . . . . .	145
7.1.1	class categories . . . . .	145
7.1.2	Interface . . . . .	146
7.2	member function . . . . .	148
7.2.1	Special member functions relationship . . . . .	148
7.2.1.1	Basic . . . . .	148
7.2.1.2	Rules of implicitly declare . . . . .	151
7.2.1.3	initializer list . . . . .	153
7.2.2	Basic pattern . . . . .	155
7.2.2.1	constructor and destructor . . . . .	155
7.2.2.2	copy and swap idiom . . . . .	157
7.2.2.3	Big zero, three and five . . . . .	158
7.2.3	operator overload . . . . .	160
7.3	inheriance . . . . .	161
7.3.1	special member functions in inheritance . . . . .	162
7.3.1.1	ctor . . . . .	162
7.3.1.2	destructor . . . . .	163
7.3.1.3	copy ctor in inheritance . . . . .	164
7.3.2	virtual function and override . . . . .	166
7.4	Classes relationship . . . . .	167
7.4.1	structure semantic . . . . .	167
7.4.1.1	Definition . . . . .	167
7.4.2	Inheritance semantic . . . . .	169
7.4.2.1	private inheritance . . . . .	169
7.4.2.2	"Has-A" relationship . . . . .	170
7.4.2.3	"Is-A" relationship . . . . .	171
7.4.3	Ownership semantic . . . . .	173
7.4.4	Summary . . . . .	174
7.4.4.1	Examples . . . . .	174
7.5	design pattern . . . . .	175
7.5.1	Common used principle . . . . .	175
7.5.1.1	Three big rules . . . . .	175
7.5.1.2	NFA and NVI . . . . .	176

7.5.2	Resource wrapper and RAI <sup>I</sup> . . . . .	176
7.5.3	MI or bridge . . . . .	180
7.5.4	factory, bridge and visitor . . . . .	181
<b>8</b>	<b>Generic programming</b>	<b>183</b>
8.1	Template Basic . . . . .	183
8.1.1	template parameter . . . . .	183
8.1.2	template instantiation . . . . .	185
8.1.3	template specialization . . . . .	186
8.1.4	template and friend . . . . .	188
8.1.5	member function templates . . . . .	190
8.2	Type Inference . . . . .	190
8.2.1	template type deduction . . . . .	190
8.2.2	auto type deduction . . . . .	192
8.2.3	decltype deduction . . . . .	193
8.2.3.1	basice knowledge . . . . .	193
8.2.3.2	decltype usage . . . . .	194
8.2.4	check type . . . . .	197
8.2.5	summary . . . . .	197
8.3	type traits and policy . . . . .	198
8.3.1	implementation . . . . .	198
8.3.2	Usage . . . . .	202
8.4	Template function . . . . .	202
8.4.1	overload resolution . . . . .	203
8.4.2	template function specification . . . . .	204
8.4.3	summary . . . . .	205
8.5	template and inheritance . . . . .	209
8.6	template common idiom . . . . .	211
8.6.1	policy, mixin and CRTP . . . . .	211
8.6.1.1	policy . . . . .	211
8.6.1.2	Mixin . . . . .	212
8.6.1.3	CRTP . . . . .	213
8.6.1.4	A practical example . . . . .	215
8.6.1.5	summary . . . . .	218
8.6.2	tag dispatch . . . . .	219
8.6.2.1	enable_if . . . . .	221
8.6.3	type erasure and concept . . . . .	223
8.6.3.1	function . . . . .	223
8.6.3.2	type erasure . . . . .	224
<b>9</b>	<b>STL</b>	<b>227</b>
9.1	Basic . . . . .	227
9.2	Container . . . . .	228
9.2.1	Basic knowledge . . . . .	228
9.2.2	Basic classifications . . . . .	229
9.2.2.1	structure classification . . . . .	229
9.2.2.2	memory classification . . . . .	231
9.2.3	Usage . . . . .	232
9.2.3.1	Search in Container . . . . .	232
9.2.3.2	Range . . . . .	234

9.2.3.3	Erasure . . . . .	235
9.2.3.4	type definition in container . . . . .	236
9.2.3.5	Sizes . . . . .	237
9.2.3.6	Usages Tips . . . . .	238
9.2.4	string . . . . .	240
9.3	Iterator . . . . .	242
9.3.1	Insert iterator . . . . .	244
9.3.2	Reverse iterator . . . . .	245
9.4	Algorithms . . . . .	246
9.4.1	Basic . . . . .	246
9.4.2	STL algorithms . . . . .	248
9.4.2.1	basic notation . . . . .	248
9.4.2.2	Applying . . . . .	248
9.4.2.3	Bounding . . . . .	250
9.4.2.4	Comparing . . . . .	250
9.4.2.5	copy . . . . .	250
9.4.2.6	Count . . . . .	251
9.4.2.7	Filling and Generating . . . . .	251
9.4.2.8	Math . . . . .	252
9.4.2.9	Merging . . . . .	253
9.4.2.10	Partitioning . . . . .	253
9.4.2.11	Permuting . . . . .	254
9.4.2.12	Random/shuffling . . . . .	254
9.4.2.13	Removing . . . . .	254
9.4.2.14	Replacing . . . . .	254
9.4.2.15	Reverse . . . . .	254
9.4.2.16	Rotating . . . . .	254
9.4.2.17	Searching . . . . .	255
9.4.2.18	set . . . . .	255
9.4.2.19	swapping . . . . .	256
9.4.2.20	sort . . . . .	256
9.5	Function object . . . . .	256
9.5.1	Basic . . . . .	256
9.5.2	Adaptable . . . . .	258
9.5.2.1	Before c++11 . . . . .	258
9.5.2.2	After c++11 . . . . .	259
9.5.2.3	member function . . . . .	259
9.5.3	functor tips . . . . .	261
9.5.3.1	when to use std::function . . . . .	261
<b>10</b>	<b>Exception and error</b>	<b>263</b>
10.1	End application . . . . .	263
10.2	Bug and assert . . . . .	264
10.2.1	Use assert . . . . .	264
10.2.2	Trace . . . . .	265
10.3	Handling exceptions . . . . .	265
10.3.1	errno in C . . . . .	265
10.3.2	exceptions in C++ . . . . .	267
10.4	Conclusion . . . . .	268

<b>11 functional programming</b>	<b>271</b>
<b>12 concurrent</b>	<b>273</b>
12.1 data race . . . . .	273
12.2 synchronization . . . . .	273
<b>13 I/O</b>	<b>275</b>
13.1 I/O basic . . . . .	275
13.1.1 I/O basic knowledge . . . . .	275
13.2 Input . . . . .	276
13.2.1 Input basic knowledge . . . . .	276
13.2.2 Input error . . . . .	278
13.2.3 Input Pattern . . . . .	279
13.3 output . . . . .	281
13.4 other stream . . . . .	281
13.4.1 file . . . . .	281
13.4.2 buffer and string buffer . . . . .	282
13.5 manipulate stream . . . . .	282
<b>14 New Feature in modern C++</b>	<b>285</b>
14.1 New Type . . . . .	285
14.1.1 New int . . . . .	285
14.1.2 new container . . . . .	286
14.2 range base . . . . .	288
14.3 lambda . . . . .	289
14.4 Other New Feature . . . . .	291
14.4.1 decltype . . . . .	291
14.4.2 alias declaration . . . . .	291
14.4.3 scoped enums . . . . .	292
14.4.4 noexcept . . . . .	293
14.4.5 Variadic Templates . . . . .	294
<b>15 Style and Guideline</b>	<b>295</b>
15.1 efficiency . . . . .	295
15.2 Style . . . . .	295
15.2.1 Basic Principles . . . . .	295
15.2.2 Naming . . . . .	295
15.2.3 Comment and Document . . . . .	296
15.2.4 Code Convention . . . . .	298



# Preface

This is the third edition, The first edition has almost 100 pages and the second one has 200 pages. Then guess what happen in the third edition? it has almost 300 pages now.

Bjarne Stroustrup, the creator of C++, said that modern C++ "feels like a new language". I totally agree with this view point. By now, modern C++ is quicker and safer. I love this language and want to spread it, teach more people to use it. That is the purpose of this book.

I am a software developer, and have worked with C/C++ almost 30 years. I have published a very famous C language book "Drop of knowledge of C", and the link is:

<http://product.dangdang.com/23340055.html>. I am able to provide C++ tutorial and training, online or onsite. please contact me if you need this kind of service.

The first edition is more like studying note than book, The third edition is more like book now. The book has three characters:

1. "**Talk is cheap, Show me the code**". OK, a lot of code. Just very short, concise description with each code block. You can even think that this is a book of source code, with some comment around it.
2. "**A graph is Worth a 1000 Words**". The book provides many graphs to help illustrate these complex conception. You can even see a figure on the cover of this book.
3. "**Design is not how it looks, but how it works**". The four chapters "pointer and smart pointer", "reference and rvalue reference", "OOP" and "Generic programming" introduce a lot of deep semantic knowledge in these field. You can learn not only language knowledge but also some design idea.

Compared with the third edition, The fourth edition added three chapters: functional programming, concurrent and style and guideline. Additionally, A lot of improvements on format and new contents on the existing chapter, specifically some new knowledge on the new C++ standard: C++20.

I appreciate my two daughters: Millie and Ivy. C++ language will be still alive when you grow up. Thank my wife Lina, you always said that writing a book was useless. You are right!. When husband says: "You are right!", the argument is over. When wife says: "you are right!", you are over.

Any suggestions and error reports are appreciated. You can contact me by:

Email : [zhaoyan.hrb@gmail.com](mailto:zhaoyan.hrb@gmail.com)

Homepage : <http://zhaoyan.website>

Wechat account : zhaoyan\_rock

I also have blog: <http://zhaoyan.website/blog/>. You can find some Chinese blogs there.



# How to?

- **How to read this book in eReader?**

1. Because the book has a lot of source code. so it prefer to read the book in landscape mode.  
If you are using kindle, you can google how to read ebooks in landscape mode on kindle.
2. If table is shown properly on the kindle, click the small icon below the table, then the whole table will be extracted and shown up in a separate page.
3. You can search keyword in the source code block.
4. You can purchase the printed book from Amazon.com. Just search "Drops of knowledge of C++" in Amazon.com. Frankly speaking, I prefer to printed book for this kind of computer programming book, because we have spent so much time on screens.

- **How to run the source code?**

1. Most of source code can run directly. In order to save space, I omit the head files,so please add the required head files and `main` function when you run the source code.
2. Most of source code illustrate the basic idea, so they are not long. You can use online C++ compiler. These light weight online tools are very suitable for the source code in the book. Just google "online C++ compiler" and select one with black background, because light abstracts bug. :)



# Chapter 1

## C++ Introduction

### 1.1 Overview

#### 1.1.1 Multi-paradigm

- C++ is a Multi-paradigm language, there are five paradigms:
  1. Procedural programming. (Traditional C programming)
  2. Object-base programming. (Class and object)
  3. Object-orient programming. (Inheritance and polymorphism)
  4. Generic programming. (Template)
  5. functional programming.(Function object)

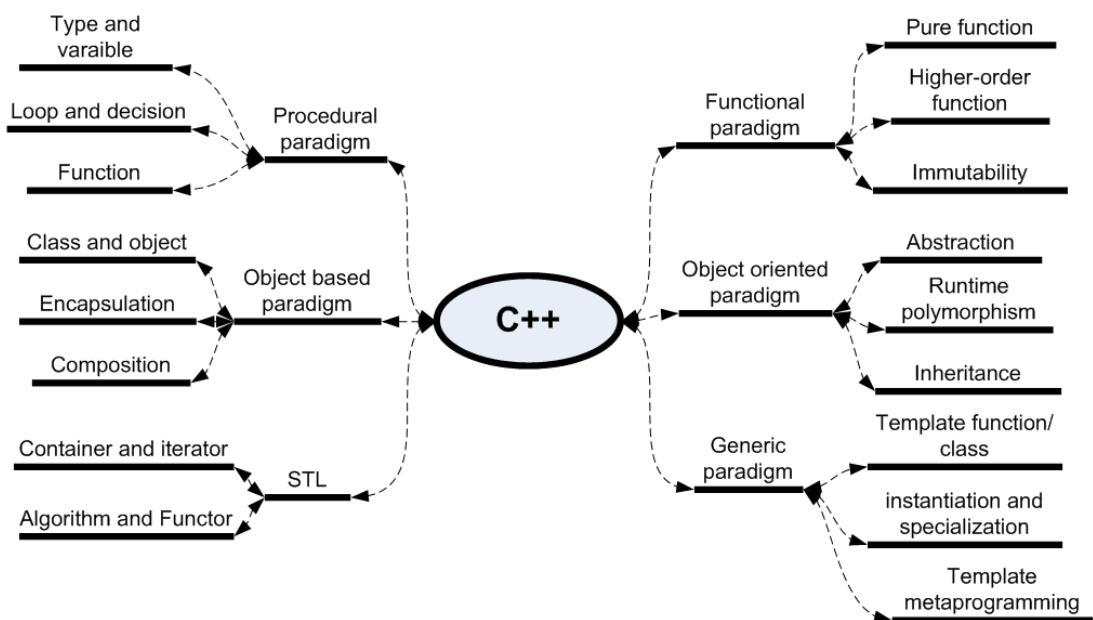


Figure 1.1: The main component of C++ language

#### 1.1.2 Statement and expression

- Statement and expression are two important conceptions, you can see their definition in [cppreference.com](http://cppreference.com) to see academic explanation. Although these two conceptions are a little theoretic,

a log of piratical knowledge and conceptions—xvalue, reference etc., are defined on these two conceptions. We need to understand them before we go deeper.

- An expression is a sequence of operators and their operands, that specifies a computation. The general operators are assignment, increment, arithmetic, logical, comparison and member access.

```
a=b, a+=b //assignment
++a, a++ //increment
a+b, a&b //arithmetic
a&&b, !a //logical
a<b, a!=b //comarision
a[b], a->b, a.b //member access
```

- Expression is different with statement.

1. b=c;
2. a=**b=c**

**Line 1:** b=c; is statement, because it has semicolon after it. It represents an action— assign a value to a variable.

**Line 2:** b=c is expression, it yields value(b), then we can use this value for outside.

- For operator, there are two important characteristic: **Precedence** and **Associativity**. About Associativity, give an example below:

1. a=b=c; //correct code
2. a<b<c; //error code
3. a<b && b<c //correct code

**Line 1:** assignment operator is **right-to-left**. It means that we 1)b=c, 2)b=c yields b 3) a=b

**Line 2:** logical operator is **left-to-right**. it means that 1)a<b, 2)a<b yields bool value 3) bool<c. That is error.code and not what you expect.

**Line 3:** That is correct expression.

- Function call is a expression, because it can yield a value.
- Statements are fragments of the C++ program that are executed in sequence. Only statement, which end with semicolon is executed.
- C++ mainly includes the following types of statement: 1) expression statement, 2) compound statement, 3)selection statement(if, switch), 4)iteration statement(while, for) and 5)Jump statement(break,continue, return, goto). Most statements are expression statements.

```
int n = 1; // declaration statement
n = n + 1; // expression statement
std::cout << n << '\n'; // expression statement
return 0; // return statement
```

- Difference between statement and expression.

1. Expression: Something which evaluates to a value. Example: 1+2/x
2. Statement: A line of code which does something. Example: GOTO 100; and statements are all end with semi-comma.

- The designers of C realized that no harm was done if you were allowed to evaluate an expression and throw away the result. In C, every syntactic expression can be made into a statement just by tacking a semicolon along the end:

```
x+y      //is expression;
x+y;    //is statement, but throw away the result
j=i;    //is a statement.
fun(i) //is expression;
```

## 1.2 Compilation and link

### 1.2.1 Separate compilation and header file

- Having only one source file for a large project is unrealistic, so we break the code up into its logical structure. In this way, only changed parts need to be recompiled, and reduce the compile time. When we use the multiple source file, each part needs to know what information about functions and variables "used" from other files. That is why we need declaring.
- Even you can declare variable, function and class many times, it's not good way to declare everywhere, **DO NOT copy declaration to the other positions in the other .cpp file. It will lead to many duplication.** If you modify it's name, you need to trace back all the declaration. If you need to use a function or variable in many different files, you should put them in a header filer. For example: export\_to.h for certain cpp source file or global.h file for the whole system.
- In your project, you can global search function or variable declaration, if you find declaration statement more than two, It's strong indication to make a global head file and put these declaration statements into it.**
- When you write an single source file (.cpp, .cxx, etc), your compiler generates a translation unit. That is to say, Each .cpp will become a translation unit. This is the object file from your source file plus all the headers you #include in it. A translation unit roughly consists of a source file after it has been processed by the **preprocessor**, meaning that header files listed in #include directives are literally included, sections of code within #ifdef may be included, and macros have been expanded.
- Basically, you should put each class definition into a single .cpp file, and make sure each .cpp file has a corresponding .h file. If two classes are highly correlated, they maybe be put in the same .cpp file.
- If you just want a function or variable visible to only current translation unit. You can declare it as static. Or you can use unnamed namespace. You don't need to put it into a header file. Just remember in C++, you have to declare function and variable before you first use them. Different with C language, In C language, compiler will guess a function prototype from it's usage, but it's not good most of time.
- A better suggestion is to have a single global.h file for a complex system. Then put some common type, defined type, constant and global function in this single global.h file. So it will help to reduce duplication, just keep once appearance.
- Three main contents in header file:
  - "Everything" that should be exported (i.e., used in other files)

2. "Nothing" that causes the compiler to immediately generate code
3. Except, for a small number of exceptions. Such as const and inline function, which has **internal linkage**.

- Three rules about header file:

1. Use **#pragma once** to add include guard, It's not standard, but It has been supported by many compiler. Including g++, clang and MSVC.
2. **Put your local/private header file in front of system header file.** Why? There are two advantages:
  - (a) You can know what header file should be included, It's helpful to achieve the goal of demand.
  - (b) Sometimes, if you have your own function with same name as system or library, It can give you a compile error; Below example will give you a compile error. But if you put `<cmath>` before `myHead.h`. Then, main will use acos in cmath, and your acos will be override.

```
#include "myHead.h" // double acos(double)
#include <cmath>
main{
    acos(0.5);
}
```

3. You will need to put the minimal set of #include statements that are needed to make the header compilable when your local/private header is included on the first place. It will make your header file self-sufficient.

- In .h file, you can include template and inline function. In fact, you have to put template into .h file.
- **Putting a semicolon in the end of head file is good suggestion.** You also need semicolon after declare class. In .cpp file, no semicolon after each function definition.
- You can google "Advanced Software Engineering with C++ Templates". The first half part introduce separate compilation.

### 1.2.2 Declaration and definition

- A declaration introduces an identifier and describes its type, be it a type, object, or function. **A declaration is what the compiler needs to accept references to that identifier.**

```
1. extern int bar;
2. extern int g(int, int);
3. double f(int, double); // it is declaration.
4. class Foo;
```

**Line 1:** Add `extern` keyword before variable name. It make it as declaration statement, not definition.

**Line 3:** `extern` can be omitted for function declarations.

**Line 4:** no `extern` allowed for type declarations.

- A definition actually instantiates/implements this identifier. It's what the linker needs in order to link references to those entities. These are definitions corresponding to the above declarations:

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
class foo {} //put {} after class definition.
```

- The difference between declaring a symbol and defining a symbol:

1. A declaration tells the compiler about the existence of a certain symbol and makes it possible to refer to that symbol everywhere where the explicit memory address or required storage of that symbol is not required. A definition tells the compiler what the body of a function contains or how much memory it must allocate for a variable.
2. One very important difference between declarations and definitions is that a symbol may be declared many times in different translating unit, but defined in one translating unit only once. For example, you can forward declare a function or class however often you want, but you may only ever have one definition for it. This is called the **One Definition Rule**.

### 1.2.2.1 Forward declaration

- In C++, there exists the concept of forward declaring a symbol. We declare the type and name of a symbol so that we can use it where its definition is not required. There are three usages as below:

1. It will reduce compile-time dependencies -PIMPL.
2. Hide all the detail.
3. Break cyclic references

```
// file.hpp
class C1; // that is forward declaring.
class C2{
...
    C1* pc1;
}
```

- Forward declaration doesn't work if you need to build or access its member. It's only work when you refer it by pointer or reference.

```
1. class Foo; // that is forward declaring.
2. Foo* f1 = new Foo; //error,
3.
4. fun(Foo* f1){
5.     f1->a;           //error,
6. }
7. fun(Foo f1)        //error,
```

**Line 2:** You need #include Foo.h, so compiler can know all the size of Foo.

**Line 5:** Compiler need to know if there is variable a inside of Foo.

**Line 7:** It's not pointer or reference type.

- About cyclic include, you need to know below:

1. You can't write the code below, because the compiler will not know the size of A and B.

```
class A{
    B b;
}

class B{
    A a;
}
```

2. Use pointer or reference to tackle cyclic dependent problem. But it still has include cyclic dependent problem.

```
#include "b.h"
class A{
    B* b;
}

#include "a.h"
class B{
    A a;
}
```

3. In the end, you can use forward declaration to remove #include statement.

```
//a.h file
class B;
class A{
    B* b;
}
```

- About pimpl, you need to know below:

1. In your .h file, when you use `Foo* p` or `Foo& rp`; You don't need include `Foo.h` file. you can use forward declaration.

```
class Foo; //forward declaration
.....
Foo* p; //use it , but only define a pointer.
```

2. Base on previous forward declaration, you can use "Pimpl" idiom.

```
class Widget { // "widget.h" file
private:
    struct Impl; // forward declaration
    Impl* pImpl;
};

#include "Foo.h" // widget.cpp file , include Foo.h
struct Widget::Impl {
    Foo f1;
};
```

3. Pimpl Idiom is one of `std::unique_ptr` most common use cases. But just like using raw pointer, you can't avoid defining destructor even you have used `std::unique_ptr`.

```
1. // widget.h file
2. class Widget {
3.     Widget::~Widget()
4. }
```

```

5. | private:
6. |   struct Impl; // Forward declaration.
7. |   std::unique_ptr<Impl> pImpl;
8. | };
9.
10. // widget.cpp file
11. #include "Foo.h"
12. struct Widget::Impl{
13.     Foo fo;
14. }
15. Widget::~Widget() = default;

```

**Line 3:** If you don't declare here, compiler will produce by itself, In the end, it will call delete struct\* pIm and struct\* pIm is a incomplete type(definition is still invisible)

**Line 15:** here, Widget's destructor can see the whole Impl definition, so delete struct\* pIm is legal now.

**Source code:** The more explanation can be seen in "effective modern C++ item 22".

### 1.2.3 Linkage

#### 1.2.3.1 ODR

- For C language, there is tentative definition rule. You can define the same variable in two different .c file. The result will be undefined. But in the C++, this is not allowed any more.

```

a.c
int g_i = 100;
///////////////////////////////
b.c
int g_i;
fun(){
    printf("%d", g_i) // will print 100
}

```

**gcc:** gcc a.c b.c report no error. in b.c, if you write g\_i= 2; gcc

**g++:** report error.

- For C++ language, tentative definition of variable is not allowed. At the same time, multi-definition of function is not allowed either. but there are another implicit risk as below.
- In the same unit, you can't define class C1 again, but if you put two class C1 in two different .cpp file. compiler will not complain at all. When you run your application probably crash, it's dangerous. It's a little different with function and global variable. Because function and global variable all need allocation memory.
- Either a name is for everyone (and declared in a header file) or is translation-unit-local in an anonymous namespace. Detail can be found in "The One-Definition Rule Andrzej's C++ blog".

#### 1.2.3.2 external and internal linkage

- Duration and scope are two different conceptions in C++. there are three kinds of duration: **automatic, static and dynamic**. There are four kinds of scopes:

1. global.

- 2. In C++, we can use namespace to add more scopes to divide global scope.
- 3. file(translation unit).
- 4. local, function local and class local.
- Scope is a property handled by compiler, whereas linkage is a property handled by linker. There are two Linkages: Internal linkage and External linkage. Internal linkage refers to everything only in scope of a translation unit. External linkage refers to things that exist beyond a particular translation unit. In other words, accessible through the whole program, which is the combination of all translation units.
- Linkage refers only to elements that have addresses at link/load time; thus, class declarations and local variables have no linkage. Only global scope variable or function definition has external or internal linkage.
- A few examples:
  1. non-const global variable has external linkage.
  2. Const global variables have internal linkage by default.
  3. Functions have external linkage by default.
  4. static global variable has internal linkage.
  5. **static function has internal linkage too.**
- **const variables internally link by default** unless otherwise declared as **extern**. It means that:

**Many copies:** You can put `const int g_num = 10;` into a header file or global.h file. Then when you need `g_num`, just include this header file into your .cpp and it will not cause redefine linkage error.

**One copy, global access:** You also can put `const int g_num = 10;` in one .cpp file. then declare `extern const int g_num;` in global.h file.

**One copy, local access:** For const used just in one .cpp, use static and put it in the .cpp file.

- When a definition has internal linkage, it means that:
  1. you can put two same static variable name in two different .cpp file, no linkage error. (Each definition will have its own memory, **do don't apply it on the large object.**)
  2. You can't access internal linkage definition from another .cpp file.
  3. You can't use `extern` with `static`, but you can use `extern` with `const`.

```
//file.h:
extern const int a_global_var;

//file.c:
#include "file.h"
const int a_global_var = /*const expression */;
```

### 1.2.3.3 Inline function

- Usage of inline keyword is very simple: **No matter for member or non-member inline function, put it into the header file.**

1. For non member function, put inline function into a header file, and when you need to use this function, include it.
2. For member function, you have two options.

```
class Foo { //option 1
public:
void method() {...}; //Give definition here
};

class Foo { //option 2
public:
void method(); //Don't put inline keyword here
};
inline void Foo::method() { //Put inline keyword here
...
}
```

- Why do we need follow these two rules about inline function? The explanation is a little complex, so if you don't want go deeper, you can skip now. If you are confident and like facing challenge, lets continue.
- **Inline function has external linkage.** We demonstrate this point by below code. We have a.cpp and b.cpp two files.

```
1. inline int foo() { //File a.cpp
2.     return 6;
3. }
4. void g() {
5.     printf("foo called from g: return value=%d, address=%p\n", foo(), &foo);
6. }
7.
8. inline int foo() { //File b.cpp
9.     return 12;
10. }
11. void g();
12. int main() {
13.     printf("foo called from main: return value=%d, address=%p\n", foo(), &foo);
14.     g();
15. }
```

**Line 1 and 8 without inline:** it will trigger multi-definition linkage error

**Line 1 and 8 one inline:** Only one inline, because inline has external linkage, so it will trigger multi-definition linkage error too.

**Line 1 and 8 two inline:** Redefining an inline function with the same name but with a different function body is illegal; however, the compiler does not flag this as an error, but simply generates a function body for the version defined in the first file entered on the compilation command line, and discards the others. Therefore, may not produce the expected results.

**Delete line 8 to 10:** Compiling error, identifier doesn't found.

**Source code:** Inline just suppress multi-definition error. It is programmer's responsibility to ensure that inline function definitions with the same name match exactly across translation units, to avoid all above bad result. You can see the best way is:**put inline function into header file**

- Inline functions are defined in the header because, in order to inline a function call, the compiler must be able to see the function body. For a naive compiler to do that, the function body must be in the same translation unit as the call. (A modern compiler can optimize across translation units, and so a function call may be inlined even though the function definition is in a separate translation unit, but these optimizations are expensive, aren't always enabled, and weren't always supported by the compiler)
- functions defined in the header must be marked inline because otherwise, every translation unit which includes the header will contain a definition of the function, and the linker will complain about multiple definitions (a violation of the One Definition Rule). The inline keyword suppresses this, allowing multiple translation units to contain (identical) definitions.
- In the end. No matter how you designate a function as inline, it is a request that the compiler is allowed to ignore: the compiler might inline-expand some, all, or none of the places where you call a function designated as inline. (Don't get discouraged if that seems hopelessly vague. The flexibility of the above is actually a huge advantage: it lets the compiler treat large functions differently from small ones, plus it lets the compiler generate code that is easy to debug if you select the right compiler options.)

#### 1.2.4 Combine C and C++

- C++ inherits basic data type, variable name, statement, expression, and operator, control flow, function, file, head file and library, array, pointer and structure from C language. C++ is superset of C, so any C programs can be compiled by C++.
- When you use `g++`, `__cplusplus` will be defined automatically. (you can't undef it in fact.) When you use C compiler, such as `gcc`, `__cplusplus` is not defined. At the same time, When you use C++ compiler, such as `g++` to compile a C file, although file extension is `.c`, but `g++` still use name mangling to change function name. The conclusion is based on `g++` and `gcc` on Linux system. **compiler will decide if `__cplusplus` is defined, not based on source file name extension**
- The C++ compiler must be used to compile `main()`, and must be used to direct the linking process. **Most of time, you want your C++ application to call some existing C functions**
- If you have c and cpp source files together, you can just use `g++` compile them all. You don't need any `__cplusplus` syntax. `g++` compiles all files using name mangling. (look them all as `c++` files). At this time file extension doesn't play a role at all.
- If your C++ file want to use a c function. You don't have C function source code(It is in a lib or obj file) or you don't want to recompile it( it's a very big C library). At this time, you have three options:
  1. You can put function declaration in to `extern "C"` directly.
  2. You can put a head file into the `extern "C"`.

3. If you can control the header file, you can use `__cplusplus + extern "C"`. it will used both in C and C++ compiler.

```

1   extern "C" { // method 1
2     c_function(int);
3   }
4
5   extern "C" { // method 2
6     #include "old_C_header.h"
7   }
8
9   #ifdef __cplusplus //method 3
10  extern "C" {
11  }
12  Foo (int a, int b);
13  #ifdef __cplusplus
14  }
15  #endif

```

- If you define a function in .cpp file(You have to use g++ to compile it), and this function will be used in legacy C system, you need to use `__cplusplus + extern "C"`. You can give lib and head file to C system, and then the C system can include head file and linked to lib.
- **In one word, if you have obj code produced by C or C++, When you want to linked it to different language, you should consider using `__cplusplus + extern "C"`**
- Can a C function directly access data in an object of a C++ Class. Yes, but with some restriction. C++ class has no virtual base and virtual function. no access control. If you just want to pass a object from or to C function, you can refer a article in "C++ FAQ, 36.05". It demonstrate how to pass object from main to cppCallingC (C++ to C), then call cCallingC++(C to C++). Pay attention to points:
  1. We pass the class pointer.
  2. we use the same header file, but use `#ifdef __cplusplus` to defines one class(used by C++) and one struct(used by C), and they have the same name.
- There are three occasions which you need to use `extern "C"`
  1. When you want to produce a DLL or SO. Why, because maybe your DLL or SO will be used in both C language and C++ language. Or different compiler which uses different name mangling rule.
  2. When the code will be used by java or python.
  3. When used with legacy C code.



# Chapter 2

## type, operator and expression

### 2.1 Numerical

#### 2.1.1 Numerical

##### 2.1.1.1 Numerical Overflow

- Integer type has **overflow** problem, and float has **precision** problem. So prefer to use **long long** and **double** as your numerical type. In modern type, memory usage is not big concern, but it can save you a lot of trouble.
- In C and C++, you can use limits.h or <limits> to get the all the type limit information.

```
1. INT_MAX //use in C
2. INT_MIN
3.
4. //use in C++
5. cout<< std :: numeric_limits<int>::lowest() << '\t'
6. cout<< std :: numeric_limits<int>::max() << '\n';
```

- For integer addition or subtraction, It just a round-trip. When you reach a position in the circle, how to understanding depends on its context and type.

```
1. unsigned int ui = 1;
2. int i = -2;
3. // i+ui will stop in one position in the round clock.
4. //how to interpret it depends on the programme context;
5. int j = i+ui; // as int interpret this position CORRECT
6.
7. (ui+i)<6 // as unsigned interpret this position. ERROR!
8. //Most unsigned implicit cast error happen
9. //when you compare with a constant number.
```

- There are three ways to deal with overflow:

1. Build your own template function.

```
1. template <class T>
2. void increment_without_wraparound(T& value) {
3.     if (value < numeric_limits<T>::max())
4.         value++;
5. }
```

2. Judge it before calculation.

```

1. if ((x > 0) && (a > INT_MAX - x)) /* 'a + x' would overflow */;
2. // a is point, x>0 clockwise turn, then it will overflow
3.
4. if ((x < 0) && (a < INT_MIN - x)) /* 'a + x' would underflow */;
5. // a is point, x<0 anti-clockwise turn, so it will underflow
6. // It's easy to understand if you draw a clock figure.
7.
8. if ((x < 0) && (a > INT_MAX + x)) /* 'a - x' would overflow */;
9. if ((x > 0) && (a < INT_MIN + x)) /* 'a - x' would underflow */;
10.
11. if (a > INT_MAX / x) /* 'a * x' would overflow */;
12. if ((a < INT_MIN / x)) /* 'a * x' would underflow */;
13. // need to check for -1 for two's complement machines
14. if ((a == -1) && (x == INT_MIN)) /* 'a * x' overflow */
15. if ((x == -1) && (a == INT_MIN)) /* 'a * x' (or 'a / x') overflow */

```

3. Judge it after calculation

```

1. uint32 a, b;
2. // assign values
3. uint32 result = a + b;
4. if (result < a) {
5. // Overflow
6. }

```

- There's no simple, general, portable way to avoid integer overflow.
- You cannot safely check whether a signed integer addition or subtraction overflowed after the fact. An overflow in signed arithmetic causes undefined behavior. Typically the result wraps around, but in principle your program could crash before you have a chance to examine the result.
- Clang 3.4+ and GCC 5+ offer checked arithmetic builtins. They offer a very fast solution to this problem, especially when compared to bit-testing safety checks.

```

1. unsigned long b, c, c_test;
2. if (_builtin_umull_overflow(b, c, &c_test)){
3. // returned non-zero: there has been an overflow
4. }

```

- When you do some calculation, you can have some tricks to avoid overflow.  $n!$  the last three digit. You need use mod to keep last two digits in each calculation.

```

1. (a+b)/2 //a+b maybe overflow
2. a/2+b/2 +(a&b&1);

```

### 2.1.2 Numerical conversions

- Type conversions happen in three contexts:
  1. Assign a value of one **arithmetic type** to a variable of another arithmetic type.
  2. Combine mixed types in expressions.
  3. Pass arguments to or return from a function.
- In an expression, C++ makes two kinds of automatic conversion.

1. Some type are automatically converted whenever they occur. For example, when you add char to char. Detail can be seen in the promotion section below.
  2. Some type are converted when they are combined with other types in an expression. When an operation involves two types, the smaller is converted to the larger. For example, when you add an int to a float, int is converted to float type. (You have to do it, because two types have the different inside binary representations.)
- There are two kinds of conversion, one is **implicit**, and the other is **explicit**.
  - Assigned to a bool, zero converts to false, and nonzero converts to true.
  - Assigning a value to a type with a greater range usually poses no problem. If shorter range or different type, maybe there are some problems.
  - When conversion, maybe lost precision(double -> float, long long ->float) loss fragment(float -> i) or Undefine (int i = 666, then char c = i;). Detail example can be seen in my evernote bookmark.

```

1. int i, float f;
2. i=f;
3. //1) fragment will be lost, f= 3.99, i will be 3 (not rounding)
4. //2) If f is too big. undefined behave
5.
6. f = i;
7. // 1) will lost precision if i is big.

```

- A implicit conversion will happen when you call a function.(Just like you use assignment operator=). Below they all compile successfully. Compile with -Wconversion flag, it doesn't included in -Wall in g++; It just give warning when standard conversion happen. (no warning for promotion conversion).

```

1. bool isLucky(int number);
2.
3. isLucky('a') //i = 'a' , promotion NO warning
4. isLucky(false) //i = false , promotion NO warning
5. isLucky(1.2f) //i = f , standard conversion. Warning

```

- In C++, introduce braces initialization {}, It will not allow narrowing happen. But in g++, it just show a -Wnarrowing message, Anyway, I think that it's helpful.

```

1. int x = 66; char c1 = {x}; //ok
2.
3. int x = 666; char c2 = {x}; // not allowed;
4.
5. int fun(){
6.     return 1.2f; //OK, no warning
7.     return {1.2f}; //ERROR, -Wnarrowing
8. }

```

### 2.1.2.1 implicit promotion

- C++ converts bool, char, unsigned char, signed char and short to int. Because int type is generally chosen to be the computer's most natural type. **It does calculations faster for that type.** It's called **integer promotions**.

- unsigned short convert to int if short is shorter than int, if they have the same size. unsigned short convert to unsigned int. So no data loss in promoting.

```

1. char c1, c2, c //c1 and c2 convert to int first.
2. c = c1+c2; // then change int result back to char.
3.
4. /* LLVM IR code below
5. store i8 97, i8* %c1, align 1
6. store i8 2, i8* %c2, align 1
7. %0 = load i8, i8* %c1, align 1
8. %conv = sext i8 %0 to i32
9. %1 = load i8, i8* %c2, align 1
10. %conv1 = sext i8 %1 to i32
11. %add = add nsw i32 %conv, %conv1
12. %conv2 = trunc i32 %add to i8
13. */
14.
15. i+f // i will promoted to f and value keep the same.
16.
17. float f1, f2, f
18. f = f1+f2 // whether f1 change to double depends on compiler
19. //clang++ has fadd in LLVM IR, so it doesn't change f to double.

```

### 2.1.2.2 Explicit numerical conversion

- In order to suppress conversion warning, you can use explicit numerical conversion to state your intention clearly and loudly.
- In C language, there exist two main syntax for generic type-casting: functional and C-style cast.  
**Prefer C-style cast**

```

1. double x = 10.3;
2. int y;
3. unsigned int n1 = (unsigned int)f; // C-style cast
4. unsigned int n2 = unsigned(f); // functional cast
5. //functional cast only be used in one word type.
6. //unsigned int (f) is not right,
7. //int *(f) is not right either.

```

- In C++ language, **You should always use static\_cast**.

```

1. float f = 3.5;
2. int a = f; // this is how you do in C
3. int b = static_cast<int>(f); //in C++

```

## 2.2 type

### 2.2.1 Type cast in c++

#### 2.2.1.1 Implicit conversion in C++

- Class implicit conversion can be happen when it has:

1. Single ctor, it means that a class can be produced from something.
2. operator Type, it mans that a class can be converted to somethong.

```

1. class A {
2. A(int i); // bad
3. operator const char*(); // bad
4. };
5.
6. A a1, a2;
7. a2 = a1*2; // implicit conversion 2 to temp A obj
8. a2 = 2 the same, then call operator =;

```

- Implicit conversion can be called by compiler implicit, (means that you don't know at all). It sometimes will lead to potential ambiguity problem.

```

1. class A{
2. A(class B&);
3. };
4. class B{
5. operator A()
6. };
7.
8. void g(const A&);
9. B b;
10. g(b)
11. // it can call A's ctor in class A
12. // or it can call B's opearator A() in class B
13. //compiler will stop, it meet ambiguity.

```

more detail can be seen effective c++ item 26.

- You should always avoid implicit conversion**

- use explicit before single parameter ctor.
- use name convert function instead of "operator Type". so string has function `c_str()` instead operator `char*() const`.

- In an example below, with explicit keyword before ctor, you have to use `A(2)` or `(A)2` to explicitly build a A temporary obj in `a1*A(2)` expression. It's a good habit and you should stick to it.

```

1. class A {
2. explicit A(int i); // good
3. const char* getInternalPoint(); //good, use a name function .
4. };

```

- convert class to basic type, you need operator `basicTypeName`, no argument, no return value and member function.
- For class, assignment operator() can support two different type assignment. But I don't think that it is a conversion. I didn't see any practical usage by now.

```

1. class A {};
2.
3. class B {
4. public:
5. // conversion from A (assignment):
6. B& operator= (const A& x) {return *this;}
7. };
8.
9. void fun(B x){};

```

```

10. B b;
11. b = a; // calls constructor

```

### 2.2.1.2 type cast operator

- There are three operators, `dynamic_cast`, `const_cast` and `static_cast`. You should always use them in C++.
- Unrestricted explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. (syntax and compiling is right, but cause run-time error). In order to overcome this problem, in C++ language, we introduce `c++ cast operator`, see below section.

```

1. char c = 10; // 1 byte
2. int *p = (int*)&c; // compile ok
3. *p = 5 //runtime error, stack corruption.
4.
5. int *p = static_cast<int*>(&c) comple error;

```

- You don't need any cast operator when you change any type pointer to `void*`, But when you want to use dereference `void pointer`, you'd better use `static_cast` to change it back to a certain type pointer.

```

1. //In C++, you have to use cast operator.
2. int* p = static_cast<int*>(malloc(sizeof(*p)));
3. //A better way is to use new instead.

```

- `static_cast<type-name>` expression will be valid only if type-name can be converted implicitly to the same type that expression has. It will stop you from change a bird class to an apple class which are two totally unrelated classes. Even change `int` to `double`, encourage you to use `static_cast<double>(i)`. it also can help you to find cast easily in your source code by search "static\_cast".
- Changing the value of an `const` object through `const_cast` pointers leads to "undefined behavior". Most often you can. But for `const` static data – the compiler may put such variables in a read-only region, the program will crash if you try to modify it.

```

1. const int a = 12;
2. int* p = const_cast<int*>(&a);
3.

```

- Using `const_cast` is not good design. Sometimes for a `const` member function, you have to use `const_cast` to change this pointer to modify a class member. If compiler support, always use "mutable" keyword. Only use `const_cast` if your compiler doesn't support mutable.
- Sometimes, For some legacy functions, You have a `const` object you want to pass to a function taking a non-`const` parameter, and you know the parameter won't be modified inside the function. The second condition is important, because it is always safe to cast away the constness of an object that will only be read, not written.

```

1. strlen( char* p );
2. const char* cp = "hello";
3.
4. strlen( const_cast<char*>(cp));

```

- `dynamic_cast` should only be used down-cast public inherited relationship. You can't use `dynamic_cast` when
  1. Not for private or protected inherited relationship.
  2. If a class doesn't have virtual function, you can't use `dynamic_cast` on this object.
- A child pointer can always be assigned to base pointer directly. (That is how polymorphic implement.) `dynamic_cast` use to **down-cast** a base pointer to child pointer. `dynamic_cast` assure that down cast is valid.
- If you frequently use `dynamic_cast`, It can be a sign that your base class offer too little functionality, you'd better to re-desing you base class API(adding more member function to base class.)
- `dynamic_cast` can also used in reference type. When cast fail, it will not return `nullptr`, (because it's reference), just throw a `bad_cast` exception.

```

1. struct A {};
2. struct D : public A {};
3. /////////////////
4. D d; // the most derived object
5. A& a = d; // upcast, dynamic_cast may be used, but unnecessary
6. D& new_d = dynamic_cast<D&>(a); // downcast

```

- `dynamic_cast` can also be used in **side-cast** in multi inheritance.

```

1. struct V {
2.     virtual void f() {};
3.     // must be polymorphic to use runtime-checked dynamic_cast
4. };
5. struct A : virtual V {};
6. struct B : virtual V {};
7. struct D : A, B {};
8.
9. D d; // the most derived object
10. A& a = d; // upcast, dynamic_cast may be used, but unnecessary
11. B& new_b = dynamic_cast<B&>(a); // sidecast
12. // Change a to d first (from parent to child, maybe fail),
13. // then d to new_b, (from child to parent, always succeed)

```

- `dynamic_cast` different with `static_cast`:

1. `static_cast` check on compile time.
2. `static_cast` no run time information, so sometimes it makes mistake.

```

1. struct V {
2.     virtual void f() {};
3.     // must be polymorphic to use runtime-checked dynamic_cast
4. };
5. struct A : virtual V {};
6. struct B : virtual V {};
7. A a;
8. V& v = a;
9.
10. B& b = static_cast<B&>(v); // ok
11. B& b = dynamic_cast<B&>(v); // not ok

```

- About `dynamic_cast`, "exceptional C++" item 44 give a good question and answer.
- `reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked. It can also cast pointers to or from integer types. **DON'T USE IT UNLESS YOU ARE IN THE CORNER.**

### 2.2.1.3 RTTI

- It's a relatively new conception in C++, you should avoid using it on old C++ compiler. It includes two methods: `typeid`, `dynamic_cast`.
- **It only work with class hierarchy that has virtual functions.**
- There are two kinds of types (for the purposes of RTTI): polymorphic types and non-polymorphic types. A polymorphic type is a type that has a virtual function, in itself or inherited from a base class. A non-polymorphic type is everything else; this includes POD types, but it includes many other types too.
- `typeid` operator will return a `type_info` class. You need to include `typeinfo.h` head file. `Typeid` operator receive pointer or class name.

```

1. int myint = 50;
2. std::string mystr = "string";
3. double *mydoubleptr = nullptr;
4.
5. cout << "myint_has_type:" << typeid(myint).name();
6. cout << "mystr_has_type:" << typeid(mystr).name();
7. cout << "mydoubleptr_has_type:" << typeid(mydoubleptr).name();
8. typeid is operator, it return type_info class
9. .name is member function of type_info

```

- If you just want to assure up casting and you don't want to know more about the class, you should prefer to use `dynamic_cast`. just know `typeid` when you have a more complicated demand, you can come back to take a look deeply.
- RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unitests, but avoid it when possible in other code. consider one of the following alternatives to querying the type:
  1. Virtual methods are the preferred way of executing different code paths depending on a specific subclass type. This puts the work within the object itself.
  2. If the work belongs outside the object and instead in some processing code, consider a double-dispatch solution, such as the Visitor design pattern. This allows a facility outside the object itself to determine the type of class using the built-in type system.
- Just like exception, It has some loss in performance, you can use flag `"-fno-rtti"` to turn off it.

### 2.2.2 cv-qualifier

#### 2.2.2.1 const in function

- Just like assert statement, Use `const` aggressively.

- `const` must be initialized when you declare it. `static` will be initialized to default value (usually zero value) if you don't set value manually.
- Top-level `const` to indicate that the pointer itself is a const. When a pointer can point to a const object, we refer to that `const` as a low level const.
- **For reference type, you always can't change it, so top-level const is default, you only can set low-level const.**

```

1. int i = 0;
2. int * const p1 = &i; // const is top-level
3. const int *p2 = &i; // const is low-level
4.
5. const int *const p3 = p2;
6. // right most const is top-level, left-most is low-level
7. const int &r = i;
8. // const in reference type is always low-level

```

- `const` mainly used in three places inside of function:

1. functions parameter.
2. function return. ( just used for `const` reference, not for return value)
3. member function.

- In fact, for value type parameter, the function signature is the same whether you include this `const` in front of a value parameter or not. It will cause redefine error. For example:

```

1. int f( int );
2. int f( const int ); // redeclares f(int)
3. // no overloading, there's only one function.

```

- Consider not writing `const` on pass-by-value function parameters when only forward-declaring a function. You can always add it on the definition to express a read-only parameter.

```

1. // value parameter: top-level const is not part of function signature
2. int f( int );
3. int f( const int ); // redeclares f(int): this is the same function
4.
5. // non-value parameter: top-level const is part of function signature
6. int g( int& );
7. int g( const int& ); // overloads g(int&): these are two functions

```

- For `const` value type parameter in the example below, the `const` qualifier prevents code inside the function from modifying the parameter itself. Such an assurance helps you to quickly read and understand a function. Under some circumstances, this might even help the compiler generate better code.

```

1. double cube (const double side){
2.     return side * side * side;
3. }

```

- **In C++, When you use pointer or reference as function parameter, You should always put const in front of it. Because 90% you don't need to modify it.** Once you compiler bark, then you can delete `const`, It will help you to use `const` aggressively.

- When function return build-in value, don't use const at return value at all.

```

1. const int foo() {
2.     return 3;
3. }
4. int x = foo(); // copies happily
5. foo() = 4; // not valid anyway for built-in types

```

- When function return user-defined type value, don't use const at all. It will not allow you to use rvalue and move.

```

1. const time operator+(const time &t) {
2.     time temp;
3.     return temp.bla = bla+t.bla;
4. }
5.
6. //because + return const, so below just copy
7. //not use time move ctor, low efficient.
8. time t3(t1+t2);

```

- When function return reference or pointer, you can use const to restrict modify it.

```

1. class String{
2.     const char& operator[](int position);
3. }

```

- Most of time, only index operator [] , assignment operator and <<, >> support return reference. For assignment operator, we only need to return &. For istream and ostream overload, you don't need return const at all.

```

1. Array &Array::operator=(const Array &right) {
2.     ...
3.     return *this; //enables x=y
4. }

```

- Common function interface: **you can see const only used in pointer or reference type. You should use more const in your projects.**

type	read	write
primitive (char, int, float)	pass value	pointer or reference
class, array, structure	const pointer or reference	pointer or reference

### 2.2.2.2 const in class

- When **const** used in class. There are four points.

1. In **const** member function, you can't change member variable,(If you change it, it will report a compile error).

```

1. time operator+(const time &t1) const{
2.     this->m_a = 100 // will compile error
3. }

```

2. Const obj can only call **const** function. But non-const obj can call ALL funcitons( const or non-const).

```

1. class Fred {
2. public:
3.     void inspect() const;
4.     // This member promises NOT to change *this
5.     void mutate();
6.     // This member function might change *this
7. };
8.
9. void userCode(Fred& changeable, const Fred& unchangeable) {
10.    changeable.inspect();      // Okay: doesn't change
11.    changeable.mutate();       // Okay: changes
12.    unchangeable.inspect();   // Okay: doesn't change
13.    unchangeable.mutate();    // // ERROR:
14. }
```

3. If you want to return a member of a object from a const method, you should return it using reference-to-const (const X& inspect() const) or by value ( X inspect() const). **const member method return value or const reference.**

```

1. class Person {
2. public:
3.     const std::string& name_good() const;
4.     // Right: the caller can't change the Person's name
5.
6.     std::string& name_evil() const;
7.     // Wrong: the caller can change the Person's name
8.
9.     int age() const;
10.    // Also right: the caller can't change the Person's age
11.    //
12. };
13.
14. void myCode(const Person& p){
15.     // const p means not to change the Person object
16.     p.name_evil() = "Igor"; // But changed....!
17. }
```

4. The most common use of **const** overloading is with the subscript operator. You should generally try to use one of the standard container templates, such as std::vector, but if you need to create your own class that has a subscript operator, here's the rule of thumb: **subscript operators often come in pairs.**

```

1. class Fred { /*...*/ };
2. class MyFredList {
3. public:
4.     const Fred& operator[] (unsigned index) const;
5.     // Subscript operators often come in pairs
6.
7.     Fred& operator[] (unsigned index);
8.     // Subscript operators often come in pairs
9. }
```

- If your member functions doesn't change member variable, put **const** after the function, it will make **const obj** can invoke this function.
- Do not use "volatile" except in low-level(embedded c) code that deals directly with hardware. **1) don't optimized code, 2) each time you read volatile, load from memory instead of using old value.**

```

1. void waitForSemaphore() {
2.     volatile uint16_t * semPtr = WELL_KNOWN_SEM_ADDR;
3.     /* well known address to my semaphore */
4.     while ((*semPtr) != IS_OK_FOR_ME_TO_PROCEED);
5. }
```

- "mutable" allows you to modify a member variable in a class by a const method. Why do we need this conception? Behind "mutable", It's bitwise const and logical const. Logical const is when an object doesn't change in a way that is visible through the public interface. An example would be a class that computes a value the first time if required, then caches the result.

```

1. class TextBook {
2. private:
3. mutable int length_;
4. mutable bool isValid;
5. public:
6. void getLength() const {
7. if(isValid == false){
8. length_ = strlen(*p);
9. isValid = true;
10. }
11. else return length_
12. };
```

- In above example, `getLength()` is member-wise const function, It just read a length value, not set it from outside. But inside this function, you need to change some private member value, At this time, you need to use mutable keyword, so `const getLength()` function can modify and cache a `length_` value.
- const function just put restraint inside of function, it not ask caller to be a const object at all. Why I declare `getLength()` as const?
  1. because if you don't do it. const obj can't call `getLength()` function at all.
  2. "logically", we just read, not change something.
- **A const obj only can call const member function, and const member function just return const reference or pointer.** A example can be seen in vector example.

```

1. iterator begin() noexcept;
2. const_iterator begin() const noexcept;
3.
4. const vector<int> cvi;
5. vector<int>::iterator vi = cvi.begin(); //Compile error
6. //cvi will call the second overload function anyway.
7. //but second just return const_iterator.
8. //and const_iterator can NOT be converted to iterator implicitly.
```

- Continue with previous item. **If you define a const obj or you have const member function, all the member variable of const obj or inside const member function is const.** For example, because `getArea` is const funciton, when it is called by a const obj, points will become a const member variable, so error will happen below:

```

1. class A{
2. getArea() const{
3. vector<int>::iterator vi = points.begin();
```

```

4. // because getArea is const
5. // so points are const implicit
6. // you have to use const_iterator here
7. }

8.
9. calArea() {
10.     vector<int>::iterator vi = points.begin();
11. }
12. vector<int> points
13. };

14.
15. const A ca;
16. aa.calArea(); // not compile
17.
18. A a;
19. a.calArea(); // obj can acces non-const member fun
20. a.getArea(); // obj can also access const member fun.

```

- **(const) expand outside, and restraint inside**

1. If you declare getArea() const function, it will make both const obj and non-const obj can call this member function. This is good. if you don't define it as const member function, only non-const obj can invoke this function. **For outside, const increase interface applicable scope.**
2. But inside const member function, You can't change member variable anymore, (if you really want, use mutable keyword.) Inside const member function, it will think all member variable const, so vector will be const implicitly, return non-const iterator will be error.

- In order to resolve this problem, you have two options: one is use const\_iterator.

```

1. class A{
2.     getArea() const{
3.         vector<int>::const_iterator vi = points.begin();
4.     }
5.     .....

```

- another method is using auto

```

1. class A{
2.     getArea() const{
3.         auto vi = points.begin();
4.     }
5.     .....

```

- A good reference article about const is GotW 6.
- when you declare mutable, you'd better use mutex to synchronize it. It's called M&M rule in "GotW #6b Solution"

### 2.2.2.3 const\_iterator

- You should know the difference between `vector<int>::const_iterator` and `vector<int>::iterator`

```

1. vector<int>::const\_iterator cvi;
2. *cvi = 12 //ERROR, can't change
3. cvi++ //OK
4.
5. const vector<int>::iterator vi;
6. *vi = 12 //OK
7. vi++ //ERROR

```

- In C++11/14, we add cbegin for STL container and interface of container has changed from non-const iterator to const iterator, so we should use const iterator more.

```

1. iterator insert (iterator position, const value_type& val); //c++98
2.
3. iterator insert (const_iterator position, const value_type& val); //c++11
4.
5. vector<int> values;
6. auto it = std::find(values.cbegin(), values.cend(), 1983); // and cend
7. values.insert(it, 1998);

```

- Another good article is "effective modern C++" item 13.

## 2.2.3 constexpr

### 2.2.3.1 constant expression

- Unlike templates and preprocessor macros, constexpr allows for loops and recursion at compile-time without extreme boilerplate.
- It can be used in places that require compile-time evaluation, for example, template parameters and array-size specifiers.
- It must and can be calculated at compiling time.
- Difference constant expression and constexpr:
  1. Declaring something as constexpr does not necessarily guarantee that it will be evaluated at compile time. It can be used for such, but it can be used in other places that are evaluated at run-time, as well.
  2. An object may be fit for use in constant expressions without being declared constexpr. for example: `const int N = 25.`
  3. constexpr is specifier
- In fact, constexpr is a lot of different with const. constexpr can be used in three places: constexpr function can be thought as a kind of "metaprogram".
  1. constexpr variable.
  2. constexpr function.
  3. constexpr obj.
- constexpr has two advantages:
  1. Improve efficiencies(if it's possible, it will calculate at compile time, not run time).
  2. expand usage scope (initialize constexpr obj and use in integer constexpr context)

```

1. constexpr int fun(int a, int b){return a+b;}
2.
3. constexpr int const foo = fun(2,3);
   // calculate at compile time
4.
5.
6. int a[fun(2,3)];
   // without constexpr in function declaration,
7. // the next two statements can't be compiled.
8.
```

### 2.2.3.2 constexpr variable

- All **constexpr** objects are **const**, but not all **const** object are **constexpr**

1. **constexpr** variable is implicitly **const**. Just like **const**, **constexpr** can't be changed.
2. must be initialized when you declare it. the full-expression of its initialization, including all implicit conversions, constructors calls, etc, must be a constant expression.(You know the exact value in compiling time.)
3. Value of **constexpr** **must be known at compile time**.If you want to get value from a function, you have to use **constexpr** function to assign value to it.

```

1. constexpr int sum(int i, int j){
2.     return i+j;
3. }
4.
5. int i, j;
6. cin>>i>>j;
7. const int result = i+j; //OK
8. constexpr int result = i+j;
9. //ERROR, Can't calculated in comiliing time
10.
11. int ce = sum(i, j); //OK
12. constexpr ce = sum(i, j);
13. //ERROR, Can't calculated in comiliing time
```

4. when you declare **constexpr** obj, it implicit **const**. It's just used in variable, When you want to use **const int \* const p**, you'd better write you own **const**

```

1. constexpr int const foo = 42;
2. constexpr int foo = 42;      // same as previous
3. constexpr int const *pb = &bar; // &bar must be constexpr
```

- For **constexpr** object, it must has corresponding **constexpr** constructor

1. can only be invoked with constant expressions.
2. can not use exception handling.
3. has to be declared as default or delete or the function body must be empty (C++11).
4. The **constexpr** user-defined type

- For class itself, it also need satisfy below:

1. can not have virtual base classes.
2. requires that each base object and each non-static member has to be initialized in the initialization list of the constructor or directly in the class body.

3. Consequently, it holds that each used constructor (e.g. of a base class) has to be `constexpr` constructor and that the applied initializers have to be constant expressions.

```

1. class MyInt{
2. public:
3.     constexpr MyInt()= default;
4.     constexpr MyInt(int fir, int sec): myVal1(fir), myVal2(sec){}
5.     MyInt(int i){ myVal1= i-2; myVal2= i+3; }
6.
7.     constexpr MyInt(const MyInt& oth)= default;
8.     constexpr MyInt(MyInt&& oth)= delete;
9.     constexpr int getSum(){ return myVal1+myVal2; }
10.
11. private:
12.     int myVal1= 1998;
13.     int myVal2= 2003;
14. };
15.
16. constexpr MyInt myIntConst1;
17. MyInt myInt2;
18.
19. constexpr int sec= 2014;
20. constexpr MyInt myIntConst3(2011,sec);
21. std::cout << myIntConst3.getSum();
22.
23. int a= 1998, b= 2003;
24. MyInt myInt4(a,b);
25. std::cout << "myInt4.getSum(): " << myInt4.getSum();
26. std::cout << myInt4 << std::endl;
27.
28. // constexpr MyInt myIntConst5(2000); ERROR
29. MyInt myInt6(2000);
30. std::cout << "myInt6.getSum(): " << myInt6.getSum();
31.
32. // constexpr MyInt myInt7(myInt4); ERROR
33. constexpr MyInt myInt8(myIntConst3);
34.
35. int arr[myIntConst3.getSum()];
36. static_assert( myIntConst3.getSum() == 4025, "2011+014==4025" );

```

- A good article is:

<https://www.modernescpp.com/index.php/constexpr-variables-and-objects>

### 2.2.3.3 `constexpr` function

- `constexpr` functions can be used as regular functions, although internally they have greater restrictions.
- `constexpr` functions can easily be converted into regular functions as requirements change.
- `constexpr` functions compile much quicker than the equivalent template-based solutions, which scale linearly with the depth of the template-recursion.
- The basic idea is "compile time input yields compile time output". How can I know if a function is `constexpr` or not?
- `std::find` is `constexpr` function now.

- The primary usage of constexpr for a function is to declare intent. The intent is that you are sure that the function is almost a pure function. The pure function has no state, and when you run it, it has no side effect.
- The pure function has some advantages compared with ordinary function. Return always the same result when given the same arguments, So the function call can be replaced by the result, and the order of function call is not important.(That is why you can run it at compiling time.) But the problem is compiler itself can't judge if a function is pure function, That's why programmer come to rescue. The programmer indicate this function is a "pure" function, so compiler can **For outside** use it in another constant expression context, **For inside** calculate it in compiling time.
- On the first sight, you should declare constexpr function everywhere, but it's not good idea either. That makes a constexpr qualifier an irrevocable design decision. You cannot remove this qualifier without an incompatible change to your API. It also limits how you can implement that function, e.g. you would not be able to do any logging within this function. Not every trivial function will stay trivial in eternity. That means you should preferably use constexpr for functions that are inherently pure functions, and that would be actually useful at compile time (e.g. for template metaprogramming). It would not be good to make functions constexpr just because the current implementation happens to be constexpr-able.
- Where compile-time evaluation is not necessary, using inline functions or functions with internal linkage would seem more appropriate than constexpr. Both constexpr and inline are for performance improvements, inline functions are request to compiler to expand at compile time and save time of function call overheads. In inline functions, expressions are always evaluated at run time. constexpr is different, here expressions are evaluated at compile time.
- constexpr can be used with both member and non-member functions, as well as constructors. It declares the function fit for use in constant expressions.
- A few examples of constexpr functions.

1. you have something that can be evaluated down to a constant while maintaining good readability and allowing slightly more complex processing than just setting a constant to a number.

```
1. constexpr int MeaningOfLife ( int a, int b ) { return a * b; }
2. const int meaningOfLife = MeaningOfLife( 6, 7 );
```

2. It basically provides a good aid to maintainability as it becomes more obvious what you are doing. Take max( a, b ) for example: Its a pretty simple choice there but it does mean that if you call max with constant values it is explicitly calculated at compile time and not at runtime.

```
1. template< typename Type >
2. constexpr Type max(Type a, Type b) { return a < b ? b : a; }
```

3. Another good example would be a DegreesToRadians function. Everyone finds degrees easier to read than radians. While you may know that 180 degrees is in radians it is much clearer written as follows:

```
1. const float oneeighty = DegreesToRadians( 180.0f );
```

- Compared with ordinary function, constexpr functions:

1. **For outside.** is able to be used in the another constant expression(such as `constexpr int sum = cfun(2,3);`) But it doesn't mean that it must be used in constant expression(such as `int sum = cfun(i,j);`)
2. **For inside.** `constexpr` function is able to be calculated in compiling time. (such as `int sum = cfun(2,3);`) But it doesn't mean that it must be calcuated in compling time(such as `int sum = cfun(i,j);`)

```

1. constexpr int cfun(int i, int j){
2.     return i+j;
3. }
4.
5. int fun(int i, int j){
6.     return i+j;
7. }
8.
9. int i, j;
10. cin>>i>>j;
11. constexpr int sum = fun(2,3); //Error
12. constexpr int sum = fun(i,j); //Error
13. constexpr int sum = cfun(2,3); //OK
14. constexpr int sum = cfun(i,j); //Error
15. //For constexpr context, only cfun with constant expression is OK
16.
17. int sum = fun(2,3); //OK
18. int sum = fun(i,j); //OK
19. int sum = cfun(2,3); //OK
20. int sum = cfun(i,j); //OK
21. //For Non-constexpr context, all fun is OK

```

- There are two contexts in which a `constexpr` function **MUST** to run at compile time. That's why**constexpr functions must be able to return compile-time results when called with compile-time values.**
  1. The `constexpr` function is executed in a context which is evaluated at compile time. This can be a `static_assert` expression such as with the type-trait library or the initialisation of a C-array.
  2. The value of a `constexpr` function is requested during compile time with `constexpr`: `constexpr auto res = func(5);`
- In C++11. For `constexpr` functions there are a few restrictions:
  1. has to be non-virtual.
  2. has to have arguments and a return value of a literal type. Literal types are the types of `constexpr` variables.
- The restriction goes on with the function body. The key points are that
  1. it has to be defined with the keyword `default` or `delete` or
  2. can only have one return statement.
  3. The function body must be non-virtual and extremely simple: Apart from `typedefs` and `static asserts`, only a single return statement is allowed. In the case of a constructor, only an initialization list, `typedefs` and `static assert` are allowed. (= `default` and = `delete` are allowed, too, though.)

```

1. constexpr int gcd(int a, int b){
2.     return (b == 0) ? a : gcd(b, a % b);
3.     //can only have one return statement.
4.     //but thank for ternary operator and recursion.
5. }
```

- In C++14, constexpr function can include:

1. conditional jump instructions or loop instructions.
2. more than one instruction.
3. fundamental data types that have to be initialized with a constant expression.
4. **You can't define below:** asm declaration, a goto statement, a statement with a label other than case and default, try-block, definition of a variable of non-literal type, definition of a variable of static or thread storage duration, definition of a variable for which no initialization is performed.
5. The arguments and the return type must be **literal types** (i.e., generally speaking, very simple types, typically scalars or aggregates)
6. constexpr function can call only other constexpr function not simple function.
7. in C++11, constexpr member function is implicit const; C++14 lift it up.

```

1. constexpr auto gcd(int a, int b) {
2.     while (b != 0) {
3.         auto t = b;
4.         b = a % b;
5.         a = t;
6.     }
7.     return a;
8. }
```

- A good article is "Demystifying constexpr".

#### 2.2.4 static

- In C++, **global and static variables initialized to default values**. But auto variable is random value, unless you use value initialization, because of efficiency consideration.
- You can't initialize a static member variable inside the class declaration, you need to put it in a .cpp file. But you can if the static data member is **const** of integer or enumeration. If you have a const member inside a class, better to use const static, because it can't be changed, so all the object can share the one static value.

```

1. class{ //in .h file
2.     static int obj_num; // you can't initialize
3.     const static int months = 12; // you can initialize
4. };
5.
6. //In .cpp file
7. Int class::obj_num = 0; // no static keyword anymore.
8. // obj_num will be default value(0) even you don't init it.
```

- static member function has two usages:

1. It can be invoked just by class name, not object instance, so you can define math class and define a lot static math function inside it. Just like name space.
  2. It can't access class data member, only can access class static member data. Because for static function, we don't pass (this) pointer
- static can be used restrain the scope.

```

1. int global = 0; //All files
2. static int s_i = 50; //just in this file
3. main() {
4.     static int s_i = 100; //just in this block
5.     printf("%d %d", ::s_i, s_i); print 50 and 100
6.     //not conflict, but if you define
7.     //int s_i in global scope it will conflict.
8. }
```

- Summary: static uses in three ways:

1. use it inside a function. unvisible outside of function, valid until program end.
2. use it inside a class. only copy for all instances, and access by static member function.
3. use it inside a file. internal link, avoid name conflict. But by now, we prefer to use unname space in modern C++.

## 2.2.5 size\_t and ptrdiff\_t

### 2.2.5.1 unsigned int

- unsigned int + signed int just wrapped on the clock. not overflow, just turn around on the clock.

```

1. unsigned int ui = 0xffffffff // -2 or UNIT_MAX-1
2. int i = 1;
3. //ui+i will be expressed 0xffffffff in memory.
4. printf("%d", ui+i); //print -1
5. printf("%u", ui+i); //print UINT_MAX;
```

- When you 1) compare with other, 2) expand 3) multiply or sub, it will interpret according to its signed semantic.

```

1. unsigned int ui = -2 // or UNIT_MAX-1
2. int i = 1;
3. ui+i < 6 // greater than 6
4. (ui+i)/4 //a bit positive number
5. int* p;
6. p+(ui+i); //on 32 bits, this ok,
7. //but on 64 bits, ui+i will be promote to 64 bits first.
8. //at this time, it will promote according to unsigned int.
```

- You can see llvm, in this reference. 1) only i32, no ui32 2) only add. 3) but has umultiply and uge and sext or sext. That is to say, these three operations need interpret signed semantic differently.
- For p+(ui+i) questions, we can use size\_t and ptrdiff\_t type.

### 2.2.5.2 size\_t

- Why we need `size_t`? Semantically, it should be pointer type, but physically, it's a kind of integer, so when we use `int` or `short` or `long` to represent, it will cause cross-platform problem
- Type `size_t` is a typedef that's an alias for some unsigned integer type, typically `unsigned int` or `unsigned long`, but possibly even `unsigned long long`. Each Standard C implementation is supposed to choose the unsigned integer that's big enough—but no bigger than needed—to represent the size of the largest possible object on the target platform.
- Using `size_t` appropriately makes your source code a little more self-documenting. When you see an object declared as a `size_t`, you immediately know it represents a size in bytes or an index, rather than an error code or a general arithmetic value.
- The main reason of `size_t` is: size of something is dependent on pointer, but on different system, size of pointer is not same as size of `int`. textbf{But size of pointer is always same as `size_t`.
- The size of `size_t` and `ptrdiff_t` always coincide with the pointer's size. Because of this, it is these types which should be used as indexes for large arrays, for storage of pointers and, pointer arithmetic.
- `size_t` type is usually used for loop counters, array indexing, and address arithmetic.
- `ptrdiff_t` type is a base signed integer type of C/C++ language. The type's size is chosen so that it can store the maximum size of a theoretically possible array of any type. On a 32-bit system `ptrdiff_t` will take 32 bits, on a 64-bit one 64 bits.

```

1. int A = -2; // should use ptrdiff_t here
2. unsigned B = 1; // should use ptrdiff_t here.
3. int array[5] = { 1, 2, 3, 4, 5 };
4. int *ptr = array + 3;
5. ptr = ptr + (A + B); //Error
6. printf("%i\n", *ptr);

```

- In one word, `int` is not always same as bits of OS. but `size_t` and `ptrdiff_t` are always same.
- A good article is "Why `size_t` matters", another one is "About `size_t` and `ptrdiff_t`". just google them!

### 2.2.6 Aggregate and POD

#### 2.2.6.1 Aggregate

- The basic idea of Aggregate type is that you can use aggregate list initialization. All the detail definition of an aggregate can be traced back to this main idea.
- An aggregate is an array or a class with no user-declared constructors, no private or protected non-static data members, no base classes, and no virtual functions.
- An aggregate can have a not Aggregate1 data member.

```

1. class NotAggregate1{
2.     virtual void f() {} //remember? no virtual functions
3. };

```

```

4.
5. class NotAggregate2{
6. int x; //x is private by default and non-static
7. };
8.
9. class NotAggregate3{
10. public:
11. NotAggregate3(int) {} //oops, user-defined constructor
12. };
13.
14. class Aggregate1{
15. public:
16. NotAggregate1 member1; //ok, public member
17. Aggregate1& operator=(Aggregate1 const & rhs) {/* */}
18. //ok, copy-assignment
19. private:
20. void f() {} // ok, just a private function
21. };

```

- Now let's see how aggregates are special. They, unlike non-aggregate classes, can be initialized with curly braces {}.

```

1. struct X{
2. int i1;
3. int i2;
4. };
5.
6. struct Y{
7. char c;
8. X x;
9. int i[2];
10. float f;
11. protected:
12. static double d;
13. private:
14. void g() {}
15. };
16.
17. Y y = {'a', {10, 20}, {20, 30}};
18. //Even we don't assign value to f, f will use value init.

```

- In C++11, Previously, an aggregate could have no user-declared constructors, but now it can't have user-provided constructors. Is there a difference? Yes, there is, because now you can declare constructors and default them:

```

1. struct Aggregate {
2. Aggregate() = default;
3. // asks the compiler to generate the default implementation
4. };

```

- In C++11, Now an aggregate cannot have any brace-or-equal-initializers for non-static data members. What does this mean? Well, this is just because with this new standard, we can initialize members directly in the class like this:

```

1. struct NotAggregate {
2. int x = 5; // valid in C++11
3. std::vector<int> s{1,2,3}; // also valid
4. };

```

- Now that we know what's special about aggregates, let's try to understand the restrictions on classes; that is, why they are there. We should understand that memberwise initialization with braces implies that the class is nothing more than the sum of its members. If a user-defined constructor is present, it means that the user needs to do some extra work to initialize the members therefore brace initialization would be incorrect. If virtual functions are present, it means that the objects of this class have (on most implementations) a pointer to the so-called vtable of the class, which is set in the constructor, so brace-initialization would be insufficient. You could figure out the rest of the restrictions in a similar manner as an exercise :).

### 2.2.6.2 POD

- An aggregate class is called a POD if it has no user-defined copy-assignment operator and destructor and none of its nonstatic members is a non-POD class, array of non-POD, or a reference.

```

1. struct POD{
2.     int x;
3.     char y;
4.     void f() {} //no harm if there's a function
5.     static std::vector<char> v; //static members do not matter
6. };
7.
8. struct AggregateButNotPOD1{
9.     int x;
10.    ~AggregateButNotPOD1() {} //user-defined destructor
11. };
12.
13. struct AggregateButNotPOD2{
14.     AggregateButNotPOD1 arrOfNonPod[3]; //array of non-POD class
15. };

```

- POD-classes, POD-unions, scalar types, and arrays of such types are collectively called POD-types.
- POD-classes are the closest to C structs. Unlike them, PODs can have member functions and arbitrary static members, but neither of these two change the memory layout of the object. So if you want to write a more or less portable dynamic library that can be used from C and even .NET, you should try to make all your exported functions take and return only parameters of POD-types.
- For objects of POD types it is guaranteed by the standard that when you memcpy the contents of your object into an array of char or unsigned char, and then memcpy the contents back into your object, the object will hold its original value. Do note that there is no such guarantee for objects of non-POD types. Also, you can safely copy POD objects with memcpy. The following example assumes T is a POD-type:
- In C++ 11, The idea of a POD is to capture basically two distinct properties:
  - It supports static initialization,
  - Compiling a POD in C++ gives you the same memory layout as a struct compiled in C.
- Because of this, the definition has been split into two distinct concepts: trivial classes and standard-layout classes, because these are more useful than POD. The standard now rarely uses the term POD, preferring the more specific trivial and standard-layout concepts.

### 2.2.6.3 trivial class

- If this is a trivial class, then it has trivial ctor/dtor/copy/assignment, When we build copy or destruct these type, we don't call these trivial ctor/copy... but just call malloc() or memcpy() to improve efficiency.
- For example, std::copy use `std::is_trivially_copyable` as a flag to decide whether to call memcpy or call object copy ctor.
- Static initialization is initialization of some variable with a compile-time value such that the value ends up being "baked into" the executable image (no code needs to be actually run):

```

1. struct Foo {
2.     int x;
3.     int y;
4. };
5.
6. Foo foo = {0,1};

```

- Trivial is the first property mentioned above: trivial classes support static initialization. If a class is trivially copyable (a superset of trivial classes), it is ok to copy its representation over the place with things like memcpy and expect the result to be the same. The detail is not easy to explain, you can refer a good article in the end of section.

```

1. // empty classes are trivial
2. struct Trivial1 {};
3. // all special members are implicit
4. struct Trivial2 {
5.     int x;
6. };
7.
8. struct Trivial3 : Trivial2 { // base class is trivial
9.     Trivial3() = default; // not a user-provided ctor
10.    int y;
11. };
12. struct Trivial4 {
13.     public:
14.     int a;
15.     private: // no restrictions on access modifiers
16.     int b;
17. };
18. struct Trivial5 {
19.     Trivial1 a;
20.     Trivial2 b;
21.     Trivial3 c;
22.     Trivial4 d;
23. };
24. struct Trivial6 {
25.     Trivial2 a[23];
26. };
27. struct Trivial7 {
28.     Trivial6 c;
29.     void f(); // it's okay to have non-virtual functions
30. };
31. struct Trivial8 {
32.     int x;
33.     static NonTrivial1 y; // no restrictions on static members
34. };
35. struct Trivial9 {

```

```

36. Trivial9() = default; // not user-provided
37. // a regular constructor is okay because we still have default ctor
38. Trivial9(int x) : x(x) {};
39. int x;
40. };
41. struct NonTrivial1 : Trivial3 {
42.     virtual void f(); // virtual members make non-trivial ctors
43. };
44. struct NonTrivial2 {
45.     NonTrivial2() : z(42) {} // user-provided ctor
46.     int z;
47. };
48. struct NonTrivial3 {
49.     NonTrivial3(); // user-provided ctor
50.     int w;
51. };
52. NonTrivial3::NonTrivial3() = default;
53. // defaulted but not on first declaration, still counts as user-provided
54. struct NonTrivial5 {
55.     virtual ~NonTrivial5(); // virtual destructors are not trivial
56. };

```

- a class is trivially copyable (a superset of trivial classes), A trivial class is a class that has a trivial default constructor (12.1) and is trivially copyable.

```

1. template <class T>
2. void copy(T* source, T* destination, int n, trivial_false_type){
3.     for (; n > 0; n--, source++, destination++){
4.         // call ctor
5.     }
6. }
7.
8. template <class T>
9. void copy(T* source, T* destination, int n, trivial_true_type){
10.    memmove(source, destination, n); //much faster here!
11. }

```

- std has some type trait class definition.

```

1. cout << typeid ( T ).name() << " "
2. cout << std:: is_pod < T >:: value << " "
3. cout << std:: is_trivial < T >:: value << " "
4. cout << std:: is_standard_layout < T >:: value << std:: endl;

```

- A good article about this topic is "What are Aggregates and PODs and how/why are they special?" in StackOverflow.

## 2.3 Name

### 2.3.1 Namespace

#### 2.3.1.1 namespace basic knowledge

- Basic namespace style:

1. If you develop a library of functions or classes, put them in a namespace, just like std namespace in STL.

2. Don't put a lot of stuff in global-scope, learn to using namespace to manage and split your global-scope.
  3. Don't put your own class into namespace std;
  4. Usually, namespace should be your project name, you can add company name in front of it if you like.
- Use :: before function will means that global namespace, For example, ::max will hide std::max, in this way, you can define your own max function and use ::max call your own max function.
  - **Namespaces can be located at the global level or inside other namespaces. They can't be placed in a block.** So it has external linkage by default, that is to say it can be accessed by multi translation unit(files).
  - There are three kinds of namespace usages.

```

1.  using sp::name;      //1) using declaration
2.  using namespace sp;  //2) using directive
3.  sp::name    //3) specific refer it

4.
5.  namespace ns{
6.  int zy;
7.  }
8.  using namespace ns;

9.
10. int main(){
11.     int zy = 0; //it will hide ns::zy
12.     cin>>zy; //read into local zy
13. }
```

- Using-declaration introduces a member of another namespace into current namespace or block scope.

```

1. #include <iostream>
2. #include <string>
3. using std::string;
4. int main(){
5.     string str = "Example";
6.     using std::cout;
7.     cout << str;
8. }
```

- For implement code, there are four methods to put it into namespace. Prefer method 3 and method 4.

```

1.  namespace Yan{ //a.h
2.  Class Foo{
3.  void mem_fun();
4.  };
5.  }

6.
7. //method 1. then use name, using declaration
8. using Yan::Foo;
9. void Foo::mem_fun() {.....}

10.
11. //method 2. using directive. It's BAD
12. using namespace Yan;
13. void Foo::mem_fun() {.....}
```

```

14. // method 3. Good style
15. void Yan::Foo::mem_fun() {.....}
16.
17. //method 4.
18. //Also good, when you have a lot of function need to be define,
19. //compared with method3, save your typing.
20. namespace Yan{
21.     void Foo::mem_fun() {.....}
22.
23. }
```

- About "using" directive usage

1. **Don't use using directive in any .h file, because It will pollute all the .cpp file which include this head file.**
2. Less use using directive in any .cpp file, you should using scope-resolution or using declaring more. It will avoid polluting namespace.
3. **You should remember below code. It's good C++ style!**

```

1. #include <iostream>
2. using std::cout;
3. using std::endl;
4.
5. Int main() {
6.     cout<<"Hello world"<<endl;
7. }
```

4. If you **have to** use using directory in your .cpp file, put it after all the include files.

- unnamed namespace just like static to specify it to local file scope. At the same time, make anonymous namespace as small as possible. see C++ primer p492

```

1. // a.cpp file
2. static int count;
3.
4. // A better method to use namespace.
5. namespace{
6.     int count;
7. }
```

### 2.3.2 Name lookup

- **Phases of the function call process.**

1. Name lookup
2. Overload resolution
3. Access control

- Function name resolution:

1. First, compiler looks in the **immediate scope**, and makes a list of all functions that has right named (regardless of whether they're accessible or even take the right number of parameters). Pay attentions here, **If found function time, even the parameter doesn't match, compiler will not continue outward search. It will stop here and bark.** It's a safe measure, compiler think the immediate scope is "priority zone". Maybe you omit parameter, compiler should not search outward implicitly.

2. Only if compiler doesn't find any same function name at all, then compiler continue "outward" into the next enclosing scope and repeat.
3. If in the searching scope there are more than one candidate functions, the compiler then stops searching and works with the candidates that it's found, performing overload resolution and then applying access rules.

**4. In one word, overload resolution will not go across scopes!**

- Name lookup include name invisibility or name hidden(introduced in the below subsection).
- Overload resolution has a good introduction: "C++ Primer plus" chapter 8 "Which Function version the compiler pick". The main idea is: **exact match non-template > template > argument conversion(promotion or implicit conversion)**

### 2.3.2.1 Name hiding

- Compiler will first match function name and number of arguments, then look for template deducted function, then use implicit conversion to try match. So implicit conversion happens after template.
- There are four examples about immediate scope.

1. class scope:

```

1. void f1(int i) { ... };
2.
3. class Foo{
4.     void f1(string & str){};
5.     void f2(void){
6.         int i = 3;
7.         f1(i); //compiler bark here!
8.     }
9. };

```

2. Child class scope: In child class scope, if it found name g, then it will stop looking for another name in outward scope. just use g, then found argument number is not match, then compiler bark.

```

1. struct B{
2.     int f( int );
3.     int f( double );
4.     int g( int );
5. };
6.
7. struct D : public B{
8.     private:
9.     int g( std::string , bool );
10. };
11.
12. D d;
13. int i;
14. d.f(i); // ok, means B::f(int)
15. d.g(i); // error: g takes 2 args

```

3. Nested namespace scope. (global includes namespace N)

```

1. void f1(int i) { ... };
2.

```

```

3.  namespace N{
4.    void f1( string & str){};
5.    void f2( void){
6.      int i = 3;
7.      f1(i); //compiler will bark
8.      // you can use ::f1(i) to specify global f1
9.    }
10. };

```

4. Nested scope. A class name or enumeration name can be hidden by an explicit declaration of that same name – as an object, function, or enumerator – in a nested declarative region or derived class.

```

1.  int x = 2;
2.  int x = 3;
3.

```

- overwrite is not standard conception in C++, most time, you can call it name hiding. There are two things: The first one is If you redefine the same name non-virtual function in both base and child classes, you can't use dynamic binding, just static binding. Detail can be seen in namespace section.

```

1.  class A{
2.    f(int) {}
3.  };
4.  class B:public class A{
5.    f(int) {} // overwrite A::f(int)
6.  };
7.
8.  A* pa = new B();
9.  pa->f(3) // will call base f.
10. //because pa is A* even it points to B

```

- How to resolve it. There are two methods:

1. using declaration.
2. You can use :: to specify global variable name or function if you have same name in your local scope. Or use Base:: to specify Base scope name if you have same name in derived class.

```

1.  // method 1;
2.  D d;
3.  int i;
4.  d.f(i); // ok, means B::f(int)
5.  d.B::g(i); // ok, asks for B::g(int)
6.
7.  // method 2:
8.  struct D : public B{
9.    using B::g;
10.   private:
11.    int g( std::string , bool );
12.  };

```

### 2.3.2.2 ADL

- Koenig(ADL) lookup: If you supply a function argument of class type (here x, of type A::X), then to look up the correct function name the compiler considers matching names in the namespace

(here A) containing the argument's type.

```

1.  namespace NS{
2.    class T { };
3.    void f(T);
4.  }
5.
6.  NS::T parm;
7.  int main() {
8.    f(parm); // OK, calls NS::f
9.  }
```

- ADL is for resolve `operator<<` problem.

```

1.  // SakBigNum.h
2.  namespace sak {
3.    struct bignum {
4.      bignum operator++();
5.    };
6.    std::ostream& operator<<(std::ostream&, bignum);
7.  }
8.
9.  // AjoBigNum.h
10. namespace ajo {
11.   struct bignum {
12.     bignum operator++();
13.   };
14.   std::ostream& operator<<(std::ostream&, bignum);
15. }
16.
17. namespace ajo {
18.   void bar(int& x) {
19.     sak::bignum b; // refers to sak::bignum
20.     ++b; // calls sak::bignum::operator++
21.     std::cout << b; // UH-OH! should call yellow one
22.     //But we call << in ajo namespace.
23.   }
24. }
```

- compiler applies ADL whenever it's doing name lookup (building a candidate set) for an unqualified function call.

If the name of the thing-being-called has any ::-qualification at all, then ADL won't kick in.

```

1.  namespace A {
2.    struct A { operator int(); };
3.    void f(A);
4.  }
5.  namespace B {
6.    void f(int);
7.    void test() {
8.      A::A a;
9.      f(a); // ADL, calls A::f(A)
10.     B::f(a); // no ADL, calls B::f(int)
11.   }
12. }
```

- if the thing is not "a function call," then ADL won't kick in. (That is, we don't try to apply Argument-Dependent Lookup to names that don't have arguments.)

- ADL can bring name lookup ambiguous problem. When you call f(parm) f is in global scope, So number 2 is in the **searching scope** default. But ADL bring namespace NS scope into searching scope. In searching scope, there are two options, so compiler will bark.

```

1.  namespace NS { // some header T.h
2.  class T { };
3.  void f( T ); // number 1, add new function
4.  }
5.  void f( NS::T ); //number 2
6.
7.  int main(){
8.  NS::T parm;
9.  f( parm ); // ambiguous: NS::f or global f?
10. }
```

- Just like the previous example, g is in the namespace B, with help of ADL, **searching scope is namespace A + namespace B**. there are two f in the searching scope.

```

1.  namespace A{
2.  class X { };
3.  void f( X ); // <-- new function
4.  }
5.
6.  namespace B{
7.  void f( A::X );
8.  void g( A::X parm ){
9.  f( parm ); // ambiguous: A::f or B::f?
10. }
11. }
```

- In name lookup, the C++ language deliberately says that a member function is to be considered more strongly related to a class than a nonmember.

```

1.  namespace A{
2.  class X { };
3.  void f( X );
4.  }
5.
6.  class B{ // <-- class, not namespace
7.  void f( A::X );
8.  void g( A::X parm ){
9.  f( parm ); // OK: B::f, not ambiguous
10. }
11. };
```

- For a class X, all functions, including free functions, that both "Mention" X and "supplied with" X are logically part of X, because they form part of the interface of X. supplied with X means that that function is defined in the same .h file with type X.
- Keep a type and its nonmember function interface in the same namespace, 1) in logic, these nonmember function can be regarded as type interface, 2) avoid name ambiguous problem in the future.

```

1.  namespace N{
2.  class X {};
3.  X operator+( const X&, const X& );
4.  }
5.
6.  x3 = x1+x2 ;
```

- Keep types and functions in separate namespaces unless they're specifically intended to work together

```

1. #include <vector>
2. namespace N {
3.     struct X {};
4.
5.     // this template should not be put in the namespace N
6.     template<typename T>
7.     int* operator+( T , unsigned ) /* do something */
8. }
```

## 2.4 overload

- Overloading just happens in the same name scope, not in the hierarchy. So in this way, any same name is base class will not visible in derived class. You can use using keyword to add it in your derived class. Then It will compile. Consider these complex things, **Don't redefine or overload base class non virtual functions in your child class**

```

1. class A{
2.     f(int) {}
3. };
4.
5. class B:public class A{
6.     using A::f;
7.     // add it, you can compile your c++ code now.
8.     f(char) {} // overload A::f(int)
9. };
10.
11. B b;
12. b.f(3) // compile error. f(int) is hiding in derived class.
```

- To provide two (or more) functions that perform similar, closely related things, differentiated by the types and/or number of arguments it accepts.
  1. In some cases it's worth arguing that a function of a different name is a better choice than an overloaded function.
  2. In the case of constructors, overloading is the only choice.
  3. operator overload is also very common.
- Consider overloading to avoid implicit type conversion. In this way, we can avoid creating a temporary obj implicitly created by ctor.

```

1. class String{
2.     bool operator==(const String& lhs, const String& rhs);
3. }
4. String str;
5. if(str == "hello") //will build a temp obj String("hello");
6.
7. //You can define overload to avoid implicit type conversion
8. bool operator==(const String& lhs, const char* rhs);
```

- Sometimes, overload and default parameter have same client usage, such as

```
1. void f();  
2. void f(int x);  
3. f() or f(10);  
4.  
5. void g(int x = 0);  
6. g() or g(10);
```

If there is a value that you can use for a default, use default parameter. The difference is logical  
- When overload, the two functions can behave completely different, whereas the second case will have more or less the same logic.



# Chapter 3

## initialization

### 3.1 Basic

#### 3.1.1 principle rule

- **always initialize variable before you use it.** Manually initialize non-member objects of built-in types, including a pointer.
- Use **member initialization list** to initialize all members inside an object.
- There are three names: 1) initializer list, 2) brace(list) initialization, 3) member initialization list(mem-init)
  1. member initialization list is used in C++ ctor. It has some advantages: Detail can be found in "OOP" section.
  2. brace initialization is a generic initialization syntax(method), it support more initialization, such as class member and aggregation. At the same time, it will avoid narrowing and vexing parsing problem.
  3. initializer list is `std::initializer_list`. It is a **new data type**. just like `std::list`.

#### 3.1.2 initialization order

- In the same translation unit, formally, C++ initializes static and global variables in three phases:
  1. Zero initialization
  2. Static initialization
  3. Dynamic initialization

```
1. int g0; //zero initialization
2. int g1 = 42; // static initialization
3. extern int f();
4. int g2 = f(); // dynamic initialization
```

- It will cause some subtle bug shown below:

```
1. int a = f(); // a exists just to call f
2. int x = 22;
3. int f() {
4.     ++x;
5.     return 123; // unimportant arbitrary number
6. }
7. // x equals to 23, not 22
8. // because static initialization
9. // is earlier than dynamic initialization
```

- Pay attention to the global object which are initialized in right order. Because the order can be arranged in the same translation unit, but not **between translation units**, see effective c++ Item 47. So we don't encourage you to use global object unless it's very necessary.
- The initialization order uncertainty that afflicts non-local static objects defined in separate translation units. The questions is below:

```

1. #include "x.h" // File x.cpp
2. X x;
3. //x maybe initialize before y or after y
4.
5. #include "y.h" // File y.cpp
6. extern X x;
7. Y y;
8. Y::Y() { //here x maybe not be constructed
9.     x.goBowling();
10. }
```

- How to resolve: build a singleton class

```

1. #include "x.h" // File x.cpp
2. X& getX() {
3.     static X x;
4.     return x;
5.     //another implementation.
6.     //static X* px = new X();
7.     //return *px;
8. }
9.
10. #include "y.h" // File y.cpp
11. Y y;
12. Y::Y() {
13.     getX().goBowling();
14. }
```

### 3.1.3 Init method

In this section, I will introduce a few conceptions together. They are related. The first one is vexing parsing. The second one is value-initialization. The third one is brace init and the last one is `initializer_list`

#### 3.1.3.1 Six init methods

- There are six initializations forms: default, value, direct, copy, aggregate init and reference init.

```

1. T t;
2. new T; // default
3. //_____
4. T t {};
5. T(); T{};
6. new T(); new T{};
7. : member(), member{} // class member initializer lists (value Init)
8. //_____
9. T object(arg, ... );
10. T(arg1, arg2, ... );
11. new T(args, ... )
12. : member(args, ...) // class member initializer lists
13. T(other) // function-style cast
```

```

14. static_cast<T>(other) // explicit static_cast
15. [arg]{}() {...}           // lambda closure arguments captured by value
16. //lambda is object, so for this object,
17. //lambda_1 obj(arg), is a direct init
18. //
19. T object = other;        // Initialization via assignment
20. T array[N] = {other};    // In array-initialization, the individual
21. // values are copy-initialized
22. f(other)                // Pass-by-value
23. return other;           // Return-by-value
24. catch (T object)        // Catch-by-value
25. throw object;

```

```

1. T object = {arg1, arg2, ...}; // If T is an array or a simple struct
2. T object{arg1, arg2, ...};   // If T is an array or a simple struct

```

- reference init is easy, it must has a reference symbol,

```

1. T & ref = object ;
2. T & ref = { arg1, arg2, ... };
3. T & ref ( object ) ;
4. T & ref { arg1, arg2, ... } ;

```

- Summary:

1. Without parentheses, it's default, tell the compiler to use it's default method;
2. Use empty parentheses or brace, it's value, tell the compiler to init them, (if there is user define one, use user-define, otherwise zero)
3. Use parentheses with value, direct init, just command compile to init what I want.
4. When pass to function or return from function by value, is copy
5. Aggregate init can only be used for aggregate type.
6. reference init is the most easy.
7. Many different syntax expression can be useful when you do generic programming.

- for list init, there are three usages: value list init, direct list init and copy list init

```

1. T object {};
2. T{};
3. new T{}
4. Class { T member{}; };
5. : member{}                      // Class member initializer lists
6.
7. T object{arg, ...};
8. T {arg, ...};
9. new T{arg, ...}
10. Class { T member{arg, ...}; }; // Class member default initializer
11. : member{arg, ...}            // Class member initializer lists
12.
13. T object = {arg, ...};
14. object = {arg, ...};
15. Class { T member = {arg, ...}; }; // Class member default initializer
16. function({arg, ...}); // Initializes temporary for the function arg
17. return {arg, ...}; // Initializes temporary for return value

```

- brace init is just in syntax level, for the detail init method, it depends on the specific data type. For example: If T is an aggregate type, **aggregate initialization** is performed. Otherwise, If the braced-init-list is empty and T is a class type with a default constructor, **value-initialization** is performed.

### 3.1.3.2 value init

- basic knowlege. googling "Value-initialization with C++"
- For value init, It's depends on if you have user define ctor. if you don't have, set it to zero. But if you have one, compiler will not set it to zero any more. That is easy to understand, you have higher authorization than compiler.

```

1. class exec{
2.     public:
3.     exec() = default;
4.     int i;
5. };
6.
7. class exec2
8. {
9.     public:
10.    exec2();
11.    int i;
12. };
13. exec2::exec2() = default;
14.
15. const exec e;           //error, exec has default ctor
16. const exec2 e2;         //right, exec2 has user-define ctor
17.
18. exec e;                //default init, so e.i is random
19. exec e{}               //value init, but there is no user-define ctor, so e.i == 0
20.
21. exec2 e;               // default init, so e.i is random
22. exec2 e{}
23. //value init, but there is user-define ctor, so e.i is random

```

- Value init also kick in when you use `new`. Detail can be found: "Do the parentheses after the type name make a difference with new?"

```

1. struct A { int m; }; // POD
2. struct B { ~B(); int m; }; //non-POD, compiler generated default ctor
3. struct C { C() : m() {}; ~C(); int m; };
4. // non-POD, default-initialising m

```

In a C++03 compiler, things should work like so:

- `new A` - indeterminate value
- `new A()` - value-initialize A, which is zero-initialization since it's a POD.
- `new B` - default-initializes (leaves B::m uninitialized)
- `new B()` - value-initializes B which zero-initializes all fields since its default ctor is compiler generated as opposed to user-defined.
- `new C` - default-initializes C, which calls the default ctor.
- `new C()` - value-initializes C, which calls the default ctor.

### 3.1.3.3 copy init and direct init

- A good article is "Is there a difference between copy initialization and direct initialization?"
- **copy init is difference with copy ctor.** calling copy ctor is direct init? If it's copy init, so it need two steps, create temp, then copy temp. if it's direct init, just call one ctor(by overload resolve). Above it's theoretical definition. When optimization kick in, it hide the deep theoretical definition.
- An example and analysis:

```

1. ClassTest ct1("ab");//direct init
2. ClassTest ct2 = "ab";//copy init
   // When optimization turn on, ct2 call single argument directly
   // But it's still copy init.
3. ClassTest ct3 = ct1;//copy init
4. ClassTest ct4(ct1);//direct init
5. ClassTest ct5 = ClassTest();//copy init Class has default ctor
6. ClassTest ct6 = {1,2} //copy init Class has two parameter ctor.

```

1. ct2 is copy init. Compiler analyzes the syntax according theoretical definition, for ct2, if copy ctor is private, then compile will stop. if copy ctor is public, then optimizaiton kick in, it call single argument ctor directly.
2. for ct3, It's copy init. Compiler thinks the ct1(temp) has been created, then call copy ctor directly. but in fact, it's still copy init.
3. ct4 is direct init call copy ctor. because in theoretical it's only one step, call a ctor by overload.
4. In one word, if it's a copy init is not depend on whether it call copy ctor. it's depend on (in theoretical) whether it has two steps(has equal sign).

- When copy ctor is explicit.

```

1. class A{
2. public:
3.     A(int a = 0);
4.     explicit A(const A &a){}
5. };
6.
7. void funcX(A a) {
8.     //ERROR to take A by value (implicit copying)
9. }
10. A funcY(){
11.     A a;
12.     return a; //ERROR - function returning A by value (implicit copying)
13. }
14. A a1 = a; //ERROR implicit copying of TestOverload not allowed
15. A a1(a); //OK - EXPLICIT copying allowed

```

1. Difference between A a(a1) and A a = a1. They are almost same. But When copy ctor is explicit, A a1 = a will not work, but A a1(a) work.
2. Usually, we don't make copy ctor explicit, because it will disable function call and return value.
3. **With explicit copy ctor, A a = 1 still work. But when the copy ctor is private, A a = 1 fail**

- When single parameter ctor is explicit.

```

1. class A{
2. public:
3.     int i;
4.     explicit A(int a = 0);
5.     A(const A &a){}
6. };
7.
8. A a = 1; // fail.
9. //A a = {1} //fail
10. A a = A{1} //work
11. A a = A(1) //work

```

- When parameterized ctor and copy ctor are both explicit.

```

1. class A{
2. public:
3.     explicit A(int k):m_a(k){};
4.     explicit A(const A& rhs){m_a = rhs.m_a;};
5.     virtual ~A(){}; //not aggregate
6.     int m_a;
7. };
8.
9. int main(){
10. A a={110};// will compile error, because A(int k) is explicit
11. A a{110} //this will work here.
12. A b = {a}; // will compile error, because copy ctor is explicit.
13. }

```

1. Most of time, Single parameter ctor is explicit, copy ctor is not.
2. With explicit A(int i) ctor, A a = 1 will fail, but A a = A1 will work.

- Another more interesting example

```

1. class A{
2. public:
3.     int i;
4.     explicit A(int a = 0):i(a){cout<<"one";}
5.     A(int a, int b):i(a){cout<<"two";}
6.     A(const A &T){ cout<<"copy_ctor";}
7. };
8.
9. A too = (1,2); //fail, it (1,2) became 2, then call single ctor
10. //() and comma is operator, it's different with {}
11. A too = {1,2}; //work, call two parameter ctor

```

- For assignment, you can use braced init and avoid type name. But when you define explicit converting ctor, it will not work.

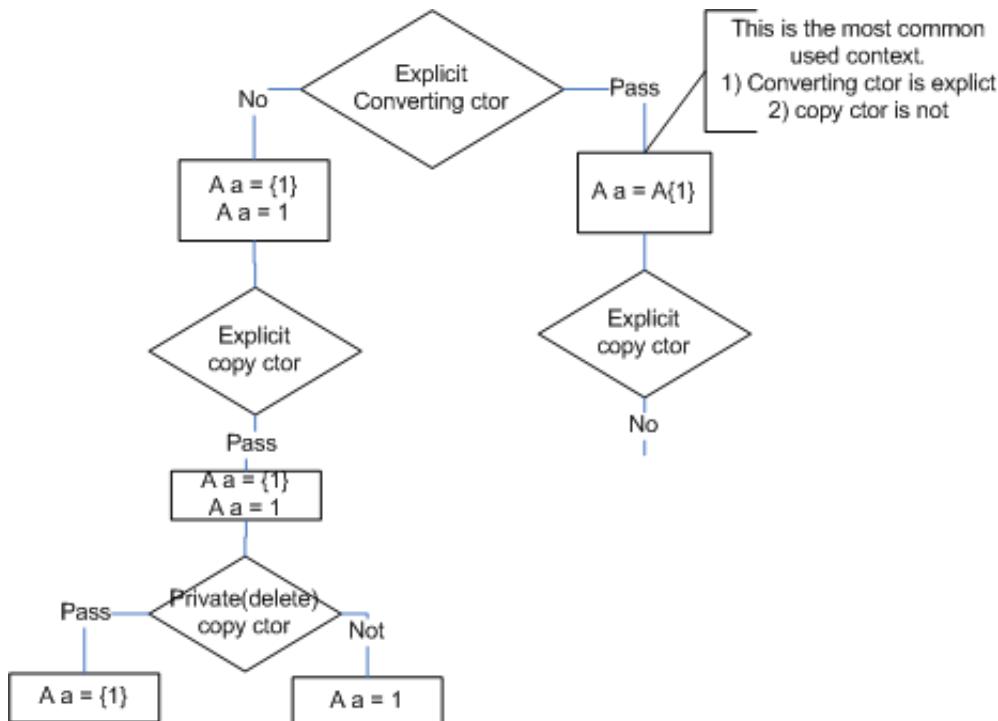
```

1. pair<int, string> p;
2. p = {1,"aaa"}
3.
4. A a;
5. a = {1} //without explicit converting ctor
6. a = A{1} //with explicit converting ctor

```

- Summary:

1. The first step is to see the right side of `=`, if converting ctor is explicit, we have to use `A{x, y}` to explicit call it. For example: `A a = A{x, y}` will work. `A a = {x,y}` will not work.
2. The second step is to see when you use brace init. if it has initializer\_list ctor, it has strong preference. we will talk about it later.
3. The third step is to see if copy ctor is explicit. if yes, `A a = A{x, y}` will fail, but `A a = {x,y}` work.
4. if copy ctor is private, both `A a = A{x, y}` and `A a = A(x,y)` fail But `A a = {x,y}` still work.



Expression	meaning
<code>A a = (i), A a={i}</code>	explicit converting ctor will not work
<code>A a = A(i), A a = A{i}</code>	explicit copy ctor will not work

- In C++, anonymous variables are primarily used either to pass or return values without having to create lots of temporary variables to do so.
- anonymous objects are treated as rvalues (not lvalues, which have an address). This means anonymous objects can only be passed or returned by value or const reference. Otherwise, a named variable must be used instead.

```

1. Cents add(const Cents &c1, const Cents &c2){
2.     return Cents(c1.getCents() + c2.getCents());
3.     // return anonymous Cents value
4. }
5.
6. cout <<add(Cents{6}, Cents{8}).getcents();

```

- a good link is <https://www.cnblogs.com/pluse/p/7088880.html>

## 3.2 Brace Init

### 3.2.1 Basic syntax

- The basic syntax examples.

```

1. int x1 = 27; //1) assignment
2. int x2(27); //2) parentheses before C++11
3.
4. int x3{ 27 }; //3) brace init, introduced in C++11
5. int x4={ 27 }; //Like 3, but for class type, a trivial difference.
6. //can't use with explicit copy ctor(introduced before)

```

- More detail syntax analysis.

```

1. widget w; // (a)
2. widget w(); // (b)
3. widget w{}; // (c)
4.
5. widget w(x); // (d)
6. widget w{x}; // (e)
7.
8. widget w = x; // (f)
9. widget w = {x}; // (g)
10.
11. auto w = x; // (h)
12. auto w = widget{x}; // (i)

```

- (a) is just default init with default ctor.
- (b) is function declaration.** That is why we prefer use brace init (c), it will avoid this vex problem.
- (d) and (e) Assuming x is not the name of a type, these are both direct initialization. That's because the variable w is initialized "directly" from the value of x by calling `widet::widget(x)`. If x is also of type `widget`, this invokes the copy constructor. Otherwise, it invokes a converting constructor.
- (e) is better than (d) to avoid narrowing, for example:

```

1. Class A{
2.   A(int){}
3. }
4. A a(1.2); //work
5. A a{1.2}; //fail

```

- For (e), it prefer constructor that takes an `initializer_list`
- For (f), when x is type of `widget`, just like (d), but can't work when copy ctor is explicit(introduced before)
- for (f), when x is not type of `widget`, it will convert x to a temporary. If x is of some other type, **conceptually** the compiler first implicitly converts x to a temporary `widget` object, then move-constructs w from that temporary rvalue, using copy construction as "the slow way to move" as a backup if no better move constructor is available. Assuming that an implicit conversion is available, (f) means the same as `widget w(widget(x))`. Note that I said "conceptually" a few times above. That's because practically compilers are allowed to, and routinely do, optimize away the temporary and, if an implicit conversion is available, convert (f) to (d), thus optimizing away the extra move operation. However, even when

the compiler does this, the widget copy constructor must still be accessible, even if is not called—the copy constructor's side effects may or may not happen.

8. For (g), just like (f)
9. (h), just like type-of-x w(x), A w(a) That will call A copy ctor directly. is same with (d).(when copy ctor is explicit, it will not work)
10. (i), Line (i) is the most consistent spelling when you do want to commit to a specific type and explicitly request a conversion if needed, and once again the { } syntax happily avoids lossy narrowing conversions. In practice on most compilers, only a single constructor is called—similarly to what we saw with (f) and (g).

```

1. class TestOverload{
2. public:
3.     int i;
4.     TestOverload(int a = 0):i(a){}
5.     TestOverload(const TestOverload &T){ cout<<"copy<u>ctor"; }
6. };
7.
8. TestOverload too = 2; //test for (f)
9. TestOverload too = {2} // test for (g)
10. //1) both (f) and (g) work and no call
11.      // copy ctor(no output:"copy ctor")
12. //2) If converting ctor has explicit
13.      //---both fail. TestOverload too{2} --succeed.
14. //3) if copy ctor is private -- (f) fail.
15. // (strange, even I don't call you, but you must accessible)
16.
17. TestOverload too; //test for (h)
18. auto w = too
19. 1) When if copy ctor has explicit --fail.
20.
21. auto w = TestOverload{2} //test for (i)
22. //1) you can use auto w = TestOverload(2),
23.      // it just call ctor, and w is not const reference.
24. //2) {} is better, it more generic and avoid narrow
25. // auto w = TestOverload{1.2} fail, auto w=TestOverload(1.2) ok
26. //3) even converting ctor is explicit, it can work too.
27. //4) don't call copy ctor at all

```

11. summary, (h) and (i) is the best way. stick to them when you write the problem in the future.

### 3.2.2 Brace init advantage

- **empty brace for single or aggregated variable** An empty pair of braces indicates value initialization. value initialization of POD types usually means initialization to binary zeros, whereas for non-POD types value initialization means 1) zero if you don't user defined ctor 2) use user defined ctor:

```

1. //C++11: default initialization using {}
2. int *p{}; //initialized to nullptr
3. char s[12]{}; //all 12 characters are initialized to '\0'
4. char *p=new char [5]{}; // all five chars are initialized to '\0'
5.
6. int i{}; // i becomes 0, can't use () here
7. std::string s{}; // s becomes "" can't use () here
8. std::vector<float> v{}; // v becomes an empty vector
9. double d{}; // d becomes 0.0, can't use () here

```

- Only brace init can be used to initialize **class aggregated member**

```

1. class Array{
2. public:
3.     Array () : myData{1,2,3,4,5} {}
4. private:
5.     int myData[5];
6. };

```

- direct init aggregated variable**

```

1. int intArray []= {1,2,3,4,5};
2. std::vector<int> intArray1 {1,2,3,4,5};
3. std::map<std::string, int> myMap{{"Scott",1976}, {"Dijkstra",1972}};
4.
5. // Initialization of a const heap array
6. const float* pData= new const float[3]{1.1,2.2,3.3};

```

- init a class event without ctor**

```

1. class MyClass{
2. public:
3.     int x;
4.     double y;
5. };
6.
7. class MyClass2{
8. public:
9.     MyClass2(int fir, double sec):x{fir},y{sec} {};
10. private:
11.     int x;
12.     double y;
13. };
14.
15. // Initializations of an arbitrary object using public attributes
16. MyClass myClass{2011,3.14};
17. MyClass myClass1= {2011,3.14};
18.
19. // Initializations of an arbitrary object using the constructor
20. MyClass2 myClass2{2011,3.14};
21. MyClass2 myClass3= {2011,3.14};

```

- omit type name**

```

1. MyClass2 fun (MyClass2 m)
2.
3. fun ( {2011,3.14} ){
4.     return {2011,3.14}; //pay attention, there is ; in the end
5. }
6.
7. fun ( MyClass2(2011,3.14) ) // fun ( (2011,3.14) ) doesn't work
8.
9. MyClass2 mc2;
10. mc2 = {2011,3.14} // or MyClass2(2011, 3.14)

```

- compared with c++98, we can see the improvement of brace init.

```

1. double *pd= new double [3] {0.5, 1.2, 12.99};
2. //before { no =, or compile error

```

```

3. // C++98
4. rectangle      w( origin(), extents() );           // oops, vexing parse
5. vector<int>   v;                                // urk, need more code
6. for( int i = 1; i <= 4; ++i ) v.push_back(i);    // to initialize this
7.
8.
9. // C++11 (note: "=" is mostly optional)
10. rectangle     w = { origin(), extents() };
11. vector<int>   v = { 1, 2, 3, 4 };

```

- **Also work in template** And note that this isn't just an aesthetic issue. Consider writing generic code that should be able to initialize any type... and while we're at it, let's gratuitously use perfect forwarding as an example:

```

1. template<typename T, typename ... Args>
2. void forwarder( Args&&... args ) {
3.     T local = { std::forward<Args>(args)... };
4.     // ...
5. }
6.
7. forwarder<int>          ( 42 );                  // ok
8. forwarder<rectangle>    ( origin(), extents() ); // ok
9. forwarder<complex<double>>( 2.71828, 3.14159 ); // ok
10. forwarder<mystruct>    ( 1, 2 );                // ok because of {}
11. forwarder<int[]>       ( 1, 2, 3, 4 );        // ok because of {}
12. forwarder<vector<int>> ( 1, 2, 3, 4 );        // ok because of {}

```

- I'd say the brace one is better, because it's future proof. If you later rename the class, the brace one will continue to work as is, while the parenthesis one will require a name change. Also, if you change the type of vec3D, the brace one will still create an object of vec3D's type, while the second one will keep creating and assigning from a float3 object. It's not possible to say universally, but I'd say the former behaviour is usually preferred.
- A good reference article is GotW #1 Solution: Variable Initialization – or Is It?
- Another good one is Overview of C++ Variable Initialization. address is <https://greek0.net/cppinitialization.html>

### 3.2.3 copy-list-initialization and traditional copy-initialization?

- A good article is "Any difference between copy-list-initialization and traditional copy-initialization?"
- Basic idea of all kinds of initialization. What's the difference?

```

1. widget w{x}; 1)
2. widget w = x; 2)
3. widget w = {x}; 3)

```

- For the number 1, behaves like a function call to an overloaded function: The functions, in this case, are the constructors of T (including explicit ones), and the argument is x. Overload resolution will find the best matching constructor, and when needed will do any implicit conversion required.
- For the number 2, Copy initialization constructs an implicit conversion sequence: It tries to convert x to an object of type T. (It then may copy over that object into the to-initialized object, so a copy constructor is needed too - but this is not important below)

- for the number 2, If x is of some other type, conceptually the compiler **first implicitly converts x to a temporary widget object**, then move-constructs w from that temporary rvalue, using copy construction as "the slow way to move" as a backup if no better move constructor is available. Assuming that an implicit conversion is available.
- For the number 2, implicitly convert x has two different ways: As said above, copy initialization will construct a conversion sequence when a has not type B or derived from it (which is clearly the case here). So it will look for ways to do the conversion, and will find the following candidates

```

1. struct B;
2. struct A {
3.     operator B() { std::cout << "<copy>" ; return B(); }
4. };
5.
6. struct B {
7.     B() { }
8.     B(A const&) { std::cout << "<direct>" ; }
9. };
10.
11. int main() {
12.     A a;
13.     B b1(a); // 1)
14.     B b2 = a; // 2)
15. }
16. // output: <direct> <copy>
```

- For the number 3, This is called "copy list initialization." It means the same as widget wx; except that explicit constructors cannot be used. It's guaranteed that only a single constructor is called. So number 3 is different with number 2
- Old initialization uses notion of user-defined conversion sequences (and, particularly, requires availability of copy constructor, as was mentioned). Brace initialization just performs overload resolution among applicable constructors, i.e. brace initialization can't use operators of conversion to class type.

```

1. struct Intermediate {};
2.
3. struct S{
4.     operator Intermediate() { return {}; }
5.     operator int() { return 10; }
6. };
7.
8. struct S1{
9.     S1(Intermediate) {}
10. };
11.
12. S s;
13. Intermediate im1 = s; // OK
14. Intermediate im2 = {s}; // ill-formed
15. // no Intermediate ctor receive S
16. S1 s11 = s; // ill-formed
17. S1 s12 = {s}; // OK
18. // S1 ctor receive Intermediate, then
19. // s implicitly convert to Intermediate
20.
21. // note: but brace initialization can use operator of conversion to int
22. int i1 = s; // OK
23. int i2 = {s}; // OK
```

### 3.2.4 vexing parsing

1. The basic reason behind of vexing parsing come from C++ standard: "If it can be a function declaration, it is".
2. Use parentheses to group when you declare a variable. **When you declare something, It's legal to use parentheses to group variable name.** Sometimes, you have to use parentheses, such as declare a function pointer.

```
1. int f(int); //a function return a int
2. int *f(int); //a function return pointer
3. int (*f)(int); // function pointer;
```

3. It also means that you can have below C++ statements. The whole idea is just like to make  $(2+3)*4$  work, you have to make parentheses applicable, so although unnecessary,  $(2)+(3)$  must be a legal statement.

```
1. int x;
2. int (x); //same as above
3. int f(int x);
4. int f(int (x)); //same as above
```

4. For function declaration, you can omit the parameter name.

```
1. int f(int x);
2. int f(int (x));
3. int f(int); //three statements are same.
```

5. For object, thing becomes interesting.

```
1. int (x); // just like int x;
2.
3. class A{
4.     A(int a);
5. }
6.
7. A(a); // You want to create a temporary obj
8. //but compile think it as A a;
9. int a = 10;
10. A(a) //error, redefine a, because if it's decaration, it is
11. const & ar = A(a); //this is OK
12.
13. (A(a));
14. //If you really want a temporary A object,
15. //use another parentheses .
```

6. For function, vexing problem happen below.

```
1. int f(double (*pf)()); //standard way
2. int f(double pf()); //You can omit parentheses
3. // just like int x and int (x);
4. int f(double()); //for function f, you can omit variable name.
```

7. Based on all above all the knowledges. We have a complex problem.

```

1. struct A{
2.     A (B const& b) {};
3.     void doSomething() {};
4. };
5.
6. int main(){
7.     A a(); // declare a function "a", return A
8.
9.     A a(B(x)); // declare a function "a",
10.    // type of parameter x is B
11.
12.    A a(B()); // declare a function "a",
13.    // receive function pointer, return B, void input
14.    and parameter name for "a" is omitted.

```

8. In previous example, Why B(x) equal B x; but B() equal a function pointer? If B(int), then it will be a function pointer. function declaration need type name, not a variable name, such as x.
9. The most complex problem. A range container constructor.

```

1. list<int> data(istream_iterator<int>(dataFile),
2. istream_iterator<int>());
3. // data is function. first parameter is dataFile, type istream_iterator<int>
4. // second parameter is function pointer.
5. // The First, you can add () around parameter name;
6. // the second, you can omit parameter name

```

10. Before C++11, you need add another pair parentheses to change it from declaration to expression.  
After C++11, you can use braces.

```

1. A a{B{x}};
2. A a{B{}};

```

11. A non-static data member initializer (NSDMI) must use a brace-or-equal-initializer.

```

1. class A{
2.     int equals = 42; // OK
3.     std::unique_ptr<Foo> braces{new Foo}; // Also OK
4.     std::vector<int> bad(6); // not allow
5.     std::vector<int> good{6}; // OK
6. }

```

12. a vexing parse example is below, there is another complex exmaple in previous section "vexing parse problem"

```

1. class float3{
2.     .....
3. };
4.
5. float3 x(); //x is function, not call default ctor
6.
7. float3 x(float3()); // a function
8. float3 x(float3(i)); // still a function, float3(i) equal float3 i;
9. //because () can be omitted by a compiler.
10. //vs.
11. float3 y{float3{}}; // a variable

```

13. A good reference is <https://timesong-cpp.github.io/cppwp/n4140/stmt.ambig> or google search "everything that can be a declaration is a declaration"

### 3.3 initializer\_list

#### 3.3.1 why initializer\_list

- the whole story comes from initialize format of array in C language. But this way can't be used in C++ vector object

```
1. int array [] = {1,2,3,4,5} //OK
2. vector<int> vt = {1,2,3,4,5} //Can't use in this way
3. //because vt is class, class is not array in C
```

- In the generic programming, we need to use the "uniform initializer" make the vt and array use the same syntactic usage. In order to make it possible, we introduce `std::initializer_list` template. It's the proxy class. We need vector to have a ctor with `initializer_list` parameter

```
1. vector( std::initializer_list<T> init)
2. vector<int> vt = {1,2,3,4,5} //will call previous ctor and it works.
```

- If the values you are initializing with are a list of values to be stored in the object (like the elements of a vector/array, or real/imaginary part of a complex number), use curly braces initialization if available.
- In STL library, list, vector, map support `initializer_list` ctor. You also can use it in your own class or function.

```
1. void f( const initializer_list<string> &slst ){
2.   ...
3. }
4. f({ "Good", "morning", "!" });
```

- `std::initializer_list` can be used in function parameter and return value and for range.

```
1. // A braced-init-list can be implicitly converted to a return type.
2. vector<int> test_function() { return {1, 2, 3}; }
3.
4. // Iterate over a braced-init-list.
5. for (int i : {-1, -2, -3}) {}
6.
7. // Call a function using a braced-init-list.
8. void TestFunction2(vector<int> v) {
9. TestFunction2({1, 2, 3});
```

#### 3.3.2 difference with brace init

- You should not only judge `initializer_list` by braces.** There are two points: 1) The class should have a `initializer_list` ctor 2) it uses braces. 3) the number of parameters inside of braces is changeable. 4) the type should be the same.

```
1. vector( std::initializer_list<T> init)
2. vector<int> vt = {1,2,3,4,5} //will call previous ctor and it works.
3.
```

```

4. class A{
5. public:
6.     int i;
7.     int j;
8. };
9.
10. A a = {1, 2} //It's brace init, not initializer|_list
11.           //because A has not ctor with initializer|_list

```

- Mostly, the two features I presented play very well together, for example if you want to initialize a map you can use an initializer-list of braced-init-lists of the key value pairs: Here, the type of the pairs is clear and the compiler will deduce, that '"Alex", 522' in fact means 'std::pair<std::string const, int>"Alex", 522'.

```

1. std::map<std::string, int> scores{
2.     {"Alex", 522}, {"Pumu", 423}, {"Kitten", 956}
3. }

```

- An empty pair of braces can be:

1. Default initialization.
2. An empty std::initializer\_list.

You should know how to distinguish them. Empty braces mean no arguments, that is to say you get default construction, not an empty std::initializer\_list:

```

1. class Widget{
2. public:
3.     int i;
4.     Widget() {cout << "default";}
5.     Widget(initializer_list<int>){cout << "init";}
6. };
7.
8. Widget w1; // calls default ctor
9. Widget w2{}; // also calls default ctor, even you have
10.          // initializer_list ctor
11. Widget w3(); // most vexing parse! declares a function!
12.
13. Widget w4({}); // calls std::initializer_list ctor with empty list
14. Widget w5{{}}; // ditto

```

### 3.3.3 problem of initializer\_list

- For empty brace init, 1) it will call default ctor if it has 2) or call init\_list ctor if it doesn't have default ctor.

```

1. class Widget{
2. public:
3.     int i;
4.     Widget() {cout << "default";}
5.     Widget(initializer_list<int>){cout << "init";}
6. };
7.
8. Widget w2{};

```

- When 1) Non-empty brace init is used, 2) and there are overload initializer\_list, it always match initializer\_list.

```

1. class Widget {
2. public:
3.     Widget(int i, bool b); // as before
4.     Widget(int i, double d); // as before
5.     Widget(std::initializer_list<long double> il); // added
6. };
7.
8. Widget w1(10, true); // calls first ctor
9. Widget w1{10, true}; // std::initializer_list ctor
10. // (10 and true convert to long double)
11.
12. Widget w2(10, 5.0); // calls second ctor
13. Widget w2{10, 5.0}; // std::initializer_list ctor
14. // (10 and 5.0 convert to long double)

```

- Compilers' determination to match braced initializers with constructors taking std::initializer\_lists is so strong, it prevails even if the best-match std::initializer\_list constructor can't be called.

```

1. class Widget {
2. public:
3.     Widget(int i, double d); // as before
4.     Widget(std::initializer_list<bool> il); // added
5. };
6. Widget w2{10, 5.0}; // still match init_list,
7. // 5.0 can't be convert bool, so compile fail.

```

- An example from vector, () and init brace are different in meaning.

```

1. std::vector<int> v1(10, 20); // use non-std::initializer_list
2. // ctor: create 10-element vector, all elements 20.
3.
4. std::vector<int> v2{10, 20}; // use initializer_list ctor:
5. // create 2-element vector, element are 10 and 20

```

- Based on previous vector example, choosing between parentheses and braces for object creation inside templates can be challenging.(it has different semantic meaning.)

## 3.4 auto init(AAA)

About the auto type deduction, please refer to the next chapter Generic programming type inference section. The basic idea is quite simple:

**take exactly the type on the right-hand side, but strip off top-level const/volatile and &/&&.**

### 3.4.1 auto declaration

#### 3.4.1.1 Basic knowledge

- When you use expr init a variable, prefer to use auto. expr can be:

1. math express, auto a = b+c.
2. function call, auto a = v.begin()
3. new to avoid type name

4. left hand type is same with right type auto a = b ( you can write Type a(b), but don't need write down the type name)
5. lambda
6. template

```
1. auto a = expr
2. auto a = type{expr} //when you want to commit to type
```

```
1. //const char* s = "Hello ";
2. auto s = "Hello";
3.
4. //widget w = get_widget();
5. auto w = get_widget();
6.
7. widget* w = new widget{}; /* auto w = new widget{}; */
8. unique_ptr<widget> w = make_unique<widget>();
9. = make_unique<widget>();
```

- **auto** sets the type of a declared variable from its initializing expression while compiling.

```
1. auto p = &x //p is int* type
2. //You prefer to use auto* p
3. auto p = new vector<pair<int, string>>;
4. //avoid complex verbose on left side
5.
6. std::vector<std::pair<int, std::string>> array;
7. auto it = array.begin();
8. //avoid complex verbose on left side
9.
10. double fm(double, int);
11. auto pf = fm // pf is double(*)(double, int)
12.
13. for (const auto& element : myarray) {
14.     //do stuff that reads from element
15. }
```

- If you want your auto declaration to be const, or if you want your auto declaration to be a reference, you have to add them explicitly

```
1. //old usage
2. const std::map<int, Module>::iterator iter = modmap.find(123);
3. Module& mod = vec[17];
4.
5. //use auto, it's better
6. const auto iter = modmap.find(123);
7. auto& mod = vec[17];
```

- Both of these are the same and will declare a pointer. But auto\* var; if you think it stresses the intent (that var is a pointer) better.

```
1. auto var = new int(1);
2. auto *var = new int(1);
```

- Four advantages:

1. avoidance of uninitialized variables.

```
1. auto x2; // error! initializer required
2. auto x3 = 0; // fine, x's value is well-defined
```

2. avoidance of verbose variable declarations.

```
1. template<typename It>
2. void dwim(It b, It e) {
3.     //typename std::iterator_traits<It>::value_type currValue = *b;
4.     auto currValue = *b; //using auto is much better
5.
6.     auto ii = find_if(people.begin(), people.end(), match_name);
7.     if(ii != people.end()) {...}
```

3. avoid what I call problems related to "type shortcuts." **It's very importance here, it guarantee correct and performance**

```
1. std::vector<int> v;
2. ...
3. unsigned sz = v.size();
4. auto sz = v.size();
5.
6. std::unordered_map<std::string, int> m;
7. for (const std::pair<std::string, int>& p : m) {
8.     ... // do something with p
9. }
10. //pair type is <const std::string, int>
11. //When you use auto, you don't have this error
12. for (const auto& p : m) {
13.     ... // as before
14. }
```

4. Good maintenance.

```
1. struct record {
2.     std::string name;
3.     int id;
4.     //GUID id; //change id type, below doesn't need change
5. };
6.
7. auto find_id(const std::vector<record> &people,
8.               const std::string &name)
9.
10. int find_id(const std::vector<record> &people,
11.               const std::string &name)
12. //Need to change return type from int to GUID if don't use auto
```

### 3.4.1.2 pitfall of auto

1. "invisible" proxy classes. An invisible proxy class example is `vector<bool> []` operator. `vector<bool>::operator[]` neither yields a bool nor a reference to a bool. It just returns a little proxy object that acts like a reference. This is because there are no references to single bits and `vector<bool>` actually stores the bools in a compressed way. So by using `auto` you just created a copy of that reference-like object. The problem is that C++ does not know that this object acts as a reference. You have to force the "decay to a value" here by replacing `auto` with `T`

```
1. template <typename T>
```

```

2. void Test(const T& oldValue, const T& newValue, const char* message){
3.     vector<T> v;
4.     v.push_back(oldValue);
5.     cout << "before : " << v[0] << '\n';
6.
7.     auto x = v[0]; // Should be a deep-copy
8.     x = newValue;
9.
10.    cout << "after : " << v[0] << '\n';
11. }
12.
13. int main() {
14.     Test<int>(10, 20, "Testing vector<int>");
15.     Test<bool>(true, false, "Testing vector<bool>");
16. }
17. // before : v[0] = 1
18. // after : v[0] = 0, //Because it returns reference,
19.           // so v[0] has been modified.

```

2. init brace return init\_list. Prior to C++17 the type for all the following objects (a, b, c and d) is deduced to std::initializer\_list<int>. There is no difference between the direct-list-initialization and the copy-list-initialization on the result of the type deduction.

```

1. auto a = {42}; // std::initializer_list<int>
2. auto b {42}; // std::initializer_list<int>
3. auto c = {1, 2}; // std::initializer_list<int>
4. auto d {1, 2}; // std::initializer_list<int>

```

3. changed in C++17 that introduced the following rules: 1) for copy list initialization auto deduction will deduce a std::initializer\_list<T> if all elements in the list have the same type, or be ill-formed.  
4. for direct list initialization auto deduction will deduce a T if the list has a single element, or be ill-formed if there is more than one element.

```

1. auto a = {42}; // std::initializer_list<int>
2. auto b {42}; // int
3. auto c = {1, 2}; // std::initializer_list<int>
4. auto d {1, 2}; // error, too many

```

### 3.4.2 auto and function

#### 3.4.2.1 parameter

- auto can be used in function parameter, it is just another kind of template function.

```

1. void fun1(auto i){
2.     cout << i << endl;
3. }
4.
5. fun1(23); //produce two overload fun1 function.
6. fun1("abc");

```

- Example auto&& comes(forward reference) from generic lambda. It can reserve if fun is rvalue.

```

1. struct Functor{
2.     void operator ()() const & { std::cout << "lvalue_functor\n"; }

```

```

3. void operator ()() const && { std::cout << "rvalue_functor\n"; }
4. };
5.
6. int main() {
7.     //define two lambdas here
8.     auto perfectLambda = [] (auto&& func, auto&&... params) {
9.         std::forward<decltype(func)>(func)(
10.             std::forward<decltype(params)>(params)...
11.         );
12.     };
13.
14.     auto lambda = [] (auto func, auto&&... params) {
15.         func(std::forward<decltype(params)>(params)...);
16.     };
17.
18.     Functor fun;
19.     lambda(fun);
20.     lambda(Functor {});
21.     perfectLambda(fun);
22.     perfectLambda(Functor {});
23. }
```

### 3.4.2.2 Generalized return type deduction

- C++11 permitted automatically deducing the return type of a lambda function whose body consisted of only a single return statement:

```

1. [=] () -> some_type { return foo() * 42; } // ok
2. [=] { return foo() * 42; } // ok, deduces "-> some_type"
```

- This has been expanded in two ways. First, it now works even with more complex function bodies containing more than one return statement, as long as all return statements return the same type:

```

1. // C++14
2. [=] { // ok, deduces "-> some_type"
3.     while( something() ) {
4.         if( expr ) {
5.             return foo() * 42; // with arbitrary control flow
6.         }
7.     }
8.     return bar.baz(84); // & multiple returns
9. }
```

- Second, it now works with all functions, not just lambdas: Of course, this requires the function body to be visible.

```

1. auto fun1() {
2.     return 12;
3. }
```

- Below will produce compile error, because it has ambiguity.

```

1. auto f(int i){ // It will cause compile error.
2.     if ( i < 0 )
3.         return -1;
4.     else
```

```

5.     return 2.0
6. }
```

### 3.4.2.3 atuo in lambda and template

- the ability to directly hold closures(lambda), and in C++14, It can be used as lambda return type

```

1. auto derefUPLess = // comparison func.
2. [] (const std::unique_ptr<Widget>& p1, // for Widgets
3. const std::unique_ptr<Widget>& p2) // pointed to by
4. { return *p1 < *p2; }; // std::unique_ptrs
5.
6. //C++14 version
7. auto derefLess = // C++14 comparison
8. [] (const auto& p1, const auto& p2) // values pointed
9. { return *p1 < *p2; }; // to by anything pointer-like
```

- generic lambda( just like template lambda)

```
1. auto adder = [] (auto op1, auto op2){ return op1 + op2; };
```

- For template function return type, we can use two different ways.

- C++ 11, use auto + trailing type.

```

1. template <class T>
2. auto addFooAndBar(T const& t) -> decltype(t.foo() + t.bar()) {
3.     return t.foo() + t.bar();
4. }
```

- C++ 14, directly use auto.

```

1. template <class T>
2. auto addFooAndBar(T const& t) {
3.     return t.foo() + t.bar();
4. }
```

- use decltype(auto), detail can be found in Effective Modern C++ item 3.

- prefer to use function return type deduction wherever applicable, avoid trailing return type unless you really need them. they make your code harder to read**

## 3.5 summary

### 3.5.1 Syntax demo

- Some syntactic examples:

```

1. class A{
2. public:
3. A() {};
4. A(int k):m_a(k) {};
5. A(const A& rhs){m_a = rhs.m_a;};
6. int m_a;
7. };
```

```

8.
9. void fun(const A &a){}
10. int i = 3;

```

- Default ctor example

Expression	meaning
A a, A a{}	default ctor
A a()	<b>declare function a,vexing problem</b>
A()	A temporary A obj
B b(A())	<b>declare function, vexing problem</b> , A() is function pointer
B b(A{})	ctor a b with temp A.
fun(A()),fun(A{})	pass an parameter to fun

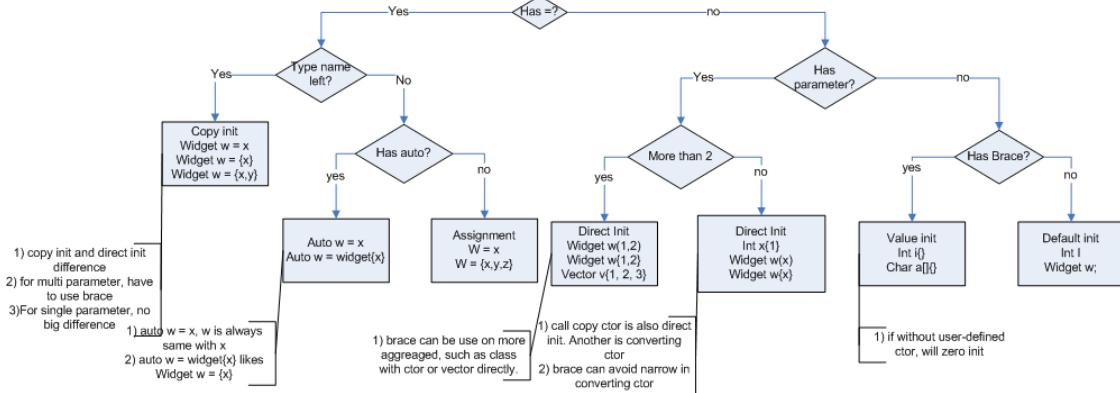
- single parameter ctor example

Expression	meaning
A a(i), A a{i}	parameterized ctor
A (i)	<b>just like A i, you have define i, so compile error.</b>
A{i}	A temporary A obj
B b(A(i))	declare a function B b(A i); vexing problem
B b(A{i})	build b with temp A
fun(A(i)),fun(A{i})	work, fun(A i) will not a declaration, because there are no return value.
B b(A a(i))	compile error, because A a(i) is not expression.
fun(A a(i))	same as above. <b>We need value for functoin parameter, not statement</b>

- If there is possible, it will use declaration first. . For B b(A a(i)), compiler will first interpret it as function declaration first, but A a(i) is not type, (A(i) and A() are both type). So it continue to interpret as B constructor, then it ask a value from expression. but A a(i) is not expression either, in the end, compiler is not happy.

### 3.5.2 How to understand init

- Understanding some basic conceptions. such as aggregate type. copy-list-init, converting ctor. All the high level usages are based on these basic conceptions.
- familiar with Six Init methods. You should know how to distinguish them. For example, when use = , it is called copy init.
- six methods and brace init are irrelevant. brace can be use in all six method and bring its own advantage. For example, there is direct-list-initializaiton and copy-list-initialization. You can see them in the cppreference.com. For differnt T type, it carry out different actions, such as carry out value init or call parameterized ctor.
- Summary picture:



### 3.5.3 How to init by yourself

- use `auto var = ...` style if possible, then use brace if possible, if there is `initializer_list`, considering use parentheses.
- Use braced initialization if possible.
  1. uniform style, this is useful when you write template program.
  2. avoid narrow
  3. For some certain situation, such as class and aggregation, only braced initialization works
  4. avoid most vexing parse problems.
- Pay attention two traps for `std::initializer_list`.
  1. If type has list initialization ctor, braced initialization will use it first, and very strongly. vector is good example to explain this trap.
  2. For auto type variable. (`auto x={1,2}` x always is `std::initializer_list` type.)

# Chapter 4

# Memory

## 4.1 malloc and remalloc

### 4.1.1 malloc

- That is a long story. In C language, if you forget a header file, what will happen? (I have add answer to my evernote). In one word, It will(c99) or will not produce(c89) warning, and compiler will assume a "implicit int" rule. That is to say, It presumes that the function without prototype (declaring in the header file) just return int type.
- continue: if you have code below: you comment stdlib.h. Then compiler will assume that malloc return a int. This code will run on 32 bit computer, because int and int\* have same length. but on some 64 bit computer, int\* is 64 bit and int is 32 bit. so half data will be lost, and your code will crash. And if you add (int\*) cast, it will suppress the warning message.(You kill the only clue, it's bad).

```
1. // include "stdlib.h"
2. // with cast, it will NOT produce a (missing prototype)warning .
3. int *sieve = (int *)malloc(sizeof(int)*length);
4.
5. // without cast, will produce a warning .
6. int *sieve = malloc(sizeof(int)*length);
```

- continue: So don't recommend to use (int\*) cast at all. Next question is that malloc actually return void\*. Can void\* automatically implicit be cast to int\* or other pointer type in assignment? the answer is **YES**.

- continue: In C language:

```
1. #define NULL ( (void*) 0)
2. int *p = NULL; //legal
3. FILE * f = NULL; //legal
```

- continue: In C++, int \*p = (void \*) 0 is not legal any more.(C++ is type safe language). So in C++ NULL is literal 0. But It produce another ambiguity problem. So In C++11, we define nullptr to resolve this problem.

```
1. #define NULL 0
2. int *p = NULL //legal
3.
4. f(int i)
5. f(int * p);
```

```

6.   f(NULL); //which one will be called. Answer is f(int i).
7.   f(nullptr) //will call f(int *p)
8.

```

- `calloc()` zero-initializes the buffer, while `malloc()` leaves the memory uninitialized. Zeroing out the memory may take a little time, so you probably want to use `malloc()` if that performance is an issue. If initializing the memory is more important, use `calloc()`. For example, `calloc()` might save you a call to `memset()`.
- With `memcpy`, the destination cannot overlap the source at all. With `memmove` it can. This means that `memmove` might be very slightly slower than `memcpy`, as it cannot make the same assumptions. For example, `memcpy` might always copy addresses from low to high. If the destination overlaps after the source, this means some addresses will be overwritten before copied. `memmove` would detect this and copy in the other direction - from high to low - in this case. However, checking this and switching to another (possibly less efficient) algorithm takes time.

#### 4.1.2 realloc

- Reallocates the given area of memory. It must be previously allocated by `malloc()`, `calloc()` or `realloc()` and not yet freed with a call to `free` or `realloc`. Otherwise, the results are undefined.
- Don't return value assign to input `ptr`, use another local pointer, such as `new_ptr`;

```

1. void *new_ptr = realloc(ptr, new_size);
2. if (!new_ptr) {
3.     // deal with error;
4. }
5. ptr = new_ptr

```

## 4.2 new operator

### 4.2.1 Basic

- There are four sub-topics about new operator:
  1. `new_handler`
  2. no throw, but return `nullptr`(since C++11)
  3. placement new (not allocate, just ctor)
  4. operator new (just allocate, no ctor)

```

1. std::set_new_handler(noMem);
2. MyClass * p1 = new MyClass; //1: if fail, call noMem
3.
4. MyClass * p2 = new (std::nothrow) MyClass; //2: no throw
5.
6. new (p2) MyClass; //3: placement new
7.
8. //4: operator new
9. MyClass * p3 = (MyClass*) ::operator new (sizeof(MyClass));
10. // allocates memory by calling: operator new
11. // but does not call MyClass's ctor

```

- **new** will call constructor function of a class, but **malloc** will not call constructor function. **So in C++, you should use new instead of malloc.**
- For default new operator, you can use **set\_new\_handler** to adjust the behavior when you can't allocate enough memory. Detail can be found in the below sub section.
- When you should provide your own operator new? You want to add log function; You want to add some cookie before and after allocated memory(Visual Studio use this way to detect overflow in debug mode). You want to have quicker speed(memory pool). You want to have better alignment and so on..
- If you use **new int[100]**, use **delete []**; **Any time you want to use new to allocate an array, ask yourself if you can replace it with vector or string.**

#### 4.2.2 Inside of new operator

- There is three level knowledges: 1) **new operator**, 2)**operator new**, 3)**new\_handler**.  
**new operator calling operator new, operator new calling new\_handler.**
- Basic logic of new operator and delete operator.

```

1. Point3d *origin = new Point3d;
2.
3. //C++ pseudo code
4. if(origin = operator new(sizeof(Point3d))){
5. try{
6. origin = Point3d::Point3d(origin);
7. }
8. catch(...){
9. operator delete(origin);
10. throw
11. }
12. }
```

```

1. delete origin;
2.
3. //C++ pseudo code
4. if (origin != NULL) {
5. origin->~Foo();
6. operator delete(origin);
7. }
```

1. call operator new to allocate space in memory. (different with new opeator), in this way, ctor will not be called.
2. call ctor of a class. (You can't call ctor explicit in this way, but compiler can.)
3. If ctor throw an exception, no memory leak by calling operator delete.
4. So for a class, if you declare its dtor private or protected, You can't use "delete p".

- Basic logic of operator new and operator delete

```

1. void * operator new(std::size_t size) throw(std::bad_alloc) {
2. // your operator new might
3. using namespace std; // take additional params
4.
5. if (size == 0) { // handle 0-byte requests
```

```

6.    size = 1;           // by treating them as
7.    }                   // 1-byte requests
8.    void *last_alloc;
9.    while (true) {
10.        *last_alloc = malloc(size)
11.        if (last_alloc)
12.            return last_alloc;
13.
14.        // allocation was unsuccessful; find out what the
15.        // current new-handling function is (see below)
16.        new_handler globalHandler = set_new_handler(0);
17.        set_new_handler(globalHandler);
18.
19.        if (globalHandler)
20.            (*globalHandler)();
21.        else
22.            throw std::bad_alloc();
23.
24.    }

```

```

1. operator delete (void *ptr){
2.     if (ptr)
3.         free(ptr)
4.

```

1. if size is 0, change it to 1
2. Why we need to call `set_new_handler` twice, The first one get the current handler, and the second one change it back. That is only way we can get the current handler.
3. This idiom is also used in `set_terminate` and `set_unexpect`.
4. In C++11, introduce `get_new_handler()` function. We don't need to call `set_new_handler` twice.
5. Most of time, we use `malloc` and `free` to allocate and free physical memory.

#### 4.2.3 new\_handler

- At program startup, `new-handler` is a null pointer. allocation function finds that `std::get_new_handler` returns a null pointer value, it will throw `std::bad_alloc`.
- As shown by the previous section, The `new_handler` function is the function called by allocation functions whenever a memory allocation attempt fails. Its intended purpose is one of three things:
  1. make more memory available. **resolve the problem by myself**.
  2. throw exception of type `std::bad_alloc` or derived from `std::bad_alloc`. **resolve the problem by a user**.
  3. terminate the program. e.g. by calling `std::terminate`. (**No resolve**)

```

1. void noMemory() {
2.     closeIE; // method 1 release mem
3.     set_new_handler(nullptr); //method 2 throw bad_alloc exception.
4.     abort(); //method 3, end application.
5. }
6. //in the beginning of main() function.
7. set_new_handler(noMemory)

```

- What can we do in side new handler function? Below choices give you considerable flexibility in implementing new-handler functions.
  1. Make more memory available. This may allow the next memory allocation attempt inside operator new to succeed. One way to implement this strategy is to allocate a large block of memory at program start-up, then release it for use in the program the first time the new-handler is invoked.
  2. Install a different new-handler. If the current new-handler can't make any more memory available, perhaps it knows of a different new-handler that can. If so, the current new-handler can install the other new-handler in its place (by calling `set_new_handler`). The next time operator new calls the new-handler function, it will get the one most recently installed. (A variation on this theme is for a new-handler to modify its own behavior, so the next time it's invoked, it does something different. One way to achieve this is to have the new-handler modify static, namespace-specific, or global data that affects the new-handler's behavior.)
  3. Deinstall the new-handler, i.e., pass the null pointer to `set_new_handler`. With no new-handler installed, operator new will throw an exception when memory allocation is unsuccessful.
  4. Throw an exception of type `bad_alloc` or some type derived from `bad_alloc`. Such exceptions will not be caught by operator new, so they will propagate to the site originating the request for memory.
  5. Not return, typically by calling `abort` or `exit`.
- If you want to have customized `new_handler` for specific class, "effective C++ item 49" gives a good example. It use Mixin idiom. Detail can be found in generic programming section in this book.

#### 4.2.4 placement new

- construct an object in memory you've already got a pointer to ,use placement new. If you use placement new to create an object in some memory, you should avoid using the delete operator on that memory. Detail can be seen in C++ primer

```

1. Foo* p;
2. // don't catch exceptions thrown by the allocator itself
3. // Return a void type pointer
4. void* raw = operator new(sizeof(Foo)*100);
5.
6. // catch any exceptions thrown by the ctor
7. try {
8.     p = new(raw) Foo(); // call the ctor with raw as this
9. }
10. catch (...) {
11.     // oops, ctor throw an exception
12.     operator delete(raw);
13.     throw; // rethrow the ctor's exception
14. }
```

- Another example to use placement new.

```

1. void* buffer = operator new(sizeof(FOO)*100);
2. p1 = new(buffer) FOO();
3. p2 = new(buffer+sizeof(FOO)) FOO();
```

```

4. p1->~FOO(); // call dtor directly ,
5. p2->~FOO(); //don't call delete p1.
6.
7.
8. delete [] buffer ;

```

- placement operator new is defined as below. **You are not allowed to customize it.**

```

1. void operator new(size_t , void* p){
2.     return p;
3. }

```

- You can build a object on an existing object. Pay attention, this is only place where you can call destructor directly. Don't call delete.

```

1. FOO* p1 = new( buffer ) FOO();
2. ....
3. p1->~FOO();
4. //Don't use delete p1!!!
5.
6. //A new object in an existing object .
7. p1 = new( buffer ) FOO();

```

- If operator new receive another parameter beside that size\_t, that is placement new. There is special version with void\* pMemeory

```

1. void* operator new ( std::size_t size , void* pMemeory ) throw();

```

- If you define your own placement operator new(non-pMemeory version). You must provide your own placement operator delete. Because when the constructor throw exception, it will call the corresponding customized placement operator delete. Detail can be found in "effective C++ item 52"

```

1. void* operator new ( std::size_t size , void* pMemeory ) throw();
2. void* operator new ( std::size_t size , ostream& logStream ) throw();
3. Widget* pw = new( std::cerr ) Widget
4. //When Widget ctor throw exception , it will call
5. void operator delete(void* pM, ostream& logStream).
6. //If no such function , placement new do nothing and memory leak .

```

- A good article is "The many faces of operator new in C++", It give detail information about operator new and how to rewrite it. I have added it to my ref.

#### 4.2.5 array new

- **When you use array new to allocate an arry, must use the array delete.** If you don't use array delete, maybe you just delete the frist object in array. If you use array delete to single new, it's undefined.

```

1. Foo pa* = new Foo[10];
2. delete [] pa;
3. //system will remember the size corresponding with pa ,
4. //with [], it will iterate with size .
5. //When you forget [], it will just free the first object .

```

- Basic logic of array new. An basic implementation can be found in "Inside the C++ Object Model" 6.2 chapter

```

1. vec_new(int elem_count, int size, funptr ctor){
2.     total_size = size*elem_count;
3.     ptr_array = new char[total_size];
4.     regist pair of ptr_array elem_count to system
5.     while(elem<end of address){
6.         (*ctor)(elem) //call the ctor
7.         elem+=size;
8.     }
9. }
```

- When you use array new with inheritance. There is one important thing to notice. Don't use base pointer to point the array with derived class.

```

1. base *bp = new derived[10]
2. //always use derived *dp = new derived[10];
3. bp[2] //dangerous, undefine. size is wrong: bp+sizeof(base)*2
4.
5. delete [] bp //dangerous, 1)size is wrong,
6. 2) it will call base::~base() destructor.
```

- Above code, Don't use pointer to understand it, When you use delete [] bp, it will think that is a base array, and each element in it is just base object, so virtual function doesn't play a role here. Please refer the section "inheritance" in this book for more detail.

#### 4.2.6 Customize operator new

- The operator new and nothrow version are also replaceable: A program may provide its own definition that replaces the one provided by default to produce the result described above, or can overload it for specific types.
- For placement new, You can't replace it. There is no "operator new array", it use operator new to allocate memory.

```

1. void* operator new (std::size_t size) throw (std::bad_alloc);
2. void* operator new (std::size_t size,
3. const std::nothrow_t& nothrow_value) throw();
4.
5. void* operator new (std::size_t size, void* ptr) throw();
```

- You can't change "new operator" behavior, but you can override global "operator new" and overload class its own "operator new". Pay attention its argument and return void\*.

```

1. //global operator new
2. void* operator new(std::size_t size){
3. cout<<"Yan's own operator new" ;
4. void* mem = malloc(size);
5. if(mem)
6.     return mem;
7. else
8.     throw bad_alloc();
9.
10.
11. class Foo{ //class operator new
12. public:
```

```

13. static void* operator new(size_t size);
14. // use static
15. }
16.
17. void* Foo::operator new(size_t size){
18. cout<<"Foo's own operator new";
19. ....
20. }
21.
22. int *p = new int[100]; //output Yan's own operator new
23. Foo* fp = new Foo(); //output Foo's own operator new

```

- Above code is just a simple demo, In the new\_handler section, you can see a better operator new demo with support of call your own new\_handler. You need to combine above code and code in new\_handler section together.
- operator new usually call malloc function. malloc usually call brk for small chunk and mmap for big chunk. So in the end, C langauge is basic langauge.
- Why do I need my own operator new?
  1. Performance: the default memory allocator is designed to be general purpose. Sometimes you have very specific objects you want to allocate, chapter 4 in "Modern C++ Design" presents a very well designed and implemented custom allocator for small objects.
  2. Debugging & statistics: having full control of the way memory is allocated and released provides great flexibility for debugging, statistics and performance analysis.
  3. Customization to cluster related object together, and reduce size. put guard block to avoid overrun and underrun. More detail can be seen in effective c++( third edition) Item 50
- **Don't rewrite operator new unless you have to.** There are not as easy as you think. such as alignment. First consider some library, such as **Boost Pool library** for large number small object allocations.
- If you have to rewrite operator new, You need to read effective c++( third edition) Item 51 in detail. For example, All operator new should contain a loop calling a new-handling function. should deal with request of zero size. you can see the pseudocode in Item 51.
- In C++, after you define a name in a scope (e.g., in a class scope), it will hide the same name in all enclosing scopes (e.g., in base classes or enclosing namespaces), and overloading never happens across scopes. And when said name is operator new, If you provide any class-specific new, provide all of the standard forms(plain, in-place, and nothrow)

```

1. class C {
2. static void* operator new(size_t , MemoryPool&);
3. // hides three normal forms
4. };
5. //1) plain new
6. void* operator new(std::size_t );
7.
8. //2) nothrow new
9. void* operator new(std::size_t , std::nothrow_t) throw();
10.
11. //3) inplace new
12. void* operator new(std::size_t , void*) ;

```

- Always provide new and delete together, see "C++ coding standards" item 45 and 46.

# Chapter 5

## pointer and smart pointer

### 5.1 pointer and new

#### 5.1.1 function pointer

- A real example of function pointer is set\_new\_handler. It accepts a function pointer and return the same function pointer, so declaring such format is a little difficult.

```
1. void failNew() {
2.     cerr<<"Fail now"<<endl
3.     abort();
4. }
5. -----
6. extern void (*set_new_handler( void (*)() ) ) ();
7.
8. // This function declaration is very complex.
9. set_new_handler(failNew);
```

- A better method is to use typedef method.

```
1. typedef void(* FunPtr)();
2. FunPtr (*set_new_handler)(FunPtr);
3. set_new_handler(failNew);
```

#### 5.1.2 When to use new?

- When do you use smart pointer? Here I change this questions to another question, When do you use pointer?

1. An new object remain in existence, until you delete it. (you control the life time of it, maybe you create it in fun1,then delete it in fun4. so the obj is neither stack life nor global life. You may or may not transfer **ownership** between functions.)
2. When you want to create a obj in runtime according runtime condition or user input dynamically. (maybe don't create it. ) Below code demonstrates previous two conditions. ToothBrush need to be control life time and created dynamically.

```
1. class Person{
2.     ToothBrush* pbrush;
3.
4.     buyNewBrush( string &name){
5.         if(pbrush != nullptr){
6.             delete pbrush;
```

```

7.     }
8.     if(name == "Orlab")
9.         pbrush = new Brush();
10.    }
11.}
~Person() {
13.    if(pbrush != nullptr){
14.        delete pbrush;
15.    }
16.}
///////////////
17. Person Yan();
18. Yan.buyNewBrush("Orlab");
19. .... Three months later .....
20. Yan.buyNewBrush("Philip");
21.
22.

```

3. Large array or resources, if you allocate in stack, it will cause stack overflow. **In practices, this requirement is deprecated, because any time when you use [] or new [], you need consider to use array or vector ans string!**

## 5.2 basic smart pointer knowledge

### 5.2.1 Smart pointer Basic knowledge

- A simple auto\_ptr source code: You can learn how to overload operator\*() and operator->().

```

1. template <class T> class auto_ptr{
2.     T* ptr;
3. public:
4.     explicit auto_ptr(T* p = 0) : ptr(p) {}
5.     ~auto_ptr() {delete ptr;}
6.     T& operator*() {return *ptr;}
7.     T* operator->() {return ptr;}
8.     // ...
9. };
10.
11. auto_ptr<int> aupr;
12. //aupr is nullptr even you don't initialize it.

```

- Why do we need smart pointer?

1. "Just remember" is seldom the best solution! So we need smart pointer to perform delete operator automatically.
2. When throw an exception, or return in the middle of code. delete will not be invoked at all, It will cause memory leaking. When you use smart pointer, If an exception is thrown in the middle of fun, there will no be memory leak.

```

1. void methodA() {
2.     unique_ptr<int> buf(new int[256]);
3.
4.     int result = fillBuf(buf)
5.     if(result == -1)
6.         return;
7. }

```

3. smart pointer is nullptr default if you don't initialize. You avoid wild pointer problem.

- 4. smart pointer can manage exclusive ownership or shared ownership automatically.
- There are four kinds of smart pointer: `auto_ptr` `unique_ptr` `shared_ptr` and `weak_ptr` But only `unique_ptr`, `shared_ptr` and `weak_ptr` are recommended to use. `auto_ptr` is now deprecated, and should not be used in new code. When you get a chance, try doing a global search-and-replace of `auto_ptr` to `unique_ptr` in your code base. `weak_ptr` is mainly used for observer, you can always use raw pointer or reference for this purpose. Raw pointer can't check if pointee object is still valid, `weak_ptr` can accomplish this task.

- Why `auto_ptr` not recommended?

1. When you assign `targetP = sourceP`, it will cause `sourceP` set to be NULL, It will cause trouble when you use `sourceP` in the future.
2. You can't create container includes `auto_ptr`, compiler prohibit you doing so!

```

1. auto_ptr<string> ps (new string ("hello_world") );
2. auto_ptr<string> ps1;
3. ps1 = ps; // ps will be set null.
4. ps->size() // it will crash the application .
5.
6. auto_ptr<string> parray [5]; //compiling error .
7. auto_ptr<string> ps = parray [2];
8. //parray [2] will be set null;
```

- When you construct a smart pointer, you must use 1) a pointer and 2) this pointer must be produced by `new`. You can't build smart pointer by address operator. such as `unique_ptr<double> ptr(&int);` Why? because smart pointer will call `delete` when it is out of scope. `delete` operator has to be used on pointer produced by `new` operator.
- Obtain the raw pointer (`get`), to relinquish control of the pointed object (`release`), and to replace the object it manages (`reset`).

```

1. string * cp = new string ("hello_world");
2. shared_ptr<string> ps(cp);
3. string * cp1 = ps.get(); //use get() get normal pointer .
```

- Smart pointer has `bool` operator, you can use `if` to test if it's `nullptr` directly.

```

1. std :: unique_ptr<int> ptr(new int(42));
2. if (ptr) std :: cout << *ptr << '\n';
3. ptr.reset();
4. if (ptr) std :: cout << *ptr << '\n';
```

- `shared_ptr<T>` and `shared_ptr<const T>` are not interchangable. It goes one way - `shared_ptr<T>` is convertable to `shared_ptr<const T>` but not the reverse.

```

1. shared_ptr<int> pint (new int(4));
2. // normal shared_ptr
3. shared_ptr<const int> pcout = pint;
4. // shared_ptr<const T> from shared_ptr<T>
5. shared_ptr<int> pint2 = pcout; // error! comment out to compile
```

- Smart pointer and const:

```

1. shared_ptr<T> p;      —> T * p;
2. const shared_ptr<T> p;  —> T * const p;
3. shared_ptr<const T> p; —> const T * p;
4. const shared_ptr<const T> p; —> const T * const p;

```

## 5.2.2 unique\_ptr

### 5.2.2.1 basic

- Basic 1-1: create `unique_ptr` from `new`. `make_unique` is better than inside `new`, inside `new` is better than outside `new`.

```

1. string * cp = new string("hello_world");
2.
3. unique_ptr<string> ps = cp; // NOT allow
4.
5. //ok, but not good style(outside new)
6. unique_ptr<string> ps(cp);
7.
8. //good style (inside new)
9. unique_ptr<string> ps(new string("hello_world"));
10.
11. //best style make_unique
12. unique_ptr<string> ps(std::make_unique<string>("hello"));

```

- Basic 1-2: Create `unique_ptr` from another `unique_ptr`, you have to use move.

```

1. unique_ptr<string> ps1(new string("hello_world"));
2.
3. unique_ptr<string> ps2(ps1); //compile error, not allow.
4. unique_ptr<string> ps2(move(ps1)); //ok
5. //ownership transfer from ps1 to ps2, nothing delete.

```

- Basic 2: When pass value into a function, 1) for some exist `unique_ptr`, use `move` to transfer ownership, 2) for new `unique_ptr`, use `make_unique` to assure exception safe.

```

1. void sink(unique_ptr<widget> arg1, unique_ptr<gadget> arg2);
2.
3. //1) use move
4. sink(std::move(exist_uptr_wi), std::move(exist_uptr_ga))
5.
6. //2) use make function to assure exception safety.
7. sink(make_unique<widget>(new ...),
8. make_unique<gadget>(new ...)); // exception-safe

```

- Basic 3: `unique_ptr` assignment.

```

1. unique_ptr<string> ps1(new string("ps1"));
2. unique_ptr<string> ps2(new string("ps2"));
3.
4. ps1 = ps2; //compile error, not allow
5.
6. //method1: use move
7. ps1 = std::move(ps2);
8. //pointer inside previous ps1 will be deleted.
9. //pointer inside ps1 point to "ps2" string now.
10. //pointer inside ps2 will set to null

```

```

11. //method2: reset
12. ps1.reset(cp); //ok
13. //pointer inside previous ps1 will be delete
14.
15.
16. string* pstr = ps1.release();
17. // use pstr get pointer managed by ps1.

```

- Basic 4-1: If a program attempts to assign one unique\_ptr to another. The compiler allows it if the source object is a temporary rvalue (It will call move ctor or assignment of unique\_ptr inside.) and disallows it if the source object has some duration. **It is a move-only type.**

```

1. unique_ptr<string> ps2 = unique_ptr<string>(new string("yo")); //OK
2.
3. unique_ptr<string> fun() {
4.     return unique_ptr<string> temp(new string("yan"));
5. }
6. pu2 = fun(); //OK

```

- Basic 4-2: unique\_ptr support source and sink idiom.

```

1. unique_ptr fun() //support source
2.
3. fun(unique_ptr up); //use move to support sink
4. fun(move(other_up));

```

- Basic 5: Just observer, not transfer ownership , you can get pointer, or use unique\_ptr reference.

```

1. unique_ptr<string> ps1 (new string("ps1"));
2.
3. fun(string* pstr); //ps1.get()
4. fun(unique_ptr<string> & ref_ptr);

```

- By default, `std::unique_ptrs` are the same size as raw pointers, so efficiency of it is just like raw pointer.
- Unique\_ptr has new [] version. The existence of std::unique\_ptr for arrays should be of only intellectual interest to you, because std::array, std::vector, and std::string are virtually always better data structure choices than raw arrays. About the only situation I can conceive of when a `std::unique_ptr<T[]>` would make sense would be when you're using a C-like API that returns a raw pointer to a heap array that you assume ownership of it.

```

1. unique_ptr<double []> pda(new double[5] );
2. // it will call delete [] inside.

```

- When in doubt, prefer `unique_ptr` by default, and you can always later move-convert to `shared_ptr` if you need it. If you do know from the start you need shared ownership, however, go directly to `shared_ptr` via `make_shared`. If you compiler report error about `unique_ptr`, then you should consider to use `shared_ptr`
- `unique_ptr` can be used inside of class. Below class B disable value-copying (or to define a suitable copy-constructor and operator= to handle it safely).

```

1. class B { // this class can't be copy
2. public:
3.     unique_ptr<int> i;
4.     B() : i(new int(0)) { }
5. };

```

- A common use for `std::unique_ptr` is as a factory function return type for objects in a hierarchy. Detail can be seen "effective modern c++ item 18"

1. During construction, `std::unique_ptr` objects can be configured to use custom deleters: arbitrary functions (or function objects, including those arising from lambda expressions) to be invoked when it's time for their resources to be destroyed. When a custom deleter can be implemented as either a function or a captureless lambda expression, the lambda is preferable.
2. When a custom deleter is to be used, its type must be specified as the second type argument to `std::unique_ptr`.

```

1. //only c++ 14 support return type deduction
2. template<typename ... Ts>
3. auto makeInvestment(Ts&&... params) {
4.     auto delInvmt = [](Investment* pInvestment) {
5.         makeLogEntry(pInvestment); // makedelete
6.         delete pInvestment; // Investment
7.     };
8.
9.     std::unique_ptr<Investment, decltype(delInvmt)>
10.    pInv(nullptr, delInvmt);
11.
12.    if (...) {
13.        pInv.reset(new Stock(std::forward<Ts>(params)...));
14.    } else if (...) {
15.        pInv.reset(new Bond(std::forward<Ts>(params)...));
16.    }
17.    return pInv; // as before
18. }
19.

```

- `std::unique_ptr` is the C++11 way to express exclusive ownership, but one of its most attractive features is that it easily and efficiently converts to a `std::shared_ptr`: This is a key part of why `std::unique_ptr` is so well suited as a factory function return type. Factory functions can't know whether callers will want to use exclusive ownership semantics for the object they return or whether shared ownership

```

1. std::shared_ptr<Investment> sp = // converts std::unique_ptr
2. makeInvestment( arguments ); // to std::shared_ptr

```

### 5.2.2.2 `unique_ptr` and container

- If your vector should hold `std::unique_ptr<Fruit>` instead of raw pointers (to prevent memory leaks). vector need copy in and copy out. but `unique_ptr` don't support copy. So

1. you can use `emplace_back` with new pointer, but it will lead memory leak if extending vector size fail.

2. use `unique_ptr`.

3. use `make_unique`

```

1. class Fruit { ... };
2. class Pear : Fruit { ... };
3. class Tomato : Fruit { ... };
4.
5. std::vector<std::unique_ptr<Fruit>> m_fruits;
6. //method 1, bad
7. m_fruits.emplace_back(new Pear);
8.
9. //method 2, good
10. m_fruits.push_back(std::unique_ptr<Fruit>(new Pear));
11. m_fruits.push_back(std::unique_ptr<Fruit>(new Tomato));
12.
13. //method 3, best using std::make_unique:
14. m_fruits.push_back(std::make_unique<Pear>());
15. m_fruits.push_back(std::make_unique<Tomato>());
```

- You can store `unique_ptr` objects in an STL container providing you don't invoke methods or algorithm, such as `copy()`, that copy or assign one `unique_ptr` to another. see effective stl item 8.

```

1. bool compare_by_uniqptr(
2.     const unique_ptr<SomeLargeData>& a,
3.     const unique_ptr<SomeLargeData>& b) {
4.     return a->id < b->id;
5. }
6.
7. sort(vec_byuniptr.begin(), vec_byuniptr.end(),
8. compare_by_uniqptr);
```

```

1. typedef std::unique_ptr<int> unique_t;
2. typedef std::vector<unique_t> vector_t;
3.
4. vector_t vec2(5, unique_t(new Foo)); // Error (Copy)
5. vector_t vec3(vec1.begin(), vec1.end()); // Error (Copy)
6. std::copy(vec1.begin(), vec1.end(),
7.           std::back_inserter(vec2)); // Error (copy)
8.
9. vector_t vec3(make_move_iterator(vec1.begin()),
10.               make_move_iterator(vec1.end())); // Ok
11.
12. std::sort(vec1.begin(), vec1.end());
13. // OK, because using Move Assignment Operator
```

### 5.2.3 `shared_ptr`

- `shared_ptr` basic 1: create `shared_ptr` from `new`. **`make_shared` is better than inside `new`, inside `new` is better than outside `new`.**

```

1. string * cp = new string("hello_world");
2.
3. shared_ptr<string> ps = cp; // NOT allow
4.
5. //method1: use constructor
6. shared_ptr<string> ps(cp); //ok, but not good style
7. shared_ptr<string> ps(new string("hello_world")); //good style
```

```

8. //method2: make_shared function
9. unique_ptr<string> ps( std::make_shared<string>("hello_world") );
10.

```

- shared\_ptr basic 1-1: First, try to avoid passing raw pointers to a std::shared\_ptr constructor.  
1) use make\_share(). 2) if you have custom deleter and can't use make\_share(). Pass the result of new directly instead of going through a raw pointer variable.

```

1. auto pw = new Widget; // pw is raw ptr
2.
3. std::shared_ptr<Widget> spw1(pw, loggingDel);
4. // create control block for *pw
5.
6. std::shared_ptr<Widget> spw2(pw, loggingDel);
7. // create 2nd control block for *pw!
8. // BAD! That will cause undefined result!

```

- shared\_ptr basic 2: Create shared\_ptr from another shared\_ptr

```

1. shared_ptr<string> ps1 (new string("hello_world"));
2.
3. shared_ptr<string> ps2 ( ps1 );
4. //use_count of ps1 and ps2 are all 2
5.
6. shared_ptr<string> ps2 ( move(ps1) );
7. //the original ps1 will become null, and
8. //the reference count does not get modified.
9. //Just transfer use_count to ps2.

```

- shared\_ptr basic 3: shared\_ptr assignment

```

1. shared_ptr<string> ps1 (new string("ps1"));
2. shared_ptr<string> ps2 (new string("ps2"));
3.
4. //method1: use assignment
5. ps1 = ps2;
6. // 1) previous use_count decrements 1 (If equal 0, will delete)
7. // 2) ps1 points to current use_count
8. // 3) and current use_count increments 1
9.
10. //method2: use move
11. ps1 = std::move(ps2);
12. // 1) previous use_count decrements 1 (If equal 0, will delete)
13. // 2) ps1 points to current use_count
14. // 3) ps2 is null now.
15.
16. //method3: reset
17. ps1.reset(cp); //ok
18. //pointer inside previous ps1 will decrement 1.
19. //by now, current ps1 reference count will be 1.

```

- shared\_ptr basic 4: Difference between ctor and assignment.

1. When assignment, previous decrements 1 and current increment 1;
2. When copy ctor, current increment 1;
3. When ctor from raw pointer, current is 1;

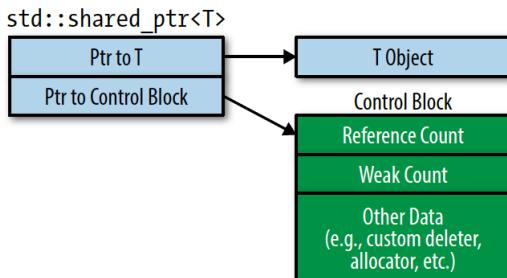
- shared\_ptr basic 5: Just observer, not transfer ownership , you can get pointer, or use shared\_ptr reference. Or use reference directly!

```

1. shared_ptr<string> ps1 (new string ("ps1")) ;
2.
3. fun( string * pstr );
4. fun( shared_ptr<string>& ref_ptr );

```

- inside of shared\_ptr:



- **When are control blocks created?** It's very important conception, when you understand it, you will know what happen when you create or assign a shared\_ptr better.

1. std::make\_shared always creates a control block. It manufactures a new object to point to, so there is certainly no control block for that object at the time std::make\_shared is called.
  2. A control block is created when a std::shared\_ptr is constructed from a unique-ownership pointer (i.e., a std::unique\_ptr). As part of its construction, the std::shared\_ptr assumes ownership of the pointed-to object, so the uniqueownership pointer is set to null.
  3. When a std::shared\_ptr constructor is called with a raw pointer, it creates a control block.
  4. std::shared\_ptr constructors taking std::shared\_ptrs or std::weak\_ptrs as constructor arguments. **NOT** create new control blocks, because they can rely on the smart pointers passed to them to point to any necessary control blocks
- Like std::unique\_ptr, std::shared\_ptr uses delete as its default resource-destruction mechanism, but it also supports custom deleters.

```

1. auto loggingDel = [](Widget *pw){ // custom deleter
2.     makeLogEntry(pw);
3.     delete pw;
4. };
5.
6. std::unique_ptr<Widget, decltype(loggingDel)>
7.     upw(new Widget, loggingDel);
8. // deleter type is ptr type
9.
10. std::shared_ptr<Widget> spw(new Widget, loggingDel);
11. // deleter type is not part of ptr type

```

- In order to correctly use shared\_ptr with an array, you must supply a custom deleter. But we don't recommend using shared\_ptr with array. **Any time you new a array, you should first consider using STL container directly.**

```

1. template< typename T >
2. struct array_deleter {

```

```

3.     void operator ()( T const * p){
4.         delete [] p;
5.     }
6. };
7.
8. std::shared_ptr<int> sp( new int[10], array_deleter<int>() );

```

- Usage of `enable_shared_from`. If you don't use it, multiple distinct `shared_ptr` objects with separate reference counts. For this reason you must never create more than one `shared_ptr` from the same raw pointer. It has become C++11 standard.

```

1. class Test : public boost::enable_shared_from_this<Test> {
2. public:
3.     boost::shared_ptr<Test> GetObject()
4.     {
5.         return shared_from_this();
6.         //return shared_ptr<test>(this); You can't do this way
7.         //Anytime you create shared_ptr
8.     }
9. };
10. int main(int argc, char *argv[])
11. {
12.     boost::shared_ptr<Test> p( new Test() );
13.     boost::shared_ptr<Test> q = p->GetObject();
14.
15.     return 0;
}

```

### 5.2.4 weak\_ptr

- The relationship begins at birth. `std::weak_ptrs` are typically created from `std::shared_ptrs`. You can only create a `weak_ptr` out of a `shared_ptr` or another `weak_ptr`. So the idea would be that the owner of the pointer hold a `shared_ptr` instead of a raw pointer.
- `std::weak_ptr`s can't be dereferenced, nor can they be tested for nullness. That's because `std::weak_ptr` isn't a standalone smart pointer. It's an augmentation of `std::shared_ptr`. Almost the only things you can do are to interrogate it to see if the managed object is still there, or construct a `shared_ptr` from it.

```

1. auto spw =
2. std::make_shared<Widget>();
3. // the pointed-to Widget's ref count (RC) is 1.
4.
5. std::weak_ptr<Widget> wpw(spw);
6. // wpw points to same Widget as spw. RC remains 1
7. ...
8. spw = nullptr;
9. // RC goes to 0, and the Widget is destroyed.
10. // wpw now dangles
11. if (wpw.expired())
12. // if wpw doesn't point to an object

```

```

1. std::shared_ptr<Widget> spw1 = wpw.lock();
2. // if wpw's expired, spw1 is null
3. auto spw2 = wpw.lock();
4. // same as above, but uses auto
5.
6. std::shared_ptr<Widget> spw3(wpw);
7. // if wpw's expired, throw std::bad_weak_ptr

```

- Potential use cases for std::weak\_ptr includes: caching, observer lists, and the prevention of std::shared\_ptr cycles. All the detail can be seen in "Effective Modern ++" Item 20.

```

1. std::shared_ptr<const Widget> fastLoadWidget(WidgetID id) {
2.     static std::unordered_map<WidgetID,
3.         std::weak_ptr<const Widget>> cache;
4.     auto objPtr = cache[id].lock();
5.     // objPtr is std::shared_ptr to cached object
6.     //(or null if object's not in cache)
7.     if (!objPtr) { // if not in cache,
8.         objPtr = loadWidget(id); // load it
9.         cache[id] = objPtr; // cache it
10.    }
11.    return objPtr;
12. }
```

- For above requirement, you also can use raw pointer, But for raw pointer, you can't detect if original one has been delete.
- A truly smart pointer would deal with this problem by tracking when it dangles, i.e., when the object it is supposed to point to no longer exists. That's precisely the kind of smart pointer std::weak\_ptr is. **You can't test if a raw pointer is dangle or not.**

### 5.2.5 make function

- **Never say new in c++14!** Use make function instead of new operator.
- Don't use make\_unique if you need a custom deleter or are adopting a raw pointer from elsewhere.
- make\_unique doesn't joined the Standard Library until c++14.
- make\_unique just perfect-forwards its parameters to the constructor of the object being created, constructs a std::unique\_ptr from the raw pointer new produces, and returns the std::unique\_ptr so created. **make\_unique doesn't support arrays or custom deleters.**
- std::make\_unique and std::make\_shared are two of the three make functions: functions that take an arbitrary set of arguments, perfect-forward them to the constructor for a dynamically allocated object, and return a smart pointer to that object. The third make function is std::allocate\_shared
- make function has tree advantage: simply, exception safety and allocate once for efficiency.

1. The method of using new repeat the type being created, but the make functions don't.

```

1. auto upw1(std::make_unique<Widget>()); // with make func
2. std::unique_ptr<Widget> upw2(new Widget); // without make func
3.
4. auto spw1(std::make_shared<Widget>()); // with make func
5. std::shared_ptr<Widget> spw2(new Widget); // without make func
```

2. The second reason to prefer make functions has to do with exception safety.
  3. It's obvious that this code entails a memory allocation, but it actually performs two. Item 19 explains that every std::shared\_ptr points to a control block containing, among other things, the reference count for the pointed-to object. That's because std::make\_shared allocates a single chunk of memory to hold both the Widget object and the control block.
- make function has its limitation:

1. For example, none of the make functions permit the specification of custom deleters.
2. the perfect forwarding code uses parentheses, not braces. The bad news is that if you want to construct your pointed-to object using a braced initializer, you must use new directly. Using a make function would require the ability to perfect-forward a braced initializer, but, as Item 30 explains, braced initializers can't be perfect-forwarded.

```

1. auto upv = std::make_unique<std::vector<int>>(10, 20);
2. //upv has 10 elements, each one is 20.
3.
4. // create std::initializer_list
5. auto initList = { 10, 20 };
6. // create std::vector using std::initializer_list ctor
7. auto spv = std::make_shared<std::vector<int>>(initList);

```

3. As long as std::weak\_ptrs refer to a control block (i.e., the weak count is greater than zero), that control block must continue to exist. And as long as a control block exists, the memory containing it must remain allocated. The memory allocated by a std::shared\_ptr make function, then, can't be deallocated until the last std::shared\_ptr and the last std::weak\_ptr referring to it have been destroyed

- If you can't use make function, you have to create shared\_ptr first, then pass it to function, but a better way is to move it.

```

1. std::shared_ptr<Widget> spw(new Widget, cusDel);
2. processWidget(spw, computePriority());
3. // correct, but not optimal; see below
4.
5. processWidget(std::move(spw), computePriority());
6. // both efficient and exception safe

```

## 5.2.6 wrapping resource handler in smart pointer

### 5.2.6.1 basic idea

- The std::unique\_ptr template has two parameters: the type of the pointee, and the type of the deleter. This second parameter has a default value, so you usually just write something like std::unique\_ptr<int>.
- The std::shared\_ptr template has only one parameter though: the type of the pointee. But you can use a custom deleter with this one too, even though the deleter type is not in the class template. The usual implementation uses type erasure techniques to do this.
- Part of the reason is that shared\_ptr needs an explicit control block anyway for the ref count and sticking a deleter in isn't that big a deal on top. unique\_ptr however doesn't require any additional overhead, and adding it would be unpopular- it's supposed to be a zero-overhead class. unique\_ptr is supposed to be static.
- You can always add your own type erasure on top if you want that behaviour- for example, you can have unique\_ptr<T, std::function<void(T\*)>>, something that I have done in the past.
- **shared\_ptr and std::functions use type erase technology.** About type erase, you can see the generic programming chapter.
- A common example can be found here:

```

1. typedef struct {
2.     int m_int;
3.     double m_double;
4. } Foo;
5.
6. Foo* createObject(int i_val, double d_val) {
7.     Foo* output = (Foo*) malloc(sizeof(Foo));
8.
9.     output->m_int = i_val;
10.    output->m_double = d_val;
11.
12.    puts("Foo created.");
13.    return output;
14. }
15.
16. void destroy(Foo* obj) {
17.     free(obj);
18.     puts("Foo destroyed.");
19. }
20.
21. std::shared_ptr<Foo> foo(createObject(32, 3.14), destroy);

```

```

1. struct FooDeleter {
2.     void operator()(Foo* p) const {
3.         destroy(p);
4.     }
5. };
6.
7. using FooWrapper = std::unique_ptr<Foo, FooDeleter>;
8. FooWrapper foo(createObject(32, 3.14));

```

### 5.2.6.2 Examples

- There are two different resource handlers: One is window api return value, The other is FILE\* in C language
- Example for FILE\* is below:

```

1. // For customized deleter, you need callable object
2. unique_ptr<std::FILE, decltype(&std::fclose)>
3.     fp(std::fopen("demo.txt", "r"), &std::fclose);
4.
5. if(fp) // fopen could failed; in which case fp holds a null pointer

```

- We can make FILE\* example better, use unique\_ptr source semantic.

```

1. struct FILEDeleter {
2.     void operator()(FILE *pFile){
3.         if (pFile)
4.             fclose(pFile);
5.     }
6. };
7.
8. using FILE_unique_ptr = unique_ptr<FILE, FILEDeleter>;
9.
10. FILE_unique_ptr make_fopen(const char* fname, const char* mode){
11.     FILE *fileHandle= nullptr;
12.     auto err = fopen_s(&fileHandle , fname, mode);

```

```

13. if (err != 0){
14.     // print info, handle error if needed...
15.     return nullptr;
16. }
17. return FILE_unique_ptr(fileHandle);
18. }
19.
20. //usage in your code!
21. FILE_unique_ptr pInputFilePtr = make_fopen("test.txt", "rb");
22. if (!pInputFilePtr)
23.     return false;

```

- Just the same idea, if you want to use shared\_ptr wrap FILE\*, see below example.

```

1. using FILE_shared_ptr = std::shared_ptr<FILE>;
2.
3. FILE_shared_ptr make_fopen_shared(const char* fname, const char* mode) {
4.     FILE *fileHandle = nullptr;
5.     auto err = fopen_s(&fileHandle, fname, mode);
6.     if (err != 0){
7.         // handle error if needed
8.         return nullptr;
9.     }
10.
11.    return FILE_shared_ptr(fileHandle, FILEDelete());
12.    //Pay attention!, use FILEDelete(), but unique_ptr use FILEDelete
13. }

```

- Just the same idea, You should know **HANDLE** in windows is just void\* type pointer, so you can use this way to deal with windows handle.

```

1. struct HANDLEDelete{
2.     void operator()(HANDLE handle) const{
3.         if (handle != INVALID_HANDLE_VALUE)
4.             CloseHandle(handle);
5.     }
6. };
7.
8. using HANDLE_unique_ptr = unique_ptr<void, HANDLEDelete>;
9.
10. HANDLE_unique_ptr make_HANDLE_unique_ptr(HANDLE handle){
11.     if (handle == INVALID_HANDLE_VALUE || handle == nullptr){
12.         // handle error...
13.         return nullptr;
14.     }
15.     return HANDLE_unique_ptr(handle);
16. }
17.
18. auto hInputFile = make_HANDLE_unique_ptr(
19.             CreateFile(strIn, GENERIC_READ, ...));
20. if (!hInputFile)
21.     return false;

```

- In unique\_ptr type. std::remove\_reference<Deleter>::type::pointer if that type exists, otherwise T\*. Must satisfy NullablePointer.
- For unique\_ptr, if you can deduct pointer type from deleter, unique\_ptr will use it directly, so you just know SC\_HANDLE is pointer, but you don't know exact type, you can write just like below:

```

1. struct SvcHandleDeleter{
2.     typedef SC_HANDLE pointer;
3.     SvcHandleDeleter() {};
4.
5.     template<class Other> SvcHandleDeleter(const Other&) {};
6.
7.     void operator()(pointer h) const {
8.         CloseServiceHandle(h);
9.     }
10. };
11.
12. typedef std::unique_ptr<SC_HANDLE, SvcHandleDeleter> unique_sch;
13.
14. unique_sch scm(::OpenSCManagerA(0, 0, SC_MANAGER_ALL_ACCESS));

```

- For shared pointer, type-erasure makes it impossible with the current interface to achieve exactly what type you want. So you can use a dumb way, Just like a pointer to pointer. If you don't know what is behind SC\_HANDLE. If you know it's type is void, you can use void directly, it will save you a lot of trouble.

```

1. std::shared_ptr<SC_HANDLE> sp(new SC_HANDLE(
2.     ::OpenSCManagerA(0, 0, SC_MANAGER_ALL_ACCESS)),
3.     [](SC_HANDLE* p){ ::CloseServiceHandle(*p); delete p; });

```

- Use shared\_ptr to wrap a handle, A good introduction is "Making a HANDLE RAI-compliant using shared\_ptr with a custom deleter" in stackoverflow
- Sometimes, I want to keep file alive, because foo and bar will use them. You can't use RAI auto object. And If you use raw pointer, It's difficult to trace and delete it. shared\_ptr is the best options right now. you don't needs to worry about deleting file - once both foo and bar have finished and no longer have any references to file (probably due to foo and bar being destroyed), file will automatically be deleted.

- For C language, you can use FILE\* and input the

```

1. void setLog(const Foo & foo, const Bar & bar) {
2.     //File file("/path/to/file", File::append); //1) RAI auto obj
3.     //FILE* file = new File("/path/to/file", File::append); //2) raw new
4.     shared_ptr<File> file =
5.         shared_ptr<File>(new File("/path/to/file")); FILEDeleter() //3) best
6.
7.     foo.setLogFile(file);
8.     bar.setLogFile(file);
9. }

```

- for fstream, it's different with FILE\*. It's based on value semantic. So common usage just fstream f1; f1.open and f1.close. If you want to manage it's time smartly, you can produce a fstream\* pointer and wrapped by shared\_ptr. **Pay attention, you don't need input customized deleter, because fstream is RAI object, it does close automatically at the end of the scope**
- fstream is value semantic, so you can use shared\_ptr directly. just like to deal with other value semantic variable, such as int and class.

```

1. shared_ptr<fstream> fp {new fstream(name, mode) };
2. if (!*fp)
3.     throw No_file {};
4.
5. foo.setLogFile(fp);
6. bar.setLogFile(fp);

```

## 5.3 smart pointer and polymorphism

### 5.3.1 pointer\_cast function

- static\_pointer\_cast

```

1. struct BaseClass {};
2.
3. struct DerivedClass : BaseClass {
4.     void f() const {
5.         std::cout << "Sample word!\n";
6.     }
7. };
8.
9. int main() {
10.     std::shared_ptr<BaseClass> ptr_to_base(make_shared<DerivedClass>());
11.     std::static_pointer_cast<DerivedClass>(ptr_to_base)->f();
12.     std::dynamic_pointer_cast<DerivedClass>(ptr_to_base)->f();
13.
14.     static_cast<DerivedClass*>(ptr_to_base.get())->f();
15. }

```

### 5.3.2 Usage

- In this section, you need to know four things:
  1. share\_ptr child to base;
  2. share\_ptr base to chile (down cast);
  3. unique\_ptr child to base;
  4. unique\_ptr base to chile (down cast);
- **Basic idea: smart pointer base and smart pointer are not covariant at all. You can think that just like vector<Base> and vector<derived>.**
- For shared\_ptr, it support derived to base shared\_ptr copy or assignment. Behind the hood, you need to know the template member function. Detail can be found in effective C++ item 45 "Use member function templates to accept "all compatible types". Why you can? because it get raw pointer and wrapped it again and compiler allow it.
- For shared\_ptr base, you can't assign it to the shared\_ptr child, so you must use static\_pointer\_cast and dynamic\_pointer\_cast
  1. they only support shared\_ptr.
  2. They use shared\_ptr aliasing ctor, get raw pointer, use static or dynamic cast, then build derived shared\_ptr too. Detail can be found:  
"std::static\_pointer\_cast vs static\_cast<std::shared\_ptr<A>>"

- For shared\_ptr base reference, You have to use const. Why do we need const?

1. from pd1 build temporary base shared\_ptr pointer.
2. static\_pointer\_cast also return a temporary base shared\_ptr.

```

1. void doSomething(const std::shared_ptr<Base>& ptr) {
2. // you must use const
3.     std::cout << ptr.use_count() << std::endl; // cout 2
4. }
5.
6. int main() {
7.     std::shared_ptr<Derived1> pd1 = std::make_shared<Derived1>();
8.     //doSomething(pd1); Will not compile here.
9.     doSomething(shared_ptr<Base> temp(pd1));
10.    doSomething(static_pointer_cast<Base>(pd1));
11. }
```

- Most of time, use smart pointer reference just for observe. at this time, you can use raw pointer or raw reference directly.
- From above, you can see static\_pointer\_cast mainly used to down cast shared\_ptr. From derived to base, you can use const reference directly.
- static cast also can used for check down cast, it will check at compile time. not like dynamic cast, it will check at run time and also require class has at lease one virtual function.
- For unique\_ptr, You can use move from derived pointer to base pointer directly.

```

1. void doSomething(const std::unique_ptr<Base> ptr) {
2.     ptr->run();
3. }
4.
5. int main() {
6.     std::unique_ptr<Derived1> pd1 = std::make_unique<Derived1>();
7.     doSomething(std::move(pd1));
8. }
```

- For down cast unique\_ptr, There are no counter part of static\_pointer\_cast. So only way you can do is use release and wrap it again.

```

1. template<typename TO, typename FROM>
2. unique_ptr<TO> static_unique_pointer_cast(unique_ptr<FROM>&& old) {
3.     return unique_ptr<TO>{static_cast<TO*>(old.release())};
4.     //conversion: unique_ptr<FROM>->FROM->TO*>->unique_ptr<TO>
5. }
6.
7. unique_ptr<Base> foo = fooFactory();
8. unique_ptr<Derived> foo2 =
9.     static_unique_pointer_cast<Derived>(std::move(foo));
```

- For const reference, there are some methods:

```

1. void f(const unique_ptr<Base>& base)
2. unique_ptr<Derived> derived = unique_ptr<Derived>(new Derived);
3. f(derived); //this fail;
4.
5. f(std::move(derived)); //method 1 work, Why?
6.     //because int i = 3; const float& dr = i; compile OK
```

```

7. void f( std :: unique_ptr<Derived> const&); //method 2
8.
9.
10. std :: unique_ptr<base> derived = std :: make_unique<Derived>(); //method 3
11. std :: unique_ptr<base> derived(new Derived);
12.
13. void f(Base & b); //method 4
14. f(*derived);

```

## 5.4 smart pointer and class

### 5.4.1 RAII

- Whenever you deal with a resource that needs paired acquire/release function call, encapsulate that resource in an object. Such as: fopen/fclose, lock/unlock, and new/delete.
- When implementing RAII, be conscious of copy construction and assignment. the compiler-generated version probably won't be correct. If it's not copyable, use =delete , if it's copyable, duplicate the resource. You also can use smart\_pointer in this scenario too.
- The basic idea of RAII is to represent a resource by a local object, so that the local object's destructor will release the resource. That is to say: To prevent resource leaks, use RAII objects that acquire resources in their constructors and release them in their destructors.

```

1. //C version ,
2. File* fp = fopen("/path/to/file");
3. // throw exception here , then resource leaking
4. fclose(fp);
5.
6. //Java version
7. try {
8.     File file = new File("/path/to/file");
9.     // throw exception here , go to finally .
10. } finally {
11.     file.close();
12. }
13.
14. //c++ version
15. fun{
16.     fstream if("path/to/file")
17.     if.getline
18.     // you don't need to if.close().
19. }
20.
21. //c++ smart version .
22. std::unique_ptr<FILE, // the wrapped raw pointer type: FILE*
23. int(*)(FILE*)> // the custom deleter type: fclose() prototype
24. myFile( fopen("myfile", "rb") , // (FILE*) is returned by fopen()
25. fclose ); // the deleter function: fclose()

```

- Another good example RAII is unique\_ptr. The idea of smart pointer is putting \*p into a local pointer-like object, then when it goes out of scope or unwind-stack when exception is thrown, It will call destructor, then delete p.
- We should consider resource generically, pointer \*p pointed to a new object is resource, A handle to a file is a resource to. **We wrap handle to a file into ifstream, and wrap pointer \*p into smart\_pointer**

- Just like return value, Exception will skip all the statement below the throw, In C++, It doesn't support finally statement sometimes. At this time, we need to use RAII.

```

1. Int *p = new int;
2. string a //a is ok, a will be destructed properly
3. //due to the C++ unwind stack.
4. throw exception.
5. delete p; // this will not run.

```

- For this problem, you should use smart pointer to declare a auto object. If you use string object, It's ok. So in previous example, you can use smart pointer, it will help you to avoid memory leakage problem.

```

1. unique_ptr<int> aupr (new int(100));
2. string a //a is ok, a will be destructed properly
3. //due to the C++ unwind stack.
4. throw exception.
5. // both a and aupr will call their own destructor function.

```

- In you ctor, If you use new and new failed and throw a exception, the destructor will not be called. You can use auto\_ptr as member data and use init list to initialzie it. see more effective C++ exception chapter.
- Don't use C FILE\* and char [] as string. Use iofile class and string object, because they are exception safe
  1. Any time when you use new, consider if there are c++ container or object.
  2. If not, use smart\_pointer.

- Another example is when you make program based on Win API.

```

1. class module {
2. public:
3. explicit module(std::wstring const& name)
4. : handle{ ::LoadLibrary(name.c_str()) } {}
5.
6. ~module{
7. ::FreeLibrary();
8. }
9. private:
10. HMODULE handle;
11. };

```

- There are three RAII implementation instances in your practical programming:

1. Use auto member; You have to keep m\_str and vc are RAII. In this way, you don't need to build dtor manually.

```

1. class RAII {
2. private:
3.     string m_str;
4.     vector<int> vc;
5. };

```

2. Use pointer and handle; In this way, You have to use pointer, Maybe you need some customized action in runtime , **Use handle is only method to use this resource** or any other reason. And this time, you have to write your own dtor.

```

1. class RAII {
2. private:
3.     string* m_str;
4.     vector<int*> vc;
5. };

```

3. Use smart point wrap pointer and handle; When you wrap handle, you can custom this delete behavior. See source code below:

```

1. class RAII {
2. private:
3.     unique_ptr<string> m_str;
4.     vector<unique_ptr<int>> vc;
5. };

```

```

1. class module {
2. public:
3.     explicit module(std::wstring const& name)
4.         : handle{ ::LoadLibrary(name.c_str()) } {}
5. private:
6.     using module_handle=std::unique_ptr<void, decltype(&::FreeLibrary)>;
7.     module_handle handle;
8. };

```

- Another question is ownership of resource:

1. For auto member resource: 1) **Same life duration(RAII)**, 2) **exclusive ownership to a single obj, but it's copyable(A a1 = a2)**. 3) **move with efficiency(A a1 = A()** . If auto member has its own copy and move special function, You don't need to write any special function in your class. You follow the "Rule of Zero".
2. For raw pointer and handle: 1) **default copy ctor will cause two pointer or handle refer the same resource**, It's absolutely BAD SMELL of code 2) So you have to follow "Rule of five" to build your special member function. 3) After you build five special member function, you get **RAII and exclusive ownership to a single object, and copyable and efficient move**

```

1. Class RawPointer{
2.     .....
3.     RawPointer(const RawPointer& rhs){
4.         pRes = new Resource( *(rhs.pRes));
5.     }
6.
7.     RawPointer(RawPointer&& rhs){
8.         pRes = rhs.pRes;
9.         rhs.pRes = nullptr;
10.    }
11. private:
12.     Resource* pRes;
13. };

```

3. For **uniqu\_ptr**; 1) **Same life(RAII)** 2) **exclusive ownership but not copyable** 3) **uniqu\_ptr support move operation.** You still follow "rule of zero"
4. Even with **uniqu\_ptr** member, If you follow "rule of zero", that is to say that you don't provide any customized special member function, then the class is not copyable. But if you build copy ctor by yourself, get raw pointer from origin side, and build a new **uniqu\_ptr**

member from origin side's raw pointer, you can implement copyable, and code smell better than raw pointer with "Rule of Five". So in this way, **It's not recommended to use raw pointer in RAII and ownership context.**

5. For shared\_ptr; 1) **Not a RAII** 2) **shared ownership**, 3) **copyable and moveable**. When you move a shared\_ptr, origin one is set to nullptr and ref count doesn't increase. You still follow "rule of zero".

- **Conclusion, If you consider RAII and ownership at the same time, thing will become complex** so I would like to give you some examples to illustrate them.

1. Prefer to use auto member for most of time! It follows "Rule of Five" and supports copyable and movable. Such as std::string
2. For special demand, for example Car class, people can **custom** its engine, and buy **two** at the same time. In this context, your car class should use raw pointer, 1) auto member doesn't support custom 2) unique\_ptr doesn't support copyable. And you have to follow "Rule of Five"

```

1. class Car{
2.     //follow "Rule of Five"
3.     Engine *pEn;
4.     ~Car() {delete pEn} // assure RAII
5. }
```

3. For special context, it doesn't support object copy: for example 1) Person class in semantic; 2) other performance consideration, Class BigInt [30000]; 3) Other implementation constraint, such as iostream class. Under such context, you can use unique\_ptr to manage the resource and implement uncopiable.

```

1. class Person{
2.     unique_ptr<Resource> pRes;
3. }
```

4. For special context, shared resource, you can use shared\_ptr. You still can follow "Rule of zero" and resource will be deleted when ref count is 0.

```

1. class Student{
2.     shared_ptr<SchoolBus> pBus
3. }
```

5. To know the semantic of two smart pointers. Don't use them just replace raw pointer.

## 5.5 smart pointer Summary

### 5.5.1 principle

- Three policy of smart pointer usage:
  1. **Owership policy: use smart pointer.**
  2. **Observer Policy : use raw pointer, reference or weak\_ptr**
  3. Nullity Policy: Not allow nullptr, prefer to use reference. **prefer reference than pointer**
- Smart pointer general guides:

1. **There are three places you can use smart pointer: braced scope(function), class member and container item.** In these three places, you can have exclusive ownership, shared ownership or observer.
2. If the program uses more than one pointer to an object, shared\_ptr is your choice. Such as you have two objects that contain pointers to the same third object. Or you may have an STL container of pointers.
3. You must ensure that there is only one manager object for each managed object. You do this by writing your code so that when an object is first created, it is immediately given to a shared\_ptr to manage, **and any other shared\_ptrs or weak\_ptrs that are needed to point to that object are all directly or indirectly copied or assigned from that first shared\_ptr.** The customary way to ensure this is to write the new object expression as the argument for a shared\_ptr constructor, or use the make\_shared function template described below.
4. The exception to this immediate assignment rule is things like factory methods that return a plain pointer to the object they create. in this case however, the callee still should generally be immediately assigning this returned object to a shared\_ptr or unique\_ptr. Methods should return a plain-pointers when it is up to the caller to handle ownership of the object(exclusive ownership or shared ownership).
5. Besides above method, objects can also be produced by factory pattern or factory function. Then in three kinds of place to mange it's lifetime.
6. Methods can take plain-pointers as their arguments for just observe it. Or use smart pointer to transfer or get ownership.

```

1. ObserveFun(Foo* p);
2. ObserveFun(smart_pointer.get() );
3. ObserveFun(unique_ptr<Foo> &p);
4. //Not use very often, can be used as fun_obj
5. // in a container of unique_ptr.
6.
7. UniqueFun(unique_ptr<Foo> p);
8. UniqueFun(make_unique_ptr<Foo>(new Foo() )); //get ownership
9. UniqueFun(move(other_unique_ptr) ) //transfer ownership
10.
11. SharedFun(shared_ptr<Foo> p);

```

7. If you want to get the full benefit of smart pointers, your code should avoid using raw pointers to refer to the same objects; otherwise it is too easy to have problems with **dangling pointers and double deletions**. In particular, smart pointers have a get() function that returns the pointer member variable as a built-in pointer value. This function is rarely needed. As much as possible, leave the built-in pointers inside the smart pointers and use only the smart pointers.

- When you want to use raw pointer in STL container, There are two things you need to consider:
  1. Be wary of remove-like algorithms on containers of pointers. "effective STL item 33"
  2. When using containers of newed pointers, remember to delete the pointers before the container is destroyed. "effective STL item 7". **All these two problems can be resolved by using smart pointer.**
- **Observer pointer,observing pointers are pointers which do not keep the pointed object alive** raw pointers are bad when used for performing manual memory management, i.e. new and delete. When used purely as a means to achieve reference semantics and pass around

non-owning, observing pointers, there is nothing intrinsically dangerous in raw pointers. Just not to dereference a dangling pointer

```

1. observe(subject * s1)
2. // subject is a class observer is function .
3. //just use raw pointer subject , use weak_ptr or raw pointer .
4. //observer doesn't have owner policy and life time policy with subject

```

### 5.5.2 Function interface

- **Function example 1: Inside a function**

1. If you don't want to create dynamically or large array, don't use new. Just use local auto object. Even you need large array, consider STL container first.
2. When you have to use new, and this function has **Ownership of pointer, means that they have the same life time**, use unique\_ptr. In this way, you don't need delete and it's exception safe.

```

1. fun () {
2.     Foo fo();
3.     // it make obj directly ,
4.     //when you don't need dynamic .
5.
6.     if (input == "Foo")
7.         uniqu_ptr<Foo> up(new Foo() );
8.         //When you need new, use uniqu_ptr
9. }

```

- **Function example 2 : argument of a function.**

1. As a parameter, pass it to a function. if you don't want to create dynamically or large array, don't use new. Just use reference.
2. Prefer passing parameters by \* or &.
3. Passing unique\_ptr by reference is for in/out unique\_ptr parameters. when the function is supposed to actually accept an existing unique\_ptr and potentially modify it to refer to a different object.
4. In this sense, const unique\_ptr & MUST be observer. So A better way is to sue raw pointer as observer directly.
5. **If you want to transfer ownership to callee from caller, use uniqu\_ptr, and use move.** Passing unique\_ptr by value means "sink."
6. **If you want to shared ownership to callee from caller, use shared\_ptr**
7. Use a non-const shared\_ptr& parameter only to modify the shared\_ptr. Use a const shared\_ptr& as a parameter only if you're not sure whether or not you'll take a copy and share ownership; otherwise use widget\* instead (or if not nullable, a widget&).
8. When you assign unique\_ptr to shared\_ptr, use move.

```

1. Foo *fo = new Foo(); //bad smell here .
2. fun(Foo * p);
3. delete fo;
4.
5. fun(Foo &p); //use reference to improve efficiency
6.

```

```

7. unique_ptr<Foo> up(new Foo() );
8. fun(unique_ptr<Foo>& up); //use reference here
9. //to avoid copy, unique_ptr can't copy
10.
11. fun(unique_ptr<Foo> down); //prototype
12. fun(std::move(up));
13.
14. std::unique_ptr<std::string> unique = std::make_unique<std::string>("test");
15. std::shared_ptr<std::string> shared = std::move(unique);

```

- **Function example3: return from function.**

1. As a function return value. if you don't want to create dynamically or large array, don't use new. **Don't use reference refer to a local object created inside of fun.**
2. If you have to use New, and you want to transfer Ownership from callee to caller. return unique\_ptr.
3. If there is no clear single owner, store and return shared\_ptr.(in this example, caller of fun() is single owner, so use unique\_ptr)

```

1. Foo* fun() { //old c style .
2.     return new Foo();
3. }
4.
5. unique_ptr<Foo> fun() { // this is better .
6.     .....
7.     return unique_ptr<Foo>(new Foo());
8. }

```

4. Below use shared\_ptr, because Server is public used, and no single owner.

```

1. shared_ptr<Server> buildNewServer() { // this is better .
2.     return shared_ptr<Server>(new Server());
3. }
4.
5. shared_ptr<Server> serverForClass1 = buildNewServer();
6. shared_ptr<Server> serverForClass2 = serverForClass1;

```

- About smart pointer and function interface. There is a good article. "GotW #91 Solution: Smart Pointer Parameters"

# Chapter 6

## reference and rvalue reference

### 6.1 reference basic

- Reference has two characteristics, no null, no change. In other words, top level of const of reference is default. You can't change it. That is why you have to initialize it.

```
1. int x = 12;
2. int* const p = &x; // you have to init it.
3. p = &y; //you can't point to different object.
4. //reference is a kind of top level const pointer.
```

1. No change, a reference always refers to the object with which it's initialized firstly.

```
1. string &rs = s1;
2. rs = s2; // s1's value is modified, rs still refer to s1.
```

2. Reference is different with pointer. Reference has not wild pointer problem. So a reference doesn't need to be tested if it's null reference.

```
1. if(pointer) // don't need to do it for a reference.
2. cout<< *pointer<<endl;
```

3. If you have a reference to a local variable inside a function, when the function finishes, it still have dangling reference problem.

- you can have reference to pointer, but pointer to reference is illegal. In grammar level, reference to reference has different means, it's called rvalue reference. This doesn't exist. As stated earlier, a reference is merely an alias to another object. You can't "point" to a reference, because it isn't an object in itself but merely another name for a real object. Of course, you can have a pointer to the object that a reference is referring to. But now we are back in vanilla pointer territory.

```
1. int *p;
2. int *& rp = p; // reference to pointer is OK
3. int &* pr //not legal in c++
4. //error: cannot declare pointer to 'int&'
5. int && rv //rvalue reference.
```

- non-const reference can't refer to const object.

```
1. const int i = 12;
2. int& j = i;
```

- A `const` object reference passed into a function, if you want to return it, it must be `const` too. Usually, It has no any practical meanings when you do in this way.

```

1. //method 1
2. const int &fun(const int& i){
3.     return i;
4. } //Why we need this? Just a syntactic demo, no semantic meaning
5.
6. //method 2
7. int &fun(const int& i){
8.     return const_cast<int&>(i);
9. }
```

- Only `const` reference can bound to temporary and prolong temporary variable life. Temporary is not lvalue, and only lvalue can bound to reference to `non-const`. Only stack-base `const` reference can work in this way. If a `const` reference is class member, it can't work. See two examples below:

```

1. //case 1:
2. Foo f(){
3.     return obj;
4. }
5. const Foo & rf = f();
6.
7. //case 2:
8. class Foo(int i); // ctor
9. f(const Foo & crf); //function declaration
10. //here, must const & in f argument.
11. f(1); //1 -->"temp obj build by ctor" -->crf
12. //if you skip const, compiler will report error:
```

- `const` reference bound to rvalue used in copy ctor widely. See example below: But in C++11, For a class with a lot allocated resource, please use move copy ctor which explained in the next section:

```

1. class Foo{
2.     Foo(const Foo & foo);
3. }
4.
5. Foo f3 = f1+f2;
6. // because a temp Foo is produced first from f1+f2
7. // without const in copy ctor, compiler will bark
```

## 6.2 lvalue, rvalue and xvalue

- In this section, there are four important points
  1. Academic definition of xvalue.
  2. Give some practical expressions which are xvalue.
  3. Relationship between xvalue and rvalue reference.
  4. Why do we need a new value type-xvalue?

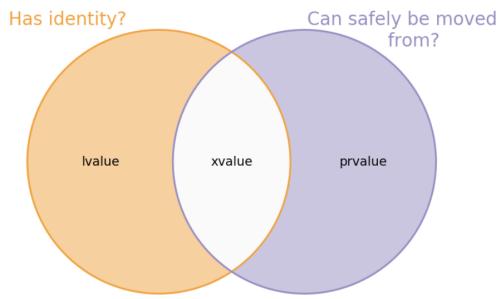


Figure 6.1:

### 6.2.1 Definition

- Each C++ expression (an operator with its operands, a literal, a variable name, etc.) is characterized by two independent properties: **a type and a value category**. Each expression is certain kind of type, such as int type, reference type or rvalue reference type. Each expression belongs to exactly one of the three primary value categories: **lvalue, prvalue, and xvalue**;
- lvalue is not defined "can be put on the left side of =". Because `const int a` is also a lvalue, and you can't put a on the left side of = . It has three characteristics:
  1. **lvalue has Identity**, (can get address by & operator),
  2. **lvalue can be persist beyond the expression**.
  3. **lvalue can't be move(stolen)**, because you need to keep original one intact.
- On the contrary, prvalue has no **identity**, and will not **persist** beyond expression and can be **move(stolen)**
- Why prvalue doesn't persist? because rvalues in C++ stored differently. It's freedom for the compiler to improve performances of your code. A more concrete way to understand this is to remember that a value can be stored in a register of your CPU and never actually be in your memory which more or less means that the value has no address. I won't bet everything i have on it but this is probably one of the main reasons why "we cannot get an address of an rvalue".
- In a more general way since an rvalue is semantically temporary it is more likely to be put in temporary places or optimised in a way where it cannot easily be mapped to an address and even if it can that would be counter productive in terms of performance.
- First, we can use **persist and identity** to classify value into lvalue and rvalue, then c++11 introduce **rvalue reference**. rref can persist and move at the same time. So we divide lvalue into lvalue and xvalue. and give them new name. lvalue+xvalue = glvalue(persist) and xvalue + prvalue = rvalue(move).
- **Any value must be one of three, lvalue, xvalue, or prvalue**. These three categories are complementary. lvalue pay attention to identity and persist, rvalue=(xvalue + prvalue) shout "I can be moved" loudly. So we call **&& rvalue reference**, don't call **xvalue reference**.
- To know this conception can help you to understand decltype.

- Summary table:

	persist(Identity)	move
lvalue	Yes	No
pvalue	No	Yes
xvalue	Yes	Yes

- An example shows that xvalue has identity and persist, and will persist beyond expression and can be move(stolen). it usually near the end of its lifetime (so that its resources may be moved).

```

1. string&& xvalue_fun();
2.
3. xvalue_fun(); // after you call this fun, value still exist.
4. xvalue_fun() = "hello" // you can modify because it persist
5. &xvalue_fun(); // you can get address because it has identity.
6. string a = xvalue_fun(); // call move ctor, can move

```

- A method to judge if a var is lvalue.

```

1. template <typename T>
2. constexpr bool is_lvalue(T&&)
3. {
4.     return std::is_lvalue_reference<T>{};
5.
6. std::string a("Hello");
7. is_lvalue(std::string()); // false
8. is_lvalue(a); // true
9. // in the case you pass a std::string lvalue then
10. // T will deduce to std::string& or const std::string&,
11. // for rvalues it will deduce to std::string

```

### • How to determine programmatic if an expression is rvalue or lvalue

```

1. if (std::is_lvalue_reference<decltype(var)>::value) {
2.     // var was initialised with an lvalue expression
3. } else if (std::is_rvalue_reference<decltype(var)>::value) {
4.     // var was initialised with an rvalue expression
5. }
6.
7. string&& xvalue_fun();
8. std::is_lvalue_reference<decltype(xvalue_fun())>::value
9. // return true

```

## 6.2.2 Example of xvalue

- Remember that any expression that evaluates to an lvalue reference (e.g., a function call, an overloaded assignment operator, etc.) is **an lvalue**. Any expression that **returns an object by value is an rvalue**.

```

1. string& lvalue_fun();
2. lvalue_fun(); // after you call this fun, lvalue still exist.
3. lvalue_fun() = "hello" // you can modify because it persist
4. &lvalue_fun(); // you can get address because it has identity.
5. string a = lvalue_fun() // call copy ctor, can't move
6.
7. string pvalue_fun();
8. pvalue_fun(); // after call this fun, value disappear.
9. pvalue_fun() = "hello" //NOT modify because it doesn't persist
10. &pvalue_fun(); // NOT get address because it no identity.

```

```
11. string a = pvalue_fun() // call move ctor, can move
```

- An xvalue is the result of certain kinds of expressions involving rvalue references. **The result of calling a function whose return type is an rvalue reference is an xvalue.**
- textbfff return lvalue reference is lvalue, fun return value is pvalue, and fun return rvalue reference is xvalue.
- In definition, xvalue is just value with Identity and can be movable. **In real life, it's return value of std::move() or static\_cast<A&&>.** A better introduction can be seen: " C++11 Tutorial: Explaining the Ever-Elusive Lvalues and Rvalues"
- More xvalue examples.

```
1. struct A {
2.     int m;
3. };
4.
5. A&& operator+(A, A); // a+a is xvalue
6. A&& f(); // f() is xvalue
7. f().m // f().m is also xvalue
8. A a;
9. A&& ar = static_cast<A&&>(a);
10. // static_cast<A&&>(a) is xvalue, but ar is lvalue
```

- After C++11, we added two new xvalues.
  - 1) a[n], the built-in subscript expression, where one operand is an array rvalue;
  - 2) a.m, the member of object expression, where a is an rvalue and m is a non-static data member of non-reference

### 6.2.3 xvalue and rvalue reference

- xvalue is defined based on rvalue reference. But you can't think that a rvalue reference is a xvalue.
- A named rvalue reference is lvalue, and unnamed rvalue reference is xvalue;

```
1. void foo(int&& t) {
2.     // t is initialized with an rvalue expression
3.     // but is actually an lvalue expression itself
4. }
5.
6. std::string a; // a, b, c are all lvalue.
7. std::string& b;
8. std::string&& c;
```

- goo is declared as an rvalue reference and does not have a name, and is therefore an xvalue.

```
1. X&& goo();
2. X x = goo();
3. // calls X(X&& rhs) because the thing on
4. // the right hand side has no name
5.
6. std::move(x) // get rid of x name,
7. // then return rvalue reference
```

- Why is there such confusion? Here are the circumstances under which it is safe to move something:
  - 1) When it's a temporary or sub-object thereof. (prvalue)
  - 2) When the user has explicitly said to move it.

```

1. SomeType &&Func() { ... }
2.
3. SomeType &&val = Func();
4. SomeType otherVal{ val}; // Do you really want to move
5. ...
6. cout<<val; //what happen if you have forget you have move.
7. //so here, val is lvalue. because it's a named rvalue reference.

```

- Another deep trap happen when you implement move ctor in derived class

```

1. Derived(Derived&& rhs) : Base(rhs) // wrong: rhs is an lvalue
2. {
3.   // Derived-specific stuff
4. }
5.
6. Derived(Derived&& rhs) : Base(std::move(rhs)) {
7.   // good, calls Base(Base&& rhs)
8.   // Derived-specific stuff
9. }

```

#### 6.2.4 Why need xvalue

- given a type T, you can have lvalues of type T as well as rvalues of type T. It's especially important to remember this when dealing with a parameter of rvalue reference type, because the parameter itself is an lvalue.

```

1. int && i = 2; //i expression is rvalue referene type lvalue
2. int && fun()
3. fun() //fun() expression is vvalue reference type rvalue.

```

- before C++11, we have used value category to overload function. plain lvalue reference can only bind to lvalue, can't bind to rvalue. So we use **const lvalue reference**. Const lvalue reference can bind to both lvalue and rvalue, but you can't change it. We have used this technology in copy ctor very common before C++11. Because const reference can bind both lvalue and rvalue, so we can't distinguish when to move, when to copy, that is the problem of efficiency.

- After C++11, in order to improve efficiency when deal with rvalue, we introduced rvalue reference. rvalue reference can only bind to rvalue, (rvalue is composed of prvalue+ xvalue). **That is how function overload kick in?**

```

1. int a = 7; // a is lvalue, It has Identity
2. int && r1 = 13, //r1, r2 and r3 are lvalue
3. int && r2= x+y; //x+y is prvalue
4. int && r3 = sqrt(2.0); //sqrt(2.0) is prvalue
5. int && r4 = a //error, rvalue reference can't refer to lvalue.
6.
7. const int & lv = x+y; //lv is lvalue
8. int & r = x+y; //error, lvalue reference can not bind with an rvalue.

```

type	value category
lvalue reference	lvalue(non-const)
rvalue reference	rvalue(non-const)
const lvalue reference	lvalue or rvalue(const or non-const)

- C++11 allows you to use move semantics not just on rvalues, but, at your discretion, on lvalues as well. A good example is swap function, here if you can move lvalue, the efficiency will be better.

```

1. template<class T>
2. void swap(T& a, T& b) {
3.     T tmp(std::move(a));
4.     a = std::move(b);
5.     b = std::move(tmp);
6. }
```

- So for std::move() function, return value can be value(rvalue), but it can't support dynamic binding in C++. return value can be reference(lvalue), but it can't be used move context. So we have to use rvalue reference as return type.
- for this kind of rvalue reference, we can put it on the left side of assignment, so it's not prvalue. At the same time, you can move it, so it's not lvalue. **In this way, we have to introduce a new value type—xvalue.**

```

1. string str = "hello"
2. std::move(str)[0] = 'z';
3. cout << str << endl // print zello here.
4. // so std::move() return a xvalue, it can persist and move.
```

- Summary:

1. We have rvalue reference to bind temporary value to move it.
2. We want to move a lvalue, such as swap, so we have std::move() function.
3. std::move() return neither lvalue nor prvalue, so we have to define a new type of value—xvalue.

## 6.3 rvalue reference and move scenario

### 6.3.1 basic of rvalue reference

- rvalues denote temporaries or objects that want to look like a temporary. What is so particular about temporaries, is the fact that they will be used in a very limited way: their value will be read once, and they will be destroyed. This is a very useful observation in implementing "move semantics."
- How to understand move semantics? Move is not really move a big chunk of data, It just move index of data. just like you move file in the hard disk.
- rvalue references will implicitly bind to rvalues and to temporaries that are the result of an implicit conversion. i.e. float f = 0f; int&& i = f; is well formed because float is implicitly convertible to int; the reference would be to a temporary that is the result of the conversion.
- Rvalue references can be used to extend the lifetimes of temporary objects (note, lvalue references to const can extend the lifetimes of temporary objects too, but they are not modifiable through them):

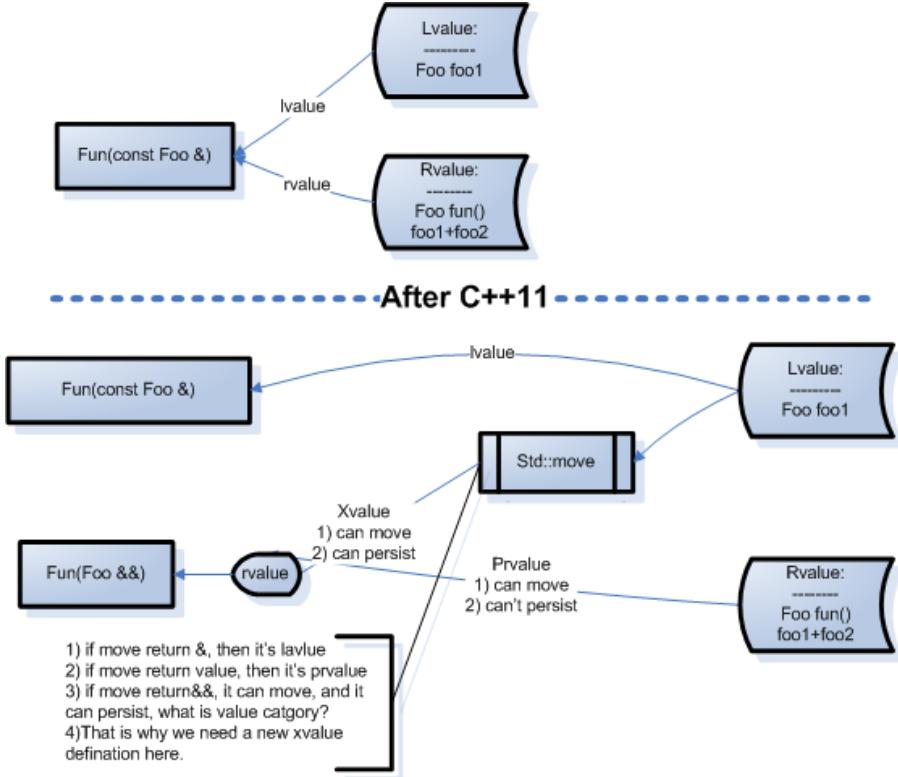


Figure 6.2:

```

1. std::string s1 = "Test";
2. // std::string&& r1 = s1;
3. // error: rvalue ref can't bind to lvalue
4.
5. const std::string& r2 = s1 + s1;
6. // okay: lvalue reference to const extends lifetime
7. // r2 += "Test";
8. // error: can't modify through reference to const
9.
10. std::string&& r3 = s1 + s1;
11. // okay: rvalue reference extends lifetime
12. r3 += "Test";
13. // okay: can modify through reference to non-const

```

### 6.3.2 move ctor and assignment

- Typically, if your class allocate a lot of allocated resource (new, or manually manage some system resources). You should implement two copy ctor. One of them uses rvalue references, which allow a function to branch at compile time (via overload resolution) on the condition "Am I being called on an lvalue or an rvalue?" A few explanations are below:

1. Move ctor will not move resource automatically, you need to coding it by your self.
2. You can't move in normal ctor. because, It must keep origin obj intact. You can steal when `Foo(f1+f2)`, but when you used `Foo(f1)`. It will destroy f1.
3. Without move ctor, normal copy ctor will treat `Foo f = f1+f2` and `Foo f = f1` the same way.

4. With move copy ctor, normal copy ctor deal with Foo f = f1, and move copy ctor deal with Foo f = f1+f2, for f1+f2, you can steal resource, because nobody need to use f1+f2 later any more.
5. In move ctor, always set rhs.ptr = nullptr;
6. No const qualifier in move ctor and move assignment

```

1. Foo::Foo(const Foo & foo) {
2.     while( ptr++ )
3.         ptr[ i ] = foo.ptr[ i ] // expensive copy
4.     }
5.
6. Foo::Foo(Foo && rhs){ //no const here
7.     ptr = other.ptr; // efficient move(steal)
8.     rhs.ptr = nullptr;
9. }
10.
11. Foo& Foo::operator=(Foo&& rhs) {
12.     delete[] ptr;
13.     ptr = other.ptr; // efficient move(steal)
14.     rhs.ptr = nullptr;
15.     return *this;
16. }
```

- When the move ctor will be called? Below are some examples.

```

1. class obj = 2 // same as class obj(2)
2. // call single argument ctor
3.
4. obj = 2 // call assignment operator
5.
6. class obj1 = obj2 // call copy ctor
7. // not call assignment operator
8.
9. obj1 = obj2 // call assignment operator
10.
11. class obj1 = obj2+obj3 // call move ctor
12. // if you don't have move ctor, It will call copy ctor
13.
14. obj1 = obj2+obj3 // call move assignment operator if you define.
```

- In below examples, Foo obj1=obj2+obj3. if you don't have move ctor, In operator + function, a temp obj is produced, and when operator+ function return, another temp obj temp2 is produced. Then in the end, objtemp2 is passed to ctor, So, ctor is called three times. and copy content is also called three times.

```

1. Foo Foo::operator+( const Foo & f ) const {
2.     Foo temp = Foo(n+f.n);
3.     //copy happen here.
4.     return temp;
5. }
6.
7. Foo obj1=obj2+obj3 // three ctor called with move ctor
```

- If you have move ctor. In operator + function, a temp objtemp1 is produced, When return objtemp1, It will not produce objtemp2. (because objtemp1 is rvalue.) then objtemp1 is passed to move ctor. In side move ctor, the resource address has been move to new obj1. Just one

temp objtemp1 and one actual copy happen. ( just new pointer = old pointer; and old pointer = NULL).

### 6.3.3 move semantic

- First, Don't declare objects `const` if you want to be able to move from them. Move requests on `const` objects are silently transformed into copy operations.
- Second, `std::move` not only doesn't actually move anything, it doesn't even guarantee that the object it's casting will be eligible to be moved. The only thing you know for sure about the result of applying `std::move` to an object is that it return a `xvalue`.

```
1. explicit Annotation(const std::string text)
2. : value(std::move(text)) // "move" text into value; this code
3. // doesn't do what it seems to, because text is const
```

- Move ctor and move assignment work with rvalue, What if you want to use them with lvalues? You can call `std::move` function, It will call you move ctor or assignment to "move" resource, not "copy" resource.

```
1. Foo choices[10];
2. Foo best;
3. best = choices[3];
4. //I don't want to keep choices after I pick up what I want
5. //here, It will call normal move assignment,
6. //because choices[3] is not rvalue.
7.
8. best = std::move(choices[3]);
9. //It will call move assignment
```

## 6.4 universal(forwarding) reference

### 6.4.1 definition

- You need to know two prerequisites before you understand forwarding reference:

1. In pre-11 C++, it was not allowed to take a reference to a reference: something like `A&&` would cause a compile error. C++11, by contrast, introduces the following reference collapsing rules1:

```
1. A& & becomes A&
2. A& && becomes A&
3. A&& & becomes A&
4. A&& && becomes A&&
```

2. forwarding reference type deduction rules: For lvalue and rvalue, we use the different rule to deduct type.

- Universal reference has been renamed as **forwarding reference**. It's more descriptive name, It tell you universal reference should always been used with forwarding.
- Universal references arise in two contexts. The most common is function template parameters. The second context is `auto&&1)must be constrained T&& form, 2) type deduction happen.`

- If the form of the type declaration isn't precisely `type&&`, or if type deduction does not occur, `type&&` denotes an rvalue reference.

```

1. auto&& var2 = var1; // universal reference
2.
3. template<typename T>
4. void f(T&& param); // universal reference
5.
6. template<class T, class Allocator = allocator<T>>
7. class vector {
8.     template <class... Args>
9.     void emplace_back(Args&&... args); //args is universal reference
10. };
11.
12. //—————below are not universal reference—————
13. template<typename T>
14. void f(std::vector<T>&& param); // rvalue reference
15. // form is quite constrained. It must be precisely "T&&".
16.
17. template<typename T>
18. void f(const T&& param); // with const
19.
20. template<class T, class Allocator = allocator<T>> // from C++
21. class vector { // Standards
22. public:
23.     void push_back(T&& x); //no type deduction
24. };

```

- For `auto&&`, basic idea just like universal reference, you need to use `decltype(var)` to get type information when you use `forward` function on the universal reference.
- If you then use **`std::forward` on your `auto&&` reference** to preserve the fact that it was originally either an lvalue or an rvalue, your code says: Now that I've got your object from either an lvalue or rvalue expression, I want to preserve whichever valueness it originally had so I can use it most efficiently.

```

1. auto&& var = some_expression_that_may_be_rvalue_or_lvalue;
2. // var was initialized with either an lvalue or rvalue, but var itself
3. // is an lvalue because named rvalues are lvalues
4. use_it_elsewhere(std::forward<decltype(var)>(var));

```

- When to use `auto&&`. I will accept any initializer regardless of whether it is an lvalue or rvalue expression and I will preserve its constness. This is typically used for forwarding (usually with `T&&`). The reason this works is because a "universal reference", `auto&&` or `T&&`, will bind to anything. A good example of

"some\_expression\_that\_may\_be\_rvalue\_or\_lvalue;"

```

1. std::vector<int> global_vec{1, 2, 3, 4};
2.
3. template <typename T>
4. T get_vector() {
5.     return global_vec;
6. }
7.
8. template <typename T>
9. void foo() {
10.     auto&& vec = get_vector<T>();
11.     //only auto&& work here.

```

```

12. // auto , auto& , const auto&, const auto&& all failed .
13. auto i = std::begin(vec);
14. (*i)++;
15. std::cout << vec[0] << std::endl;
16. }
17.
18. foo<std::vector<int>>();
19. std::cout << global_vec[0] << std::endl;
20. foo<std::vector<int>&>();
21. std::cout << global_vec[0] << std::endl;

```

### 6.4.2 pros

- It can provide the unify interface, and it supports variadic number. That is make\_unique and make\_shared and emplace-kind function possible.
- If you have a template class or template fun, universal reference is your only choice. An example can be seen in the last chapter, "decltype deduction" section.
- For overload method, more source code to write and maintain (two functions instead of a single template).
- For overload method, it can be less efficient. For example, consider this use of setName: w.setName("Adela Novak"); With the version of setName taking a universal reference, the string literal "Adela Novak" would be passed to setName, where it would be conveyed to the assignment operator for the std::string inside w. w's name data member would thus be assigned directly from the string literal; no temporary std::string objects would arise. With the overloaded versions of setName, however, a temporary std::string object would be created for setName's parameter to bind to, and this temporary std::string would then be moved into w's data member.

```

1. class Widget {
2. public:
3.     template<typename T>
4.     void setName(T&& newName) // newName is universal reference
5.     { name = std::forward<T>(newName); }
6.
7.     ...
8.     string name;
9. };

```

- overload has the poor scalability of the design. Widget::setName takes only one parameter, so only two overloads are necessary, but for functions taking more parameters, each of which could be an lvalue or an rvalue, the number of overloads grows geometrically: n parameters necessitates 2n overloads. Such as make\_shared function, It's also support variadic parameter

```

1. template<class T, class ... Args> // from C++11
2. shared_ptr<T> make_shared(Args&&... args); // Standard
3.
4. template<class T, class ... Args> // from C++14
5. unique_ptr<T> make_unique(Args&&... args); // Standard

```

### 6.4.3 cons

- You can use universal reference, but inside, you have to use forward function, and implementation is a little difficult. Detail can be seen "effective modern c++ item 41". As a template,

implementation must typically be in a header file. It may yield several functions in object code, because it not only instantiates differently for lvalues and rvalues, it also instantiates differently for std::string and types that are convertible to std::string (see Item 25).

- There are argument types that can't be passed by universal reference (see Item 30), and if clients pass improper argument types, compiler error messages can be intimidating (see Item 27).

```

1. template<typename T> // reference
2. fun(T&& value) {
3.     vector<Foo> vect;
4.     vect.push_back(std::forward<T>(value)); // use move here to implement
5. }
```

- Overloading on universal references almost always leads to the universal reference overload being called more frequently than expected.
- Perfect-forwarding constructors are especially problematic, because they're typically better matches than copy constructors for non-const lvalues, and they can hijack derived class calls to base class copy and move constructors. detail can be seen in "effective modern c++ item 26"

#### 6.4.4 Usage of forwarding reference

- When to use universal references? universal means that:
  1. you have to support different type or a lot of parameter.
  2. You have to use reference, means that need refer a existing one, you refer it because you want to copy it inside of your template function.
- there are some points here:
  1. **The first of first, it's only used in a template function.**
  2. **Inside the function, A copy will happen, if just read or write, use reference or const reference directly**
  3. **When copy, move is cheap, if there is no pointer or container, only has POD type value, move is just like copy. You don't need to use universal references.** Most of time, in your universal function, you have a container(string is container too), because move container is much cheaper than copy it.
  4. **For universal reference, you must forward it to ctor, container, object or function.** and these four things will take different action toward lvalue and rvalue.
  5. **There are more than 2 parameters, and overload function will cause exponential increase.**
  6. I will accept any initializer regardless of whether it is an lvalue or rvalue expression and I will preserve its constness.
  7. This is typically used for forwarding (usually with T&&). The reason this works is because a "universal reference", auto&& or T&&, will bind to anything.
  8. You might say, well why not just use a const auto& because that will also bind to anything? The problem with using a const reference is that it's const! You won't be able to later bind it to any non-const references or invoke any member functions that are not marked const.

```

1. auto&& vec = some_expression_that_may_be_rvalue_or_lvalue;
2. auto i = std::begin(vec);
3. (*i)++;
4.
5. auto          // will copy the vector, but we wanted a reference
6. auto&        // will only bind to modifiable lvalues
7. const auto&
8. // will bind to anything but make it const, giving us const_iterator
9. const auto&& // will bind only to rvalues
10. // with const, It's not universal reference any more.

```

- There are three function emplace, make\_shared and make\_unique. They use:

1. **variadic template**, because it need to receive **any number and any type** parameter.
2. **universal reference**, because it's template, and I also want to keep rvalue semantic to improve efficiencie.
3. **Forward**, because I want to forward parameter to corresponding ctor. Forward only used inside a wrapper, that is to say, to receive universal reference from template wrapper, and forward it to the specific function. The specific function has "COPY" semantic.

#### 6.4.5 std::move and std::forward implementation

- Basic implementation of move

```

1. template<typename T> // in namespace std
2. typename remove_reference<T>::type&&
3. move(T&& param) {
4.     using ReturnType = typename remove_reference<T>::type&&; // see Item 9
5.     return static_cast<ReturnType>(param);
6. }

```

- Basic implementation of forward

```

1. template<typename T>
2. T&& forward(typename remove_reference<T>::type& param) {
3.     return static_cast<T&&>(param);
4. }

```

- basic usage of forward is like this:

```

1. template<class T>
2. void wrapper(T&& arg) {
3.     // arg is always lvalue
4.     foo(std::forward<T>(arg));
5.     // Forward as lvalue or as rvalue, depending on T
6. }

```

- In forward function, why parameter is std::remove\_reference\_t<T>& t?

```

1. void fun(int&& r) {
2.     // int& r1 = r; //OK
3.     int&& r1 = r; //error: cannot bind 'int' lvalue to 'int&&'
4.     cout << r1 << endl;
5. }
6. //-----
7. int r = 2;
8. fun(std::move(r));

```

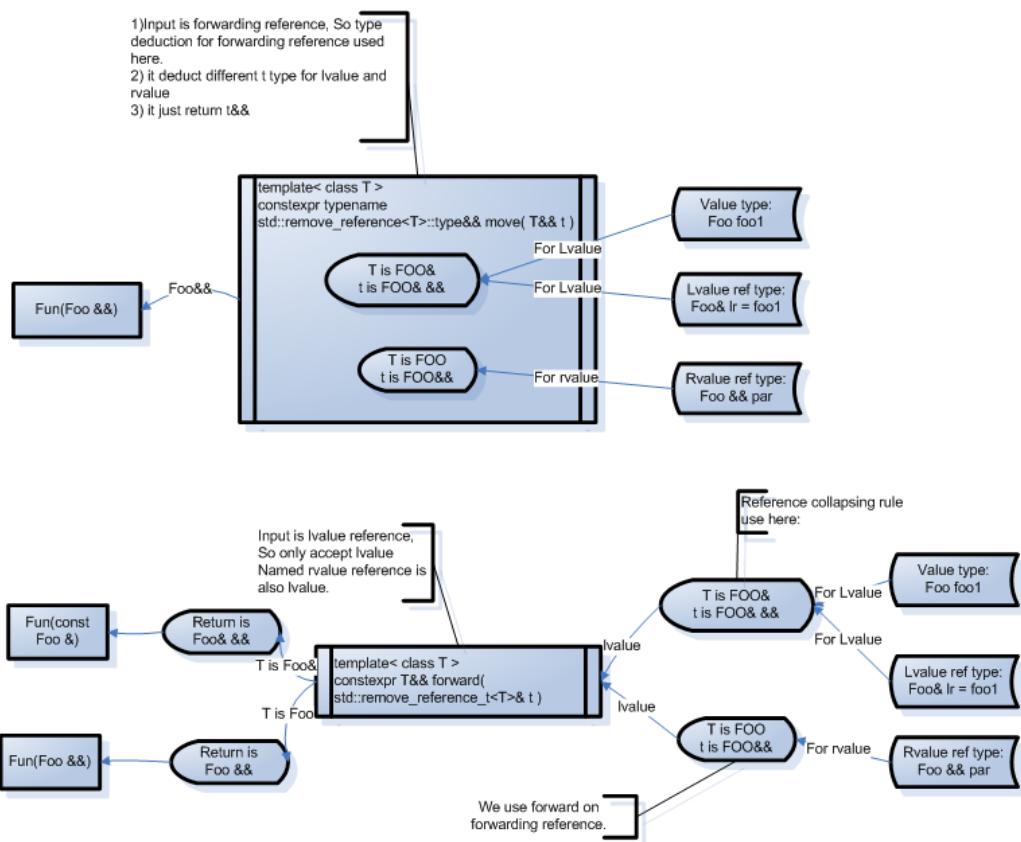
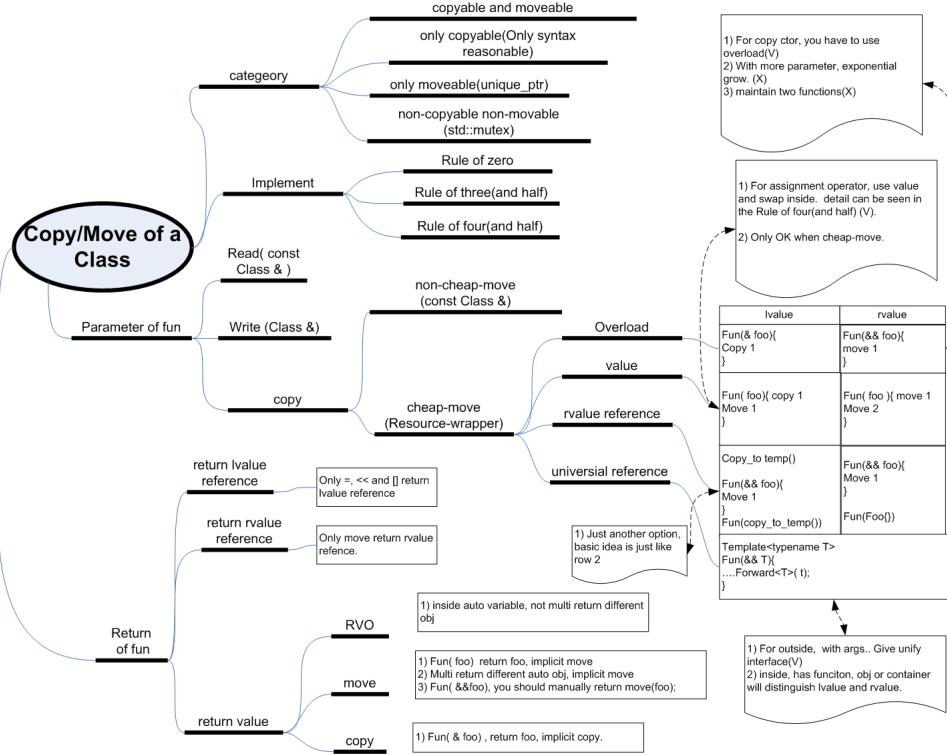


Figure 6.3:

## 6.5 function interface-parameter

### 6.5.1 Generic function parameter design

- Main content in this section illustrated below:  
copyable and movable



- By now, you imagine you have a function with parameter, then you have three main operations inside to operate on the parameter: **1) only read 2) copy from the parameter 3) write to parameter.**

- For only read, "const type&" is the best, For write, "type&" is good, because I don't need to write any new thing into prvalue. Based on previous explanation, below code will not compile.

```
1. writeFun( Foo &);  
2.  
3. writeFun( foo1+foo2 ) // will not compile , foo1+foo2 is rvalue .
```

- For only read build-in type, such as int, char, short, double.. You can use value directly, it has the same performance as reference. If you want to change it, you still need to use reference.
  - If the function read in some values and **build** other something based on the value which just read in. You need to consider the parameter design. When I say **build**, it could be:

1. call ctor inside fun
  2. assign these values to other.
  3. put values to container.

### 6.5.2 read-copy function parameter design

- For read-copy, things become interesting. Because the move constructor and move assignment operator give us an idea how to improve the function interface design. For rvalue, we can move directly. Our goal is that we can use move for rvalue, so there are four options:

1. overload fun to support const Foo& and Foo &&.
2. Use pass-value fun(Foo)
3. Use pass-rvalue-reference fun(Foo&&)
4. Use universal reference

	lvalue	rvalue
copyFun(Foo foo)	copy parameter move inside	move parameter move inside
copyFun(Foo &)	copy inside	(NOT support)
copyFun(const Foo &)	copy inside	copy inside
copyFun(Foo &&)	(NOT support)	move inside
template<T&&>	copy inside	move inside
copyFun(T &&)		

### 6.5.2.1 overload solution

- Use Overload function to deal with rvalue only. So usually function which accepts rvalue reference doesn't exist by itself, it just stay with another version to accept lvalue, fun(const Foo& foo);

```

1. //rvalue reference
2. fun(Foo&& foo);
3. fun(foo1+foo2) // will compile, It means that inside fun
4. // you will steal resource.
5. fun(f_return_foo()); // call move ctor.

```

- The most common use of this is move ctor and move assignment.

### 6.5.2.2 only value solution

- There is such scenarios:

1. You have origin obj, but in your function , you want to copy from it, then modify, at last return this copied one
2. Obviously, in your function, you have to return value.
3. At this time you have two different options. Value parameter or reference parameter.

```

1. std::vector<std::string>
2. sorted(std::vector<std::string> names) {
3.     std::sort(names);
4.     return names;
5. }
6.
7. // names is an lvalue; a copy is required so we don't modify names
8. std::vector<std::string> sorted_names1 = sorted( names );
9.
10. // get_names() is an rvalue expression; we can omit the copy!
11. std::vector<std::string> sorted_names2 = sorted( get_names() );

```

```

1. std::vector<std::string>
2. sorted2( std::vector<std::string> const& names) {
3.     std::vector<std::string> r( names );      // and explicitly copied
4.     std::sort(r);
5.     return r;
6. }

```

- A basic explanation can be found here:

	Implementation	lvalue	rvalue
Method 1 Only value	Foo Fun(Foo foo){ //change foo Return foo; }	Foo foo = Fun(foo1); //one copy in parameter //one move in return	Foo foo = Fun(Foo()); //one ctor in parameter with copy-elision //one move in return.
Method 2 overload	Foo Fun(Foo& foo){ Foo foo1(foo); //change foo1; Return foo1; }	Foo foo = Fun(foo1); //one copy in foo1; //copy elision in return	XXX
	Foo Fun(Foo&& foo){ Return std::move(foo); Return foo; }	XXX	Foo foo = Fun(Foo()); // with std::move, one move // without move, one copy
Method3 const reference	Foo Fun(const Foo& foo){ Foo foo1(foo); //change foo1; Return foo1; }	Foo foo = Fun(foo1); //one copy in foo1 //copy elision in return	Foo foo = Fun(Foo()); //one copy in foo1 //copy elision in return

Left value input	ctor	Copy	move
Method1	1	1	1
Method2	1	1	0
Method3	1	1	0

Right value input	ctor	Copy	move
Method1	1	0	2(1)
Method2	1	0	1
Method3	1	1	0

- For method 1 for rvalue, if you use -fno-elide-constructors, then it use 2 move, otherwise use 1 move
- Method 2 is the best, but you need to write two overload function.
- If don't use method 2. Previous lessons told us that reference is more efficient than value(it can avoid coping). But in our specific scenario(we still copy inside the function even we use reference), if move is cheaper than copy, then method3 is better. Although for lvalue, It use one more move, but for rvalue, it also use move, not copy.

- basic analysis:

1. For method 3, When return, you can have RVO, no copy or move when we return. Only one copy inside the function.
2. In method1, when return for lvalue, you have one move, It's bad and color is blue.
3. In method1, for rvalue, although one more move, but It doesn't need a copy (Copy-elision at parameter, compared with method3 red part)

- summary:

1. For c++03, we don't have move semantic, all the move will be decayed to copy. **So reference WIN!**
2. For non-cheap-move, such as all primitive type included class, move is just like copy, **So reference WIN!**
3. For c++11 and cheap-move semantic, **value WIN!**

4. For = operator, we don't need return value, but return reference, In this way, **value WIN!**.  
 Detail can be found in use swap to implement = operator below.

- The same idea can be seen in the "more effective C++ item 41"
- You can apply this guideline immediately is in assignment operators. The canonical, easy-to-write, always-correct, strong-guarantee, copy-and-swap assignment operator is often seen written this way:

```

1. T& T::operator=(T const& x){ // x is a reference to the source
2.   T tmp(x);           // copy construction of tmp does the hard work
3.   swap(*this, tmp);  // trade our resources for tmp's
4.   return *this;       // our (old) resources get destroyed with tmp
5. }
6.
7. // A better one is below
8. T& operator=(T x){// x is a copy of the source;hard work already done
9.   swap(*this, x); // trade our resources for x's
10.  return *this; // our (old) resources get destroyed with x
11. }
```

- Two good articles about this topic are:"Want Speed? Pass by Value." and "WANT SPEED? DON'T (ALWAYS) PASS BY VALUE."

### 6.5.2.3 only rvalue reference solution

- A more academic optoins is to just pass rvalue reference to deal with lvalue and rvalue at the same time. This idea can be googled by "Pass By Rvalue Reference Or Pass By Value";

```

1. struct S{
2.   void initByRef(SomeType&& param);
3. };
4.
5. SomeType t;
6. s.initByRef(std::move(t)); // this works, and even better (less moves)
7. s.initByRef(SomeType()); // this also works
```

- But for lvalue, you need a help function here.

```

1. template<typename T>
2. T copy_to_temp(const T& t) { return t; }
3.
4. SomeType t; // I'll need t later, so cannot std::move(t)
5. s.initByRef(copy_to_temp(t)); // but can copy
6.
7. s.initByVal(t); // note that here t is also copied, it just happens implicitly
```

- the same idea can be applied to move-only types (such as unique\_ptr<T>)

```

1. template<typename T>
2. T move_to_temp(T& t) { return std::move(t); }
3.
4. void foo(unique_ptr<SomeType>&& param);
5.
6. unique_ptr<SomeType> p;
7. foo(move_to_temp(p)); // note: this does move, p will become empty
8. foo(std::move(p)); //here, p will not become empty.
```

- More interesting example:

```

1. template<typename T>
2. T move_to_temp1(T& t) { return std::move(t); }
3. //here move ctor is called when return to a value
4.
5.
6. T&& move_to_temp2(T& t) { return std::move(t); }
7. //here no move ctor is called, just reference value assignment.
8.
9. void foo(unique_ptr<SomeType>&& param);
10.
11. unique_ptr<SomeType> p;
12. foo(move_to_temp1(p)); //compile ok, return value is prvalue
13. foo(move_to_temp2(p)); //compile ok, return value is xvalue

```

#### 6.5.2.4 forwarding reference

- The last option is universal reference. At this time, you have to use std::forward

```

1. template<typename T>
2. Fraction // by-value return
3. reduceAndCopy(T&& frac) // universal reference param
4. {
5.     frac.reduce();
6.     return std::forward<T>(frac); // move rvalue into return
7. }

```

## 6.6 function interface-return

- These are two most important knowledge to understand all the detail below.

1. For no RVO, there are two steps when we return value.
2. For RVO, we implicit pass the result by reference.

- Three basic knowledges about function return when no RVO:

1. It's very important understand there are two phrases when you return from function. The first step is from inside function to outside of function ftemp(unname temporary), then ftemp disappear(call destructor for value). Then in second step, **Move** from ftemp to flast. because ftemp is rvalue.

```

1. Foo fun() {
2.     return fauto;
3. }
4.
5. Foo flast = (ftemp created here)fun();

```

2. ftemp is not on the stack, so you can use const ref or rref to prolong its life.
3. ftemp is same with fauto, but maybe not same with flast

```

1. Foo fun() {
2.     return fauto;
3. }
4.

```

```

5. \\fun return Value, if run return reference, it's dangerous.
6. const Foo& flast = (femp created here) fun();
7. Foo&& flast = fun();

```

- The basic RVO is implemented by this way:

```

1. X bar() {
2.     X xx;
3.     return xx;
4. }
5.
6. void bar(x &__result) {
7.     __result.X::X() // default ctor call
8.     ...
9.     return
10. }

```

### 6.6.1 return plain reference

- Don't return reference or pointer to private member variables through you public member functions. It will break encapsulation.**
- Now talk about return reference: Never return reference which refers to local variable, it will cause dangling problem, So return plain reference only happen:
  - You input a reference first, such as overload <<.
  - member function return some member data, such as copy ctor and overload [] inside a class
- About return reference, by now, I only know three functions which return reference. = and << are for support cascading syntactic usage: such as cout<<a<<b, a=b=c. [] is for support assignment obj[3]= 12. That is all.

```

1. operator =
2. operator []
3. operator << and >>

```

- When the client programmer does something like this and uses a reference beyond its lifetime, the bug will typically be intermittent and very difficult to diagnose. Indeed, one of the most common mistakes programmers make with the standard library is to use iterators after they are no longer valid, which is pretty much the same thing as using a reference beyond its lifetime

```

1. string& a = FindAddr( emps, "John Doe" );
2. emps.clear(); // This statement will invalid a.
3. cout << a; // may or may not work, It's difficult to debug.

```

- If you want to return reference, there is a defensible option that allows returning a reference and thus avoiding a temporary. But it's your last resort.

```

1. const string&
2. FindAddr( /* pass emps and name by reference */ ) {
3.     for( /* ... */ ) {
4.         if( i->name == name ) {
5.             return i->addr;
6.         }
7.     }
8.     static const string empty;

```

```

9.     return empty;
10.

```

- Why **don't** we usually return plain reference?

1. You want to return a reference to avoid copy of auto obj inside of function, but in fact it's totally wrong. We have RVO and implicit std::move.
2. If you return non-auto obj, You have to 1) new a obj, in this way, you can return pointer directly. 2) input a reference for read, in this way, you can input const reference, You don't need to return it at all. 3) input a reference for write, in this way, you don't need return it either, modification will act on inputted reference directly. So when do we use return plain reference?

### 6.6.2 return rvalue reference

- **rvalue reference is also reference first, so if we have reason don't return reference from function, all these reason is also valid for rvalue reference.**
- The most common case for returning rvalue reference is std::move. It doesn't involve any move action inside the function, but is just a type cast operation.
- return rvalue reference 1: If you want to return plain reference, or rvalue reference from a function, you have to input a plain reference or rvalue reference first, because you can't return any reference bound to local auto obj.

```

1. A&& rrfun1(A&& arg) {
2.     return std :: move(arg);
3.     //You have to use move here, because arg is lvalue.
4. }
5.
6. A&& rrfun2(A& arg) {
7.     return std :: move(arg);
8. }
9.
10. A a;
11. A b= rrfun1(std :: move(a));
12. A b = rrfun2(a);

```

- return rvalue reference 2: below code will call move ctor once. 1) move ctor from arg to ftemp, 2) rvalue reference b bound to ftemp(rvalue). Only reasonable in syntax, No any practical meaning, and **It's dangerous,because a has been empty..**

```

1. A rrfun(A& arg) {
2.     return std :: move(arg);
3. }
4.
5. A a;
6. A&& b = rrfun(a);
7. //after this a is invalid, and b refer a ftemp

```

- return rvalue reference 3: below code will not call move ctor at all. So you mean that I want to keep watch for a while, and obj a is still intact right now.

```

1. A&& rrfun(A& arg) {
2.     return std::move(arg);
3. }
4.
5. A a;
6. A&& b = rrfun(a);

```

- You can understand all these examples by three function knowledge when no RVO in the begining of this section. And all these exmaples are just have academic meaning, no any practical sense at all. Why?
- Why return value is better than return rvalue reference?
  1. more clear semantic
  2. when use with auto, it's support RVO.
- In Summary 1, Three operator overload return plain reference, one std::move return rref. It doesn't involve value semantic, just a type change. All the others return value, That's all!
- In summary 2, If function return value, For only copy lvalue, return value; for rvalue reference, move; For universal refence, forward.

```

1. Matrix operator+(/Hilight5Matrix& lhs, const Matrix& rhs) {
2.     return lhs
3. }
4.
5. Matrix operator+(/Hilight5Matrix&& lhs, const Matrix& rhs) {
6.     return move(lhs)
7. }
8.
9. template<type T>
10. T operator+(/Hilight5T&& lhs) {
11.     return forward<T>(lhs)
12. }

```

- The ref-qualifier && says that the second function is invoked on rvalue temporaries, making the following move, instead of copy

```

1. struct Beta {
2.     Beta_ab ab;
3.     Beta_ab const& getAB() const& { return ab; }
4.     Beta_ab && getAB() && { return move(ab); }
5.     //Hilight30// return && is not Good interface design.
6.
7.     Beta_ab ab = Beta().getAB();
8.     // It will call move version.
9.     //Beta_ab && ab = Beta().getAB(); ab is dangling rref
10.
11.    2) Beta_ab getAB() && { return move(ab); }
12.    /Hilight30//Good interface design.
13. };

```

- the same idea just like previous example, but this time I use auto.

```

1. DataType data() && { return std::move(values); } // why DataType?
2. auto values = makeWidget().data();
3. //with ROV, just move once.
4.
5. DataType && data() && { return std::move(values); }
6. auto&& values = makeWidget().data();
7. //values will be dangling because makeWidget() return value disappear.

```

- about rvalue reference qualifier, please google "C++ Gems: ref-qualifiers"
- A few article you need to read later;
  - 1) Efficiency of C++11 push\_back() with std::move versus emplace\_back() for already constructed objects
  - 2) view the default functions generated by a compiler?
  - 3) One variable init form to rule them all, via mandatory elision.
  - 4) Episode Eleven: To Kill a Move Constructor

### 6.6.3 return value-RVO

#### 6.6.3.1 common RVO case

- RVO is a kind of copy elision.
  1. it happens when you return value from a function.
  2. The type of the **local** object is the same as that returned by the function.
  3. the local object is what's being returned.
  4. parameter is not eligible for RVO.
- Three common cases are:
  1. Name RVO, t has a name. It's called NRVO.

```

1. Thing f() {
2.     Thing t;
3.     return t;
4. }
5. Thing t2 = f();

```

2. RVO, No name, just return temporary.

```

1. Thing f() {
2.     return Thing();
3. }
4. Thing t2 = f();

```

3. temporary is passed by value

```

1. void foo(Thing t);
2.
3. foo(Thing());

```

- 4. exception is thrown and caught by value

```

1. void foo() {
2.     Thing c;
3.     throw c;
4. }
5.
6. int main() {
7.     try {
8.         foo();
9.     }
10.    catch(Thing c) {
11.    }
12. }
```

### 6.6.3.2 RVO limitations

- Different named object sample will not trigger RVO

```

1. RVO MyMethod (int i)
2. {
3.     RVO rvo;
4.     rvo.mem_var = i;
5.     if (rvo.mem_var == 10)
6.         return RVO();
7.     return rvo;
8. }
```

- returning a parameter or Global

```

1. Snitch global_snitch;
2.
3. Snitch ReturnParameter(Snitch snitch) {
4.     return snitch; //no RVO here
5. }
6.
7. Snitch ReturnGlobal() {
8.     return global_snitch; //no RVO here either
9. }
```

- return by std::move()

```

1. Snitch CreateSnitch() {
2.     Snitch snitch;
3.     return std::move(snitch);
4. }
```

- In some cases even an unnamed variable can't RVO:

```

1. struct Wrapper {
2.     Snitch snitch;
3. };
4.
5. Snitch foo() {
6.     return Wrapper().snitch;
7. }
8.
9. int main() {
10.     Snitch s = foo();
11. }
```

### 6.6.3.3 Some practical demos ans analysis

- given below code as experiment code:

```

1. class Snitch {    // Note: All methods have side effects
2.     Snitch() { cout << "c'tor" << endl; }
3.     ~Snitch() { cout << "d'tor" << endl; }
4.
5.     Snitch(const Snitch&) { cout << "copy_c'tor" << endl; }
6.     Snitch(Snitch&&) { cout << "move_c'tor" << endl; }
7.
8.     Snitch& operator=(const Snitch&) {
9.         cout << "copy_assignment" << endl;
10.        return *this;
11.    }
12.
13.    Snitch& operator=(Snitch&&) {
14.        cout << "move_assignment" << endl;
15.        return *this;
16.    }
17. };
18.
19. Snitch CreateSnitch() {
20.     return Snitch();
21. }
```

- test 1 and output

```

1. int main() {
2.     Snitch s = CreateSnitch();
3. }
4.
5. // with -fno-elide-constructors
6. // c'tor      //Snitch()
7. // move c'tor //Snitch() to return value
8. // d'tor      //Snitch() destroy
9. // move c'tor //return value to s
10. // d'tor      //return value destroy
11. // d'tor      //s destroy
12. //
13. // without -fno-elide-constructors
14. // c'tor      //s
15. // d'tor      //s destroy
```

- test 2 and output

```

1. int main() {
2.     Snitch s;
3.     s = CreateSnitch();
4. }
5.
6. // with -fno-elide-constructors
7. // c'tor      //Snitch s
8. // c'tor      //Snitch() in side CreateSnitch
9. // move c'tor //Snitch() to return value
10. // d'tor      //Snitch() destroy
11. // move assignment //return value to s
12. // d'tor      //return value destroy
13. // d'tor      //s destroy
14. //
```

```

15. // without -fno-elide-constructors
16. // c'tor //Snitch s
17. // c'tor //Snitch()
18. // move assignment //s has been implicit passed to CreateSnitch,
19. // so s = Snitch() happen inside CreateSnitch
20. // d'tor //Snitch() destroy
21. // d'tor //s destroy

```

- test 3 and output

```

1. Snitch CreateSnitch() {
2.     return Snitch();
3. }
4.
5. Snitch CSnitch(Snitch&& rs) {
6.     return rs;
7. }
8.
9. int main() {
10.     Snitch s = CSnitch(CreateSnitch());
11. }
12.
13. // with -fno-elide-constructors
14. // c'tor //
15. // copy c'tor //
16. // d'tor //
17. // copy c'tor //
18. // copy c'tor //
19. // d'tor //
20. // d'tor //r
21. // d'tor //
22. // -----
23. // without -fno-elide-constructors
24. // c'tor //Snitch s
25. // copy c'tor //
26. // d'tor //Snitch() destroy
27. // d'tor //s destroy

```

#### 6.6.3.4 RVO and move

- For local variable, don't use move. compiler will use RVO , so It has already had high efficiency.

```

1. Foo fun() {
2.     Foo foo;
3.     return foo;
4.     //Don't use move here.
5. }

```

- A common error is when there is no RVO, use std::move to avoid copy: but Although parameter is not eligible for RVO, or different path is not eligible, but you don't need to explicit use std::move, the compiler will use them implicitly.

```

1. Widget makeW(Widget w) {
2.     .....
3.     return w
4.     //return std::move(w) //don't write it, compiler will do it for you.
5. }

```

- If a function is value return, and you want to return a rvalue reference parameter, you have to use move in the return.

### 1. lhs is not local object, but reference, so No RVO

2. you may say: but lhs is rvalue reference, but rhs is name rvalue reference. It's lvalue.

3. Just like you pass a lvalue reference, you can't move it unless you want to do it explicitly.  
That is why we need std::move here.

```

1. Matrix // by-value return , by rvalue parameter
2. operator+(Matrix&& lhs, const Matrix& rhs) {
3.     lhs += rhs;
4.     return std::move(lhs); // move lhs into return value
5.     //return lhs // will copy lhs into return value
6. }
```

#### 6.6.3.5 RVO summary

- First use RVO, then compiler implicit use std::move, the last one is use std::move explicitly.

1. Apply std::move to rvalue references and std::forward to universal references the last time each is used.
2. Never apply std::move or std::forward to local objects if they would otherwise be eligible for the return value optimization.

# Chapter 7

## OOP

### 7.1 Object based

#### 7.1.1 class categories

- Basic class categories:
  1. Value class, such as std::pair, std::vector, std::string.
    - (a) Has a public destructor, copy ctor and assignment with value semantics
    - (b) Has no virtual function. so intended to be used as a concrete class, not as a base class.
    - (c) instantiated on stack or as a member of an other class.
  2. Base class.
    - (a) Has a destructor that is public and virtual, But for some trait class, destructor can be protected, such as std::unary\_function
    - (b) Establish interface.
    - (c) Usually instantiated on heap, and used via a (smart) pointer or reference to support polymorphism.
  3. Trait class.
    - (a) Contain only typedef and static functions, It has no modifiable state.
    - (b) Is not instantiated( ctor is private or disable)
    - (c) Usually instantiated on heap, and used via a (smart) pointer.
- Policies are classes (or class templates) to **inject behavior** into a parent class, typically through inheritance. Through decomposing a parent interface into orthogonal (independent) dimensions, policy classes form the building blocks of more complex interfaces. An often seen pattern is to supply policies as user-definable template (or template-template) parameters with a library-supplied default. An example from the Standard Library are the Allocators, which are policy template parameters of all STL containers

```
1. template<class T, class Allocator =  
2.     std::allocator<T>> class vector;
```

- Traits are class templates to **extract properties** from a generic type. There are two kind of traits: single-valued traits and multiple-valued traits. Examples of single-valued traits are the ones from the header <type\_traits>. Single-valued traits are often used in template-metaprogramming and SFINAE tricks to overload a function template based on a type condition.

```

1. template< class T >
2. struct is_integral{
3.     static const bool value
4.     /* = true if T is integral, false otherwise */;
5.     typedef std::integral_constant<bool, value> type;
6. };
7.
8. template <class T>
9. T f(T i){
10.     static_assert (std::is_integral<T>::value, "Integer_required.");
11.     return i;
12. }
13.
14. int main() {
15.     std::cout << f(123) << '\n'; // output 123
16. }
```

### 7.1.2 Interface

- Nesting a class does not create a class member of another class. Instead, it defines a type that is known just locally to the class that contains the nested class declaration. A good example is Class queue nest class node, because node is just used inside the class Queue. Another good example is vector and its iterator.
- Virtual function must be member, operator>> and << are never be members, or It maybe be a friend. Only non-member functions get type conversions on their left-most argument. In the previous example, If you want to use support 2\* obj, You need make operator \* to be non member function. Detail can be seen in effective c++.
- Keep in mind that only a class declaration can decide which functions are friends, so the class declaration still controls which functions access private data.
- Protect keyword don't use very often, it is just used in inheritance context. Child class can access base class protected member. you should use private keyword first if you real want to have good **Encapsulation** and **Never return reference or pointer to a private or protected member data**.
- In C++ primer p653, you can see a good example class interface. You should remember it as a basic pattern. If you use new allocate memory inside of your class, you should define: copy ctor, assignment operator and destructor, move copy ctor, and move assignment.

```

1. //below is string.h file
2. #pragma once
3. namespace Yan{
4.     class String{
5.     public:
6.         String(); //default constructor
7.         String(const char *a); // specify constructor
8.
9.         String (const String &); //copy ctor
10.        String (String && other); //move copy ctor
11.
12.        String& operator=(const String &); //assignment
13.        String& operator=(String&& other); //move assignment
14.        String& operator=(const char*a); // option.
15. }
```

```

16.     ~String() ; //usually , you should have these Seven member
17.     //functions if you use new inside your class .
18.
19.     friend ostream& operator<<(ostream & os, const String & st) ;
20.     friend istream& operator>>(istream & is , String &st) ;
21.
22. private :
23.     const static int NUM= 1000; // const used inside of this class .
24.     char* m_str;
25.
26. };
27. ostream& operator<<(ostream & os, const String & st) ;
28. istream& operator>>(istream & is , String &st) ;
29. }
```

1. Put class definition into a namespace.
2. Use #pragma once
3. You need to declare operator << inside of namespace outside of class
4. If you don't use smart pointer and allocate use **new operator**. you should follow five rules.( including move ctor and move assignment) if you class includes a resource.
5. Member function can access all the instance private data, such as other.m\_str, and no semicolon after each function.

```

1. //user .cpp
2. String s;
3. s = "aaa" //two actions
4. String s("aaa"); //one actions
5. String s{"aaa"}; // new feature in c++11
6. String s={"aaa"}; // same as previous one
7.
8. String str; char temp[40];
9. str= temp // make it more efficient
```

- `String& operator=(const char*a);` is an option, why it make `str=temp` more efficient, see C++ Primer P652
- Friend has three categories:
  1. Friend Class: just as TV and RemoteControl, you can declare RemoteControl a friend class inside TV.
  2. Friend Member functions: You can select some member functions to be friend of another class, In this way, you need forward declaration. When you write class TV; It doesn't define a TV class, it just tell compile, TV is a class, definition can be done later.
  3. Common Friend method, a good example is overload `operator <<`
- Prefer minimal classes to monolithic classes: big class is difficult to reach error-safe because it tackle multiple responsibilities. It's also difficult maintain, understand and deploy.

```

1. Class Matrix{
2.     //100 member function .
3. } ; //bad design .
4.
5. //Good design with small class with
6. nonmember function .
```

```

7. | namespace MAT{
8. |   Class Matrix{
9. |     //core data and member function
10. |   }
11.
12. |   Cal1( Matrix &);
13. |   Cal2( Matrix m1, Matrix m2)
14. |   .....
15. |

```

- Interface Principle: For a class X, all functions, including free functions, that both:

1. "Mention" X.
2. Are "supplied with" X

are logically part of X, because they form part of the interface of X. In this definition, 1) Cal1 Mention X, and 2) Cal1 in the namespace MAT, so caller can use ADL(argument depend lookup) loop Cal1 in namespace MAT, so **Cal1 is an interface of class Matrix, even it's not a member function of it.** Detail can be see in "exceptional C++ item31 to item34.

- Prefer writing nonmember nonfriend functions

1. operator =, ->, [], () must be members
2. needs a different type as its left-hand arguent, such as operator <<, use nonmember
3. leftmost argument needs type conversion, use non-member
4. can be implemented using the class public interface alone, use nonmember.

## 7.2 member function

### 7.2.1 Special member functions relationship

#### 7.2.1.1 Basic

- When you write an empty class, compiler will produce at least six member functions. New compiler will produce move ctor and move assignment too.

```

1. class Empty{ };
2.
3. Empty();
4. Empty( const Empty& rhs);
5. Empty& operator=(const empty & rhs);
6. Empty* operator&(){return this;};
7. const Empty* operator&() const;
8. ~Empty();

```

- Why you need to pay attention to these special member functions? Given half a chance, the compiler will write them for you. Another reason is that C++ by default treats classes as value-like types, but not all types are value-like. Know when to write and disable them make you get correct code.
- A good reference is "Everything You ever wanted to know about move semantics" in slideshare.net.
- For these special member functions, main operations can be:

1. Compiler implicitly declare one
  2. Use explicitly declare one
  3. Once you define one, Compiler maybe Not declare another
  4. You can ask compiler declare one
  5. You can ask compiler delete one.
- First question is what "declare" mean?

- The special members can be:
    - not declared
    - implicitly declared
    - user declared
- 
- ```

graph TD
    A[not declared] --> C[deleted]
    B[implicitly declared] --> C
    B --> D[defaulted]
    E[user declared] --> C
    E --> D
    D --> F[user-defined]
  
```

- If you just define a class without any special member function, all six member function will be declared by compiler implicitly.

compiler implicitly declares

|         | default constructor | destructor | copy constructor | copy assignment | move constructor | move assignment |
|---------|---------------------|------------|------------------|-----------------|------------------|-----------------|
| Nothing | defaulted           | defaulted  | defaulted        | defaulted       | defaulted        | defaulted       |

user declares

- “defaulted” can mean “deleted” if the defaulted special member would have to do something illegal, such as call another deleted function.
- Defaulted move members defined as deleted, actually behave as not declared.

- Default can mean "deleted", See an example below:

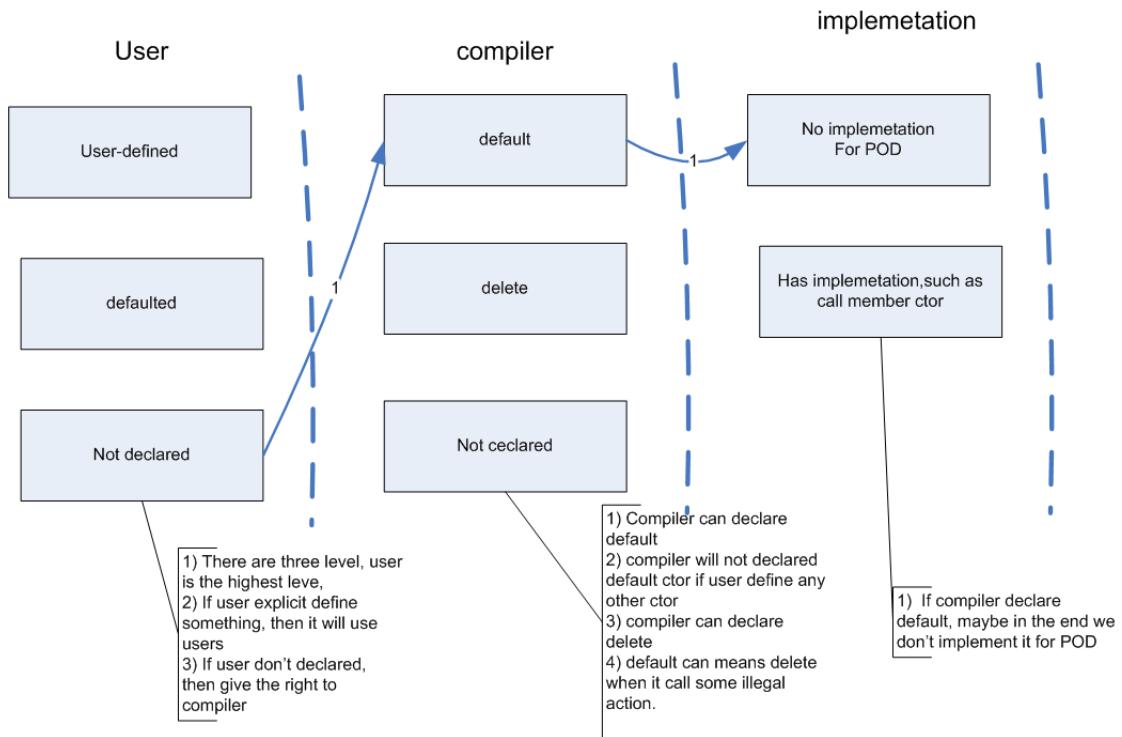
```

1. class A{
2.     A() = delete;
3. };
4.
5. class B{
6.     A a;
7. };
8.
9. int main() {
10.     B b;
11. }
12. //main.cpp:16:7: note: 'B::B()' is implicitly deleted because the
13. //default definition would be ill-formed:
  
```

- Why defaulted move member sometimes "deleted"? Detail can be found in "CWG 1402 is (imho) the most important bug fix to C++11". Another good article is google "Why is the move constructor neither declared nor deleted with clang? " In clang, it support C++11 standard better.

- There are two differences:

1. user define empty and declared default
2. not declare and declare delete



- Next question is what differences are between `=default` and user define empty ctor? problems of below code snippet are:

```

1. struct noncopyable {
2.     noncopyable() {};
3. private:
4.     noncopyable(const noncopyable&);
5.     noncopyable& operator=(const noncopyable&);
6. };

```

1. The copy constructor has to be declared privately to hide it, but because it's declared at all, automatic generation of the default constructor is prevented. You have to explicitly define the default constructor if you want one, even if it does nothing.
2. Even if the explicitly-defined default constructor does nothing, it's considered non-trivial by the compiler. **It's less efficient than an automatically generated default constructor and prevents noncopyable from being a true POD type.**
3. Even though the copy constructor and copy-assignment operator are hidden from outside code, the member functions and friends of noncopyable can still see and call them. If they are declared but not defined, calling them causes a linker error.

4. Although this is a commonly accepted idiom, the intent is not clear unless you understand all of the rules for automatic generation of the special member functions.

- C++11 new keyword default and delete give below advantages:

```

1. struct noncopyable {
2.     noncopyable() =default;
3.     noncopyable( const noncopyable&) =delete;
4.     noncopyable& operator=(const noncopyable&) =delete;
5. };

```

1. Generation of the default constructor is still prevented by declaring the copy constructor, but you can bring it back by explicitly defaulting it.
2. Explicitly defaulted special member functions are still considered trivial, so there is no performance penalty, and noncopyable is not prevented from being a true POD type.
3. The copy constructor and copy-assignment operator are public but deleted. It is a compile-time error to define or call a deleted function.
4. The intent is clear to anyone who understands =default and =delete. You don't have to understand the rules for automatic generation of special member functions.

- Another question is what differences are between =delete and "not declare"

```

struct X
{
    template <class ...Args>
        X(Args&& ...args);

    X() = delete;
};

```

- Deleted members participate in overload resolution.
- Members not-declared do not participate in overload resolution.

### 7.2.1.2 Rules of implicitly declare

- The Default ctor, will not be implicitly generated if:
  1. you have explicitly declared any constructor. Compiler doesn't create a default constructor if we write any constructor even if it is copy constructor.
  2. There is a member in your class that is not default-constructible (such as a reference, a const object, or a class with no or inaccessible default constructor)
  3. (C++11) you have explicitly told the compiler to not generate one using A() = delete;
- The copy ctor, will not be implicitly generated if:
  1. you have explicitly declared a copy constructor (for class X a constructor taking X, X& or const X&)

- 2. there is a member in your class that is not copy-constructible (such as a class with no or inaccessible copy constructor)
- 3. (C++11) you have explicitly told the compiler to not generate one using `A(const A&) = delete;`
- The Copy Assignment Operator will not be implicitly generated if
  1. you have explicitly declared a copy-assignment operator (for class X an operator = taking X, X& or const X& )
  2. there is a member in your class that is not assignable (such as a reference, a const object or a class with no or inaccessible assignment operator)
  3. (C++11) you have explicitly told the compiler to not generate one using `A& operator=(const A&) = delete;`
- The Destructor will not be implicitly generated if
  1. you have explicitly declared a destructor
  2. (C++11) you have explicitly told the compiler to not generate one using `A() = delete;`
- The Move Constructor or Move Operator(C++11) will not be implicitly generated if
  1. you have explicitly declared a move constructor or move assignment(for class X, a constructor taking X&&)
  2. there is a member in your class that cannot be moved (have deleted, inaccessible, or ambiguous)
  3. you have defined a copy assignment operator, copy constructor, destructor, or move assignment operator
  4. you have explicitly told the compiler to not generate one using `A(A&&) = delete;`
- The Move Assignment Operator (C++11) will not be implicitly generated if
  1. you have explicitly declared a move assignment operator (for class X, an operator = taking X&&)
  2. you have defined a copy assignment operator, copy constructor, destructor, or move constructor
  3. you have explicitly told the compiler to not generate one using `A& operator=(A&&) = delete;`
- The justification is that declaring a copy operation (construction or assignment) indicates that the normal approach to copying an object(memberwise copy) isn't appropriate for the class, and compilers figure that if memberwise copy isn't appropriate for the copy operations, memberwise move probably isn't appropriate for the move operations either.
- The same idea as the previous item, declaring a move operation (construction or assignment) in a class causes compilers to disable the copy operations.
- The two copy operations are independent: declaring one doesn't prevent compilers from generating the other. The two move operations are not independent. If you declare either, that prevents compilers from generating the other.

- C++11 deprecates the automatic generation of copy operations for classes declaring copy operations or a destructor. This means that if you have code that depends on the generation of copy operations in classes declaring a destructor or one of the copy operations, you should consider upgrading these classes to eliminate the dependence. Provided the behavior of the compiler-generated functions is correct (i.e., if memberwise copying of the class's non-static data members is what you want), your job is easy, because C++11's "!= default" lets you say that explicitly:

- If you declared a destructor, implicitly defaulted copy member are deprecated.

| compiler implicitly declares |                     |               |                  |                 |                  |                 |
|------------------------------|---------------------|---------------|------------------|-----------------|------------------|-----------------|
|                              | default constructor | destructor    | copy constructor | copy assignment | move constructor | move assignment |
| user declares                | Nothing             | defaulted     | defaulted        | defaulted       | defaulted        | defaulted       |
|                              | Any constructor     | not declared  | defaulted        | defaulted       | defaulted        | defaulted       |
|                              | default constructor | user declared | defaulted        | defaulted       | defaulted        | defaulted       |
|                              | destructor          | defaulted     | user declared    | defaulted       | defaulted        | not declared    |

- A user-declared destructor will inhibit the implicit declaration of the move members.
- The implicitly defaulted copy members are deprecated.
  - If you declare a destructor, declare your copy members too, even though not necessary.

- A summary can be seen blow

| compiler implicitly declares |                     |               |                  |                 |                  |                 |
|------------------------------|---------------------|---------------|------------------|-----------------|------------------|-----------------|
|                              | default constructor | destructor    | copy constructor | copy assignment | move constructor | move assignment |
| user declares                | Nothing             | defaulted     | defaulted        | defaulted       | defaulted        | defaulted       |
|                              | Any constructor     | not declared  | defaulted        | defaulted       | defaulted        | defaulted       |
|                              | default constructor | user declared | defaulted        | defaulted       | defaulted        | defaulted       |
|                              | destructor          | defaulted     | user declared    | defaulted       | defaulted        | not declared    |
|                              | copy constructor    | not declared  | defaulted        | user declared   | defaulted        | not declared    |
|                              | copy assignment     | defaulted     | defaulted        | user declared   | not declared     | not declared    |
|                              | move constructor    | not declared  | defaulted        | deleted         | user declared    | not declared    |
|                              | move assignment     | defaulted     | defaulted        | deleted         | not declared     | user declared   |

### 7.2.1.3 initializer list

- Always use initializer list instead of assignment inside ctor.
- When to use member initialize list?
  1. to non-static const data members.

2. reference member
3. member objects which do not have default constructor: (why I need default ctor can be explained here too)
4. need pass argument to base class ctor

```

1. class A {
2.     int i;
3. public:
4.     A(int);
5. };
6.
7. // Class B is derived from A
8. class B: A {
9. public:
10.    B(int);
11. };
12.
13. B::B(int x) :A(x) { //Initializer list must be used
14.     cout << "B's Constructor called";
15. }

```

5. need to by copy between obj to member obj. (only one copy ctor, more efficient! see below source code.)

<http://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/>

```

1. //method 1:
2. class(string &a, string &b): m_a(a),m_b(b){}
3. // just call string copy ctor,
4. // so you don't need string default ctor
5.
6. //method 2:
7. class(string &a, string &b){ // two action.
8.     m_a = a; //call default constructor to build m_a
9.     m_b = b; // then call assignment operator.
10. }

```

- List member in a initialization list in the order in which they are declared in class. see effective c++ item 13. Order is important.
- Member variables are always initialized in the order they are declared in the class definition. The order in which you write them in the ctor initialization list is ignored. **So you'd better not have one member's initialization depend on other members.**

```

1. class Student{
2.     string m_email; //m_email will be init first, ignore order
3.     string m_first_name; // in the ctor initialization list.
4.     Student(first_name) :m_first_name(first_name),
5.                 m_email(m_first_name+"@gmail"){}

```

- If GetType() is a static member function, or a member function that does not use its this pointer (that is, uses no member data) and does not rely on any side effects of construction (for example, static usage counts), then this is merely poor style, but it will run correctly. Otherwise (mainly, if GetType() is a normal nonstatic member function), we have a problem. **Nonvirtual base classes are initialized in left-to-right order as they are declared**, so ArrayBase

is initialized before Container. Unfortunately, that means we're trying to use a member of the not-yet-initialized Container base subobject.

```

1. template<class T>
2. class Array : private ArrayBase, public Container
3.
4. typedef Array AIType;
5. public:
6.     Array( size_t startingSize = 10 ) : Container( startingSize ) ,
7.             ArrayBase( Container::GetType() ),

```

## 7.2.2 Basic pattern

### 7.2.2.1 constructor and destructor

- Normally, ctor , destructor and assignment should be public. **In inheritance context, all the base class destructor should be virtual.**
- A constructor shall not be virtual or static. If you need something like this, you can look up the virtual constructor idiom. This idiom uses virtual clone() member function (for copy constructing), or a virtual create() member function (for the default constructor).

```

1. class Shape {
2. public:
3.     virtual ~Shape() { } // A virtual destructor
4.     virtual void draw() = 0; // A pure virtual function
5.     virtual void move() = 0;
6.     // ...
7.     virtual Shape* clone() const = 0; // Uses the copy constructor
8.     virtual Shape* create() const = 0; // Uses the default constructor
9. };
10.
11. class Circle : public Shape {
12. public:
13.     Circle* clone() const; // Covariant Return Types; see below
14.     Circle* create() const; // Covariant Return Types; see below
15. };
16. Circle* Circle::clone() const { return new Circle(*this); }
17. Circle* Circle::create() const { return new Circle(); }
18.
19. Shape* s2 = s.clone();
20. Shape* s3 = s.create();
21. // ...
22. delete s2; // You need a virtual destructor here
23. delete s3;

```

- Avoid calling virtual functions in ctor and dtor. Detail can be found in "C++ Coding Standards" item 49.
- **If you define a specific ctor, you also need to define default ctor.** Because system will not produce any default ctor for you. So below statement will produce error when compiling.

```

1. class obj; //error
2. class* obj = new class(); //error
3. class arra[10] //error
4. template<class T>
5. class Array{
6.     T t;

```

```

7. } ;
8.
9. Array<class> a; // error

```

- If no special demand, you can declare your own default ctor and use system implicit generated one. you can use keyword `default`

```

1. class Empty{
2.     Empty() = default;
3.     // you don't need to give implementation of default ctor.
4.     Empty(int i) ;
5. }

```

- Make Constructors Protected to prohibit direct Instantiation. Make constructors Private to prohibit Derivation.
- Use default arguments to reduce the number of ctor.

```

1. class Brush{
2.     Brush();
3.     Brush(Color c);
4.     Brush(Texture t);
5.
6.     Brush(Color c = Black, Texture t = Solid); // it will be better.
7. }

```

- From previous example, you can see that default ctor is very important. but when class MUST need another information when create, such as worker class, You must provide SSN when you create a worker. At this time, if you create default ctor, It's not good idea. A NULL SSN will cause a lot of trouble in the future. So you have to use Worker pointer, and `vector<Worker>`. You also need to use `delete` to disable default ctor. That is C++ spirit, **You never have the best answer, only have context answer.**

```

1. class Worker{
2.     char* SSN;
3.     Worker(const char* );
4.     Worker() {SSN=nullptr}; //bad smell.
5. }

```

- Normally you will have to explicitly declare your own destructor if:
  1. You are declaring a class which is supposed to serve as a base for inheritance involving polymorphism, if you do you'll need a virtual destructor to make sure that the destructor of a Derived class is called upon destroying it through a pointer/reference to Base.
  2. You need to release resources required by the class during its lifetime:
    - (a) Example 1: The class has a handle of a file, this needs to be closed when the object destructs; the destructor is the perfect location.
    - (b) Example 2: The class owns an object with dynamic-storage duration, since the lifetime of the object can potentially live on long after the class instance has been destroyed you'll need to explicitly destroy it in the destructor.
- A copy constructor is called whenever a new variable is created from an object. This happens:
  1. When a new object is initialized to an object of the same class.

2. When an object is passed to a function by value.
3. When a function returns an object by value.
4. When the compiler generates a temporary object.

```

1. 1) Person r(p);    or   Person p = q;           // copy constructor
2. p = q;  // but not called in assignment; p has existed before
3.
4. 2) A value parameter is initialized from its argument.
5. fun(Person r)      fun(p);
6. // copy constructor, produce a new object r.
7. //just like Person r = p;
8.
9. 3) An object is returned by a function.
10. Person fun();
11. Person r = fun();
12. //here copy ctor is called twice.
13. // use -fno-elide-constructors in g++ produce two copy ctor
14. // or It will use Return value optimization.
15.
16. 4) class(string &a, string &b): m_a(a),m_b(b){}
17. //Initialization list

```

- In previous example, you can see when you pass value of obj, It will call copy ctor, It's not very efficient. So you should use reference or pointer if it's possible, don't pass object directly.
- An Assignment operator examples: 1) avoid assignment self 2) return \*this reference.

```

1. class & class::operator=(class &a) {
2.     if(this == &a)
3.         return *this; // avoid assignment
4.         ..... // assignment operation here.
5.     return *this; // return *this reference.
6. }

```

- For reference, once assigned, a reference cannot be re-assigned. So if a class has a reference member, It can be initialized by initializer list in ctor and copy ctor. **But you can't overload assignment operator any more, If you really need assignment operator, change reference to pointer**

### 7.2.2.2 copy and swap idiom

- Any class that manages a resource (a wrapper, like a smart pointer) needs to implement The Big Three. While the goals and implementation of the copy-constructor and destructor are straightforward, the copy-assignment operator is arguably the most nuanced and difficult. How should it be done? What pitfalls need to be avoided?
- Conceptually, it works by using the copy-constructor's functionality to create a local copy of the data, then takes the copied data with a swap function, swapping the old data with the new data. The temporary copy then destructs, taking the old data with it. We are left with a copy of the new data.
- The copy-and-swap idiom is the solution, and elegantly assists the assignment operator in achieving two things: avoiding code duplication, and providing a strong exception guarantee.

- In order to use the copy-and-swap idiom, we need three things: a working copy-constructor, a working destructor (both are the basis of any wrapper, so should be complete anyway), and a swap function.
- We first notice an important choice: the parameter argument is taken by-value. While one could just as easily do the following (and indeed, many naive implementations of the idiom do): Not only that, but this choice is critical in C++11, which is discussed later. On a general note, a remarkably useful guideline is as follows: if you're going to make a copy of something in a function, let the compiler do it in the parameter list.

```

1. dumb_array& operator=(const dumb_array& other) {
2.     dumb_array temp(other); //low efficient, only copy here.
3.     swap(*this, temp);
4.     return *this;
5. }
```

- A simple example code:

```

1. class dumb_array{
2.     //copy ctor
3.     dumb_array(const dumb_array& other)
4.     : mSize(other.mSize),
5.       mArray(mSize ? new int[mSize] : nullptr){
6.         std::copy(other.mArray, other.mArray + mSize, mArray);
7.     }
8.     //copy assignment operator
9.     dumb_array& operator=(dumb_array other){
10.        swap(*this, other); // (2)
11.        return *this;
12.    }
13.
14. private:
15.     std::size_t mSize;
16.     int* mArray;
17. };
18.
19. friend void swap(dumb_array& first, dumb_array& second) {
20.     // enable ADL (not necessary in our case, but good practice)
21.     using std::swap;
22.
23.     // by swapping the members of two objects,
24.     // the two objects are effectively swapped
25.     swap(first.mSize, second.mSize);
26.     swap(first.mArray, second.mArray);
27. }
```

### 7.2.2.3 Big zero, three and five

- The Rule of the Big Three states that if you have implemented either
  1. A destructor
  2. An assignment operator
  3. A copy constructor

You should also implement the other two. To implement the Copy-Swap idiom your resource management class must also implement a swap() function to perform a member-by-member swap. That is called **Rule of big three and half**.

- Big Five is just like big three, but add move copy ctor and move assignment operator.
- The Rule of The Big Four (and a half) states that if you implement one of
  1. The copy constructor
  2. The assignment operator. **Don't need move assignment, in assignment operator, we pass only one kind of type-value.**
  3. The move constructor
  4. The destructor
  5. The swap function(that is half)

then you must have a policy about the others.

```

1. S(S& s) : S{} { swap(*this, s); }
2.
3. S& operator=(S s) { swap(*this, s); }
```

- "The Rule of The Big Four (and a half)" says if you've written one of the above functions then you must have a policy about the others. It doesn't say you have to write them. In fact, you have to have a resource management policy for every class you create. Your policy can be one of the following:
  1. Use the compiler-provided versions of these functions. In other words, you're not doing any resource management in the class.
  2. Write your own copy functions to perform deep copy, but don't provide move semantics.(low efficiency)
  3. Write your own move functions, but don't support copying.(std::unique\_ptr)
  4. Disable copying and move semantics for the class, because it doesn't make sense to allow it.
- Value-like types, such as int or vector<widget>. These represent values, and should naturally be copyable. In C++11, generally you should think of move as an optimization of copy, and so all copyable types should naturally be moveable... moving is just an efficient way of doing a copy in the often-common case that you don't need the original object any more and are just going to destroy it anyway.
- Reference-like types that exist in inheritance hierarchies, such as base classes and classes with virtual or protected member functions. These are normally held by pointer or reference, often a base\* or base&, and so do not provide copy construction to avoid slicing; if you do want to get another object just like an existing one, you usually call a virtual function like clone. These do not need move construction or assignment for two reasons: They're not copyable, and they already have an even more efficient natural "move" operation – you just copy/move the pointer to the object and the object itself doesn't have to move to a new memory location at all.
- Most types fall into one of those two categories, but there are other kinds of types too that are also useful, just rarer. In particular here, types that express unique ownership of a resource, such as std::unique\_ptr, are naturally move-only types, because they are not value-like (it doesn't make sense to copy them) but you do use them directly (not always by pointer or reference) and so want to move objects of this type around from one place to another.
- What does a typical user defined move constructor do?

```

1. class x : public Base{
2.     Member m_;
3.     X(X&& x) : Base( std :: move(x) ), m_( std :: move(x.m_) ){
4.         x.set_to_resourceless_state();
5.     }
6. }
```

- What does a defaulted move assignment do?

```

1. class x : public Base{
2.     Member m_;
3.     X& operator=(X&& x) {
4.         Base::operator=(static_cast<Base&&>(x));
5.         m_ = static_cast<Member&&>(x.m_);
6.         return *this;
7.     }
8. }
```

- Assuming the only non-static data in the class is a std::string, here's the conventional way (i.e., using std::move) to implement the move constructor:

```

1. class Widget {
2.     Widget(Widget&& rhs)
3.     : s(std :: move(rhs.s)){ ++moveCtorCalls; }
4. private:
5.     static std :: size_t moveCtorCalls;
6.     std :: string s;
7. };
```

### 7.2.3 operator overload

- Never overload `&&`, `||` and `comma` in C++. Just remember it!
- You can declare `operator` as member function. if it's not member function, declare it as a friend function.
- If you want to overload `+` operator, and support time `t`; `3+t`; You need to define a friend function. In order to improve a `t+3` efficiency, you also can define a member function `time operator+(int i)` member function.

```

1. time operator+(const time &t) const;
2. //member function.
3. // t = t1+t2
4.
5. friend time operator+(int, const time &t)
6. nonmember friend function
7. // 3+t
8.
9. time operator+(int i)
10. member function
11. // t+3,
12. //to avoid implicit conversion from 3 to obj.
```

- For Binary Arithmetic Operators `+` `-` `*`, implement Compound Assignment Operators `+=` `-=` `*=` first, then use these Compound Assignment operators

- Use `+=` implememnt operator `+`. It's very good design. It provides two advantages. 1) give another function, 2) avoid code duplication.
- Compound assignment operators should be overloaded as member functions, as they change the left-hand operand. Like all other operators (except basic assignment), compound assignment operators must be explicitly defined, they will not be automatically. A code example.

```

1. Vector2D& Vector2D::operator+=(const Vector2D& right) {
2.     this->x += right.x;
3.     this->y += right.y;
4.     return *this;
5. }
```

```

1. Foo operator+(const Foo& lhs, const Foo& rhs) {
2.     Foo result = lhs;
3.     result += rhs;
4.     return result;
5. }
```

- Why `<<` need to be friend? You need to consider: Because you need to write `cout << obj`. If you declare `<<` as a member operator, you have to write `obj << cout`; it looks weird.
- `ob = ob1+ob2` will changed to: `ob1.operator+(ob2)`; The last `const` make the `ob` to invoke this operator doesn't change the value in this class. Don't return `const` value type, It will make "move" not work.
- assignment operator overload. 1) return a non-`const` reference, and 2) to avoid assign self. Don't return `const` reference, somebody said that it can avoid  $(x=y)=z$ . But in fact, it's not a normal way to write such code. In STL string, assignment operator just return reference. It give you a clue, **any time you have questions about interface design, you can see the STL library.**

```

1. class A & operator=(const class A& rhs) {
2.     if(this ==&rhs) return *this;
3.
4.     //Do some things other
5.     return *this;
6. }
```

## 7.3 inheriantce

- OO just kick in when you use reference and pointer to base class

```

1. class A{
2.     public:
3.         virtual void fun() { cout<<"A"; }
4. };
5.
6. class B: public A{
7.     public:
8.         void fun() { cout<<"B"; }
9. };
10.
11. int main() {
12.     B b;
13.     A a = b;
14.     a.fun(); //output A
```

```

15.     A& ra = b;
16.     ra.fun(); // output B
17.     A* pa = &b;
18.     pa->fun(); // output B
19.     (*pa).fun(); // output B
20.
21.     A* pa1 = new B;
22.     (*pa1).fun(); // output B
23.
24. }
```

### 7.3.1 special member functions in inheritance

#### 7.3.1.1 ctor

- Subclass ctor will always call base class ctor.

1. Constructor should NOT be virtual
2. If subclass doesn't define any ctor, compiler will implicitly define a default ctor, and this ctor will call base class default ctor.
3. If subclass has a ctor, but it doesn't explicitly call base specify ctor, ctor of subclass will call base default ctor.(without any parameter.) If base ctor only has specify ctor, no default ctor, produce compiler error.
4. If you want explicitly call base specify ctor, use initialization list syntax.

```

1. class base{
2. public:
3.     base(); // default ctor
4.     base(int b); // specify ctor
5. private:
6.     int b;
7. };
8.
9. class subclass: public base{
10.    subclass(); // default ctor
11.    subclass(int s, int b); // specify ctor1
12.    subclass(int s); // specify ctor2
13.    int s;
14. };
15.
16. subclass::subclass(int s, int b): base(b){
17.     m_s = s;
18. }
19. /////////////////////////////////
20. subclass sc1(2,3);
21. // call specify ctor1, then
22. // explicit call base specify ctor
23.
24. subclass sc2(2); // implicit call base default ctor
25. subclass sc3; // implicit call base default ctor
26. // if base has not default ctor,
27. // sc2 and sc3 will produce error.
28. }
```

- If you don't use explicitly initialization list syntax to call base specify ctor, You will get uninitialized value or you can't initialize base member, Neither are good.

```

1. //method 2:
2. DeriveClass::DeriveClass(int base_a, b){
3.     //it will call base class default constructor.
4.     //in this case, base_a is not assigned at all
5.     Derive_b = b
6. }
7. /////////////////////
8. //method 3:
9. DeriveClass::DeriveClass(int base_a, b) {
10.    base_a = a      //It can be thought as a bad design.
11.    //You can't access private base member data.
12.    //base_a need to be public member data,
13.    Derive_b = b
14. }
```

- Idea behind rules: Do best to make sure a obj can be built.

### 7.3.1.2 destructor

- For destructor:
  - If a base class has a destuctor, but sub class doesn't have, compiler will produce an implicit default destructor, and this implicit default destructor will call base class destuctor.
  - If you define a subclass destructor, It will call base destructor automatically, you don't need to call it explicitly.
  - The question is, How can you make sure you sub class destructor will be called if you use a base class pointer or reference, answer is below:
- Don't call base desctructor explicitly, It will called automatically in the reverse order of construction. And you should give a base desctructor a definition, Or linker will report error it even you don't call it in your source code.
- Make base class destructor public and virtual (polymorphic deletion by base class pointer or reference), or proteced and nonvirtual, base classes need not always allow polymorphic deletion. For example, consider class templates such as std::unary\_function. This time, you should make destructor protected and nonvirtual.

```

1. template <class Arg, class Result>
2. struct unary_function{
3.     typedef Arg argument_type;
4.     typedef Result result_type;
5. };
6.
7. Illegal code that you can assume will never exist.
8. void f( std::unary_function* f ){
9.     delete f; // error, illegal
10. }
```

- If base class still need to build itself, You can change it back public virtual, Even without polymorphic deletion by now, you still need to declare it as virtual for the future safety(Even with a little dynamic-binding efficiency penalty. ) At same time, If a base class is not abstract class, usually, it's BAD design.
- Never throw exception from dtor, if exception A is thrown, then stack-unwinding, when a obj is destracted, then dtor is called, when another exception B is thrown by the dtor, application will call terminat function immediately. If you have exception, catch it inside of the dtor.

### 7.3.1.3 copy ctor in inheritance

- copy ctor(assignment ctor) in inheritance will cause slicing problem. and slicing. Below four are all SLICING. number 5 and number 4 is not very obvious. They are call base class ctor. No matter what you input a reference to a derived class or not.

```

1. class Base{};
2. class Derived1 : Base{};
3. class Derived2 : Base{};
4. Derived1 d1;
5. Derived1 d2;
6. /////////////////
7. Base b = d1; //1)
8.
9. Fun(Base b);
10. Fun(d1); //2)
11.
12. Base& Bref = d1;
13. Fun(Bref) //3)
14.
15. Base* bp = new B(Bref); //4)
16.
17. Base* bp1 = new Derived1(); //5) sibling slicing
18. base* bp2 = new Derived2();
19. *bp1 = *bp2; // bp1 just copy Base part in Derived2.
20. //so bp1 now is MIXTURE of d1 and d2.

```

- Think a problem as below: how to make deep copy and avoid slicing in base class copy ctor?

```

1. class Base{};
2. class Derived1 : Base{};
3. class Derived2 : Base{};
4. Derived d1;
5. Derived d2;
6.
7. Base* Copy(Base& Bref){
8.     //How to avoid slicing and make deep copy.
9. }
10.
11. Base& Bref = d1
12. Base* d1p = Copy(Bref)
13.
14. Base& Bref = d2
15. Base* d2p = Copy(Bref)

```

- Continue- Think this problem: error method

```

1. Base* Copy(Base& Bref){
2.     Base* p = new Base(Bref)
3.     //Slicing happen. bad
4. }

```

- Continue- Think this problem: TypeID method.

```

1. Base* Copy(Base& Bref){
2.     //Use tyid and dynamic_cast
3.     //involve a lot of if and switch about type.
4.     //anytime if you use dynamic_cast and if,
5.     //you can think about virtual function
6. }

```

- Continue- Think this problem: Virtual Clone method. A function's return type is never considered part of its signature. You can override a member function with any return type as long as the return type could be used wherever the base class return type could be used.

```

1. class Base{
2.     virtual Base* Clone() = 0;
3. };
4.
5. class Derived1 : Base{
6.     virtual Derived1* Clone(){return new Derived1(*this);}
7. }
8.
9. Base* Copy(Base& Bref){
10.    Base* p = Bref.Clone();
11. }
```

- Continue- Think this problem: Change Design. Base is concrete class, More Effective C++ Item 33 said "Making Non-leaf class abstract. So maybe you can change the inheritance system.

- **Assignment operator and copy ctor in inheritance summary:**

1. Default Assignment operator and copy constructor in derived class which are implicitly produced by compiler will call default base assignment operator and copy constructor.
2. If derived class has no new operation. Don't need to define derived class Assignment operator and copy constructor, implicit one will call base one automatically
3. If derived class has new operation. You have to define derived class Assignment operator and copy constructor, it will not invoke assignment operator and copy constructor in base class any more. Inside, manually invoke base class Assignment operator and copy ctor Detail can be found in C++ primer p760. Syntax looks like below: see effective C++ Item 16.
4. For copy ctor, just init list syntax. For assignment operator, use two different methods depends on if base class declare its own assignment operator(). Source code is below:

```

1. DerivedClass::DerivedClass(const DerivedClass &dc) : \
2.     BaseClass(dc) {...} //init list syntax here.
3.
4. DerivedClass & DerivedClass::operator=(const DerivedClass &dc) {
5.     BaseClass::operator=(dc);
6.     // base class declare explicitly operator
7.
8.     ( (BaseClass&) *this ) = dc
9.     //base class no explicitly operator
10.    // change *this to BaseClass reference,
11.    // if you change to BaseClass, It will call copy ctor.
12. }
```

- There are three articles, you should read them together.

<https://herbsutter.com/2013/05/09/gotw-1-solution/>

<https://stackoverflow.com/questions/21825933/any-difference-between-copy-list-initialization-and-traditional-copy-initializat>

<https://stackoverflow.com/questions/1051379/is-there-a-difference-between-copy-initialization-and-direct-initialization>

### 7.3.2 virtual function and override

- Only virtual function come into vtbl. Friend can't be virtual function, because it's not a member of class.
- When a method is declared virtual in a base class, it is automatically virtual in the derived class, but it is a good idea to explicitly declare it by using the keyword virtual in the derived class declarations too.
- **Almost all the base class has virtual function, if a class doesn't contain a virtual function, It is an indication that it is not meant to be used as a base class**
- Don't rewrite non-virtual base member function, see effective c++
- For overriding to occur, several requirements must be met:
  1. The base class function must be virtual.
  2. The base and derived function names must be identical (except in the case of destructors).
  3. The parameter types of the base and derived functions must be identical.
  4. The constness of the base and derived functions must be identical.
  5. The return types and exception specifications of the base and derived functions must be compatible.
  6. To these constraints, which were also part of C++98, C++11 adds one more: The functions' reference qualifiers must be identical.

- reference qualifier explain:

```

1. class Widget {
2. public:
3.     void doWork() &; // this version of doWork applies
4.     // only when *this is an lvalue
5.     void doWork() &&; // this version of doWork applies
6. }; // only when *this is an rvalue
7.
8. Widget makeWidget(); // factory function (returns rvalue)
9. Widget w; // normal object (an lvalue)
10.
11. w.doWork();
12. // calls Widget::doWork for lvalues (i.e., Widget::doWork &)
13. makeWidget().doWork();
14. // calls Widget::doWork for rvalues (i.e., Widget::doWork &&)

```

- Any small error will not real override base virtual function. but creating a new virtual method with a different signature. such as examples below.

```

1. class Base {
2. public:
3.     virtual void mf1() const;
4.     virtual void mf2(int x);
5.     virtual void mf3() &;
6.     void mf4() const;
7. };
8.
9. class Derived: public Base {
10. public:
11.     virtual void mf1();

```

```

12.     virtual void mf2(unsigned int x);
13.     virtual void mf3() &&;
14.     void mf4() const;
15. };

```

- override specifier should be used in derived class member function used to check if they are match with member function in base class.

```

1. struct A{
2.     virtual void foo();
3.     void bar();
4. };
5.
6. struct B : A{
7.     void foo() override; // OK: B::foo overrides A::foo
8.     void bar() override; // Error: A::bar is not virtual
9. };

```

- Use final in Base class, to stop sub class override.

```

1. class Base{
2.     virtual void method1() final;
3. };

```

## 7.4 Classes relationship

### 7.4.1 structure semantic

#### 7.4.1.1 Definition

- OOP has four relationships:

**Composition** exists when a member of a class has a part-of relationship with the class. In a composition relationship, the class manages the existence of the members. To qualify as a composition, an object and a part must have the following relationship:

1. The part (member) is part of the object (class)
2. The part (member) can only belong to one object (class) at a time
3. The part (member) has its existence managed by the object (class)
4. The part (member) does not know about the existence of the object (class)

More descriptions:

1. Compositions are typically implemented via normal member variables, or by pointers where the class manages all the memory allocation and deallocation. If you can implement a class as a composition, you should implement a class as a composition.
2. Ownership, same life time, not change in the middle. Person and Head

**Aggregations** exists when a class has a has-a relationship with the member. In an aggregation relationship, the class does not manage the existence of the members. To qualify as an aggregation, an object and its parts must have the following relationship:

1. The part (member) is part of the object (class)
2. The part (member) can belong to more than one object (class) at a time
3. The part (member) does not have its existence managed by the object (class)

4. The part (member) does not know about the existence of the object (class)

More descriptions:

1. Aggregations are typically implemented via pointer or reference.
2. Ownership, maybe same life time, may change in the middle, Container and pointer(same life time, changeable). Airport and airplane and department and teacher.

Aggregations are typically implemented via pointer or reference.

**Associations** are a looser type of relationship, where the class uses-an otherwise unrelated object. To qualify as an association, an object and an associated object must have the following relationship:

1. The associated object (member) is otherwise unrelated to the object (class)
2. The associated object (member) can belong to more than one object (class) at a time
3. The associated object (member) does not have its existence managed by the object (class)
4. The associated object (member) may or may not know about the existence of the object (class)

More descriptions:

1. Associations may be implemented via pointer or reference, or by a more indirect means (such as holding the index or key of the associated object). No Ownership, different life time,
2. People and toothbrush(same life time, changeable) Teacher and student( not same life time, changeable), person and father(Not Nullity), person and wife( Nullity)

**dependency** exists when a member of a class has a part-of relationship with the class. In a composition relationship, the class manages the existence of the members. To qualify as a composition, an object and a part must have the following relationship:

More descriptions:

1. No Ownership, Not Includes as a member, person and friend.
2. Dependency definition: If class X's member function argument is class Y, X is dependency of Y.
3. Dependency definition extention: For a class X, all functions, including free functions, that both "Mention" X and "supplied with" X are logically part of X, because they form part of the interface of X. Supplied with means that they appear in the same header file.

- Summary:

| Relationship type                    | Composition<br>Whole/part | Aggregation<br>Whole/part | association<br>otherwise unrelated | Dependency<br>otherwise unrelated |
|--------------------------------------|---------------------------|---------------------------|------------------------------------|-----------------------------------|
| Members can belongs to multi classes | no                        | yes                       | yes                                | yes                               |
| member existence managed by class    | Yes                       | no                        | no                                 | no                                |
| Directionality                       | uni                       | uni                       | uni or bidirectional               | uni                               |
| relationship verb                    | part-of                   | has-a                     | uses-a                             | depends-on                        |

- From Structure perspective, toothbrush and people are association relationship. Although you can say people has a brush, but you can't say brush is part of people. But pay attention, this kind of definition is trick and vague, **sodon't treat it pedantically**. See some examples:
  - Animal class contain name(string):composition.
  - classroom and students: aggregation, but associate if class room is shared. it can be changed in the specific context.
- Composition is specific aggregation, and aggregation is specific associate.**

## 7.4.2 Inheritance semantic

### 7.4.2.1 private inheritance

- Code and example and explanation:

```

1. class B { /* ... */ };
2. class D_priv : private B { /* ... */ };
3. class D_prot : protected B { /* ... */ };
4. class D_publ : public B { /* ... */ };
5. class UserClass { B b; /* ... */ };

```

- None of the derived classes can access anything that is private in B.
  - In D\_priv, the public and protected parts of B are private.
  - In D\_prot, the public and protected parts of B are protected. Protect used in three generation inheritance. By protected Inheritance, grandfather member become protected member in father, so Grandson can still use grandfather's members.
  - In D\_publ, the public parts of B are public and the protected parts of B are protected (D\_publ is-a-kind-of-a B).
  - Class UserClass can access only the public parts of B, which "seals off" UserClass from B.
- To make a public member of B public in D\_priv or D\_prot, state the name of the member with a B:: prefix. E.g., to make member B::f(int,float) public in D\_prot, you would say:

```

1. class D_prot : protected B {
2. public:
3.   using B::f; // Note: Not using B::f(int, float)
4. };

```

- In short, composite uses object names to invoke a method, whereas private Inheritance uses the class name and scope resolution operator Instead. If you need the base class itself, use a type case (const string&) \* this; detail can be seen in C++ primer, P800
- If you want to reuse string code in class student(also means **has-a** relationship, student has a string name) you have two options, The first is composition, the second is private inheritance. See below:

```

1. class Student{ // 1) use composition.
2.   int getNameLen();
3. private:
4.   string m_name;
5.   vector<double> score;
6. };
7.

```

```

8. int Student :: getNameLen() {
9.     return m_name.length();
10. }
```

```

1. class Student: private string, private std::vector<double>{
2.     int getNameLen();
3.     // We don't need private member data any more
4. }
5.
6. int Student :: getNameLen() {
7.     return string::length();
8.     // use class name and scope-resolution operator
9. }
10.
11. const string& Student :: getName() {
12.     return (const string&) *this;
13. }
```

#### 7.4.2.2 "Has-A" relationship

- You can use **1)private inheritance, 2)composition and 3)template** three methods to describe has-a relationship.
- Private and protect inheritances are used to implement has-a relationship. Introduce their access policy in inheritance:
- **Prefer composition, use private inheritance when you have to.** Reusing by private inheritance is weird in syntax and difficult to understand. One exception is you have to access string or vector<T> protect member function, at this time, you MUST use private inheritance.
- By now, I want to not only reuse class A, I also want to make a class adaptable. or make it possible to select different algorithms from the outside. An example is to replace brakes of a car (at runtime) Intend to pass Car around to non-template functions There are three options.
  1. Version 1: Abstract base class, In fact, consider **same life time and composition relationship**, Using Brake obj directly is better. But you want to have runtime polymorphism at the same time, So have to use pointer or reference. Here you can use smart pointer(uniqu\_ptr).

```

1. class Brake {
2.     public: virtual void stopCar() = 0;
3. };
4.
5. class BrakeWithABS : public Brake {
6.     public: void stopCar() { ... }
7. };
8.
9. class Car {
10.     Brake* _brake;
11.     public:
12.         Car(Brake* brake) : _brake(brake) { brake->stopCar(); }
13. }
```

2. Version 2a: Template

```

1. template<class Brake>
2. class Car {
```

```

3.     Brake brake;
4.     public:
5.     Car () { brake.stopCar(); }
6.   };

```

### 3. Version 2b: Template and private inheritance

```

1. template<class Brake>
2. class Car : private Brake {
3.   using Brake::stopCar;
4.   public:
5.     Car () { Brake::stopCar(); }
6.   };

```

- I would generally prefer version 1 using the **runtime polymorphism**, because it is still flexible, and All Car have the same type. In template implementation, Car<Opel> is another type than Car<Nissan>. If your goals are great performance while using the brakes frequently, i recommend you to use the templated approach**static binding**. By the way, this is called **policy based design**.
- Another example about policy based design is std::map, you can input a functor type, use to compare two elements inside map. At this time, because your host(std::map) is template, you have to use policy based design.

#### 7.4.2.3 "Is-A" relationship

- some basic syntactic knowledge:
  1. No source code, just header and lib, you also can use inheritance in C++ language.
  2. Pure virtual class can't be instance, it just a abstract interface, it's agreement.
  3. Both reference and pointer support polymorphism.
- Inheritance has three level knowledge.
  1. The basic design knowledge, manager is a person, apple is a fruit, and bla bla bla.
  2. The basic pure virtual, virtual, and non virtual syntax knowledge.
  3. **Isolate change between Client and Implement**. That is the highest level in design pattern. How to understand interface? Client<->Interface<->Implement. through Interface, you keep all the change happen implement side, not affect Client at all.
- **Public inheritance is substitutability, not to reuse, but to be reused.** For example, client use class A, and class B inherit class A. I don't think class B reuse something in class A. we should think that class B could be used by client too, just like client use class A.
- A common mistake is inheriting from classes that were not designed to be base class. For example, you want to customize some current class, you have string class, but you want to change some behavior of string. So you write: `class myString : public string`. But it violate LSP( Liskov Substitution Principle): **Functions That use pointers or references to base classes must be able to use objects of derived classes without knowing it.**
  1. client use string class through base class pointer or reference
  2. base class has virtual function.
  3. derived class redefine base class virtual function.

- See previous three points: all requirement are not satisfied. Clients most times use string by value directly, string doesn't has any virtual function, it's a value class, and you didn't design before hand.
- If you want to change find function, You don't need inherit, Just write non-member function and pass a string object.

```

1. myFind(const string& str){
2.   //you customized behavior.
3. }
4. // use myFind(str) and myStr.find()
5. //I think only difference is syntax.

```

- Even you have specific state, you also can use composition to avoid inheritance.

```

1. class MyString{
2.   size_t length(){return str.length}
3.   //just need to write forward function here;
4.
5.   myFind(){
6.     //You customized behavior.
7.   }
8.
9.   std::string str;
10. }

```

- **Another mistake is public inheritance is work-like-a, not is-a.** For example, circle is a eclipse, and square is a rectangle, but in oop, you can't make circle inherit from eclipse, because, eclipse has two centers. you can't make square inherit from rectangle, because setWidth is not work as the same as in rectangle, (setWidth in square will change height at the same time.) It break LSP. A solution is make an abstract base class(ABC) then make circle and ellipse inherit from ABC
- Previous example also explain, when you design a class, it doesn't need to be a practical object, such as animal, car, people, etc. It can be abstract conception. (So desgin pattern is so important conception).
- Composite VS Inheritance:

1. Does TypeB want to expose the complete interface (all public methods no less) of TypeA such that TypeB can be used where TypeA is expected? Indicates Inheritance. e.g. A Cessna biplane will expose the complete interface of an airplane, if not more. So that makes it fit to derive from Airplane.
2. Does TypeB only want only some/part of the behavior exposed by TypeA? Indicates need for Composition. In this case, it makes sense to extract it out as an interface / class / both and make it a member of both classes.
3. Inheritance must pass Liskov Substitution Principle. Previous example about ellipse and circle fail this test. because ellipse has setLongAxis() and setShortAxis(), but circle doesn't have them at all.

```

1. class Shape{
2.   virtual draw()=0 // you have to re
3. }
4.
5. class Circle: public Shape{
6. }

```

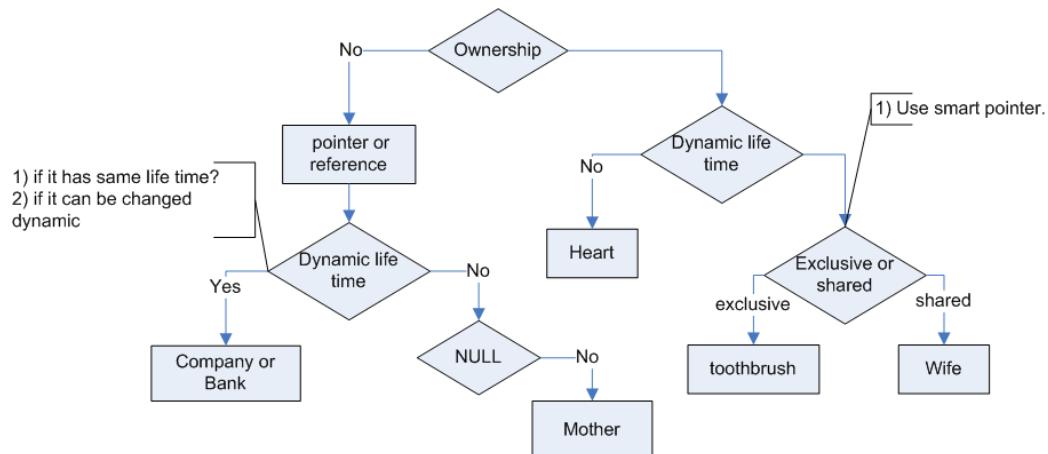
### 7.4.3 Ownership semantic

- Ownership semantic is different with structure semantic. Structure semantic care about the whole-part relationship. Ownership semantic.
- There are two kinds of Ownership: exclusive ownership and shared ownership. Usually:
  1. Composition has exclusive ownership.
  2. Aggregation has shared ownership.
  3. Ownership semantic is different with structure semantic, Associate can have ownership, such as people and his brush. people and his wife.
- Ownership semantic has two children policies: life time policy and Nonnullity policy.
- Ownership may describe the relationship between two classes or class and its member.
- Code example and explanation.

```

1. class Man{
2.     Heart t_           // 
3.     unique_ptr<Brush> pb_; // 
4.     shared_ptr<Woman> pwife_; // 
5.     Company* pb_; // 
6.     Woman& mother; // 
7.     void getMoney(const Bank&); 
8. }
```

#### 1. Whole part



**Company and Bank** No ownership. Company is a member, It's important for you to have a work and it is used by many member functions. so it's a member. For Bank, you just use it when you want to get Money, so use it in the function as parameter. Company is associate relationship and Bank is dependency relationship.

**Monther** No ownership. It can not be null, and can not be changed. so we use reference.

**Heart** Ownership, same life time, (No dynamic life)

**toothbrush** Ownership, dynamic life, and exclusive ownership, so we use unique\_ptr

**wife** Ownership, dynamic life, and shared ownership, so we use shared\_ptr

#### 7.4.4 Summary

##### 7.4.4.1 Examples

- OOP example 1: If it's a people-head(Composition) relationship, and it's compiler-given(not dynamic), just use **member obj**. Pay attention, If Engine has no default ctor, you have to use initialization list in car ctor. initialization list can be used to call base ctor.

```

1. class Car{
2.     Engine eng; // member obj, not use pointer here.
3.     Car(int carArg, int engArg): eng(engArg){}
4. }
```

- OOP example 2:

- OOP example 2: If it's an peole-brush(Association) relationship, If has same life time and exclusive ownership, use **unique\_ptr**. If you want to change, use **unique\_ptr** reset function or move semantic from another **unique\_ptr**. TootuBrush example, use **unique\_ptr**.

```

1. class Person{
2.     unique_ptr<Brush> unpbrush;
3.
4.     buyNewBrush(string &name){
5.         unpbrush.reset(new Brush());
6.         // you can't use unpbrush = new Brush()
7.         // unpbrush assignment only support( unique_ptr<T> && );
8.         // use reset, it will make original deleted automatically
9.     }
10.
11.    // don't need destructor any more.
12. }
```

- OOP example 3: if it's wife-husband(Association) relationship, If you want to express **Strong Not Nullity**, use reference, such as Mother-Son association, You need to use initialization list to init mother. If Nullity, such as wife, just use raw pointer, **weak\_ptr** or **shared\_ptr**. **Because no ownership involved, don't use unique\_ptr at all.**

- OOP example 3-1: What's different with raw pointer, **weak\_ptr** or **shared\_ptr**?

```

1. class Man{
2.     Woman* wife; // can be set to nullptr.
3.     // maybe change or live longer than you.
4.
5.     weak_ptr< Woman &mother>; // must give a mother in initial list
6. }
```

- OOP example 4: if it's Dependency, such as friend relationship, Most of time, we just use pointer or reference as function parameter.

```

1. class Man{
2.     lendMoney(Friend* mike);
3. }
4. // just use it in function. not a member of class.
```

- OOP example 5: Suppose Man and Computer(Associate from structure semantic) is has-a relationship(from inheritance semantic) and same life time. Don't use pointer or reference, just copy from a common computer, and maybe later you can customize your computer, and it will not

effect common one. We can copy from commonComputer and init all Worker object. And this time use initialization list can improve efficiency.

```

1. class Worker{
2.     Computer m_desktop;
3.     Worker (Computer u): m_desktop(u){}
4. }
5.
6. Computer commonComputer;
7. Worker Yan(commonComputer);
8. Worker Han(commonComputer);

```

- OOP example6: Change a semantic, Unit may have a Bus, but **owner policy** tell us that bus can be shared by different unit. and **life time policy** tell us that bus and unit has separate life time. so here, we should use shared\_ptr.

```

1. class Unit{
2.     shared_ptr<Bus> shr_p_bus;
3. }

```

- OOP example7: One class provides a container to hold multiple objects of another type. A value container is a composition that stores copies of the objects it is holding. A reference container is an aggregation that stores pointers or references to objects that live outside the container.

## 7.5 design pattern

### 7.5.1 Common used principle

#### 7.5.1.1 Three big rules

- There are three basic rules.
  1. Liskov Substitution Principle—Clients (Functions and Class) that use pointers or references to base classes must be able to use objects of derived classes without knowing it. So all overrides of virtual member functions must **require less and provide more**. So you can make substitution successfully.
  2. Dependence Inversion Principle. Clients only depends on interface.
  3. Interface segregation principle. Make Interface simple and small.
- Example, see code below. 1)LSP: Car can use GasPower or ElePower without know it. 2)DIP: Car only is depends on Power\* interface 3)ISP: Interface should be small and separately. An example can be seen in MI section below.
- There are three points:
  1. Pointer or reference to the base class (Power\* pow)
  2. Virtual function support dynamic-binding
  3. You need to design the base class and it should include at least one virtual function.

```

1. class Car{
2.     .....
3.     Power* pow
4. };
5.

```

```

6. Car :: start () {
7. .....
8. pow->ignite () ;
9. .....
10. }
11.
12. class Power{
13.     virtual ignite () ;
14. };
15.
16. class GasPower : public Power
17. class ElePower : public Power
18.
19. Car car(gas);
20. car.start ();

```

### 7.5.1.2 NFA and NVI

- Inheritance three usage: 1) I want to inherit a interface. (pure virtual, you have to rewrite ) 2) I want to inherit a implement, but I want to change it. (virtual, you may or maynot rewrite) 3) I want to inherit a implement, but I don't want to change it. (Non-virtual, you can't rewrite it at all)

```

1. class shape{
2. public:
3.     virtual draw () = 0 // you have to re
4.     virtual error () ; // there is default implement .
5.     // but you may change it .
6.     int objectID () ; // you don't need to rewrite it .
7. }

```

- NFA can be found in More effective 33 "making non-leaf classes abstract" and C++ coding standards 36 "Prefer providing abstract interfaces." They all said that you should only inherit from an abstract interfaces. It also follow DIP.
- When you found abstract conception appear in more than one context, you need build abstract interface for it.
- There are some advantages. use interface will separate client and implement, make addition and modification implement easier. (But it need good design at the beginning). If you inherit from concrete base class, It will cause Slicing problem, Detail can be seen in "ctor in inheritance" section.
- NVI is making virtual functions nonpublic, and public function nonvirtual. This is similar with Template Method design pattern.
- About NVI, more detail can be found in "C++ coding standards Item 39" and "Virtuality herb sutter"
- A design comparison:

### 7.5.2 Resource wrapper and RAI

- Whenever you deal with a resource that needs paired acquire/release function call, encapsulate that resource in an object. Such as: fopen/fclose, lock/unlock, and new/delete.

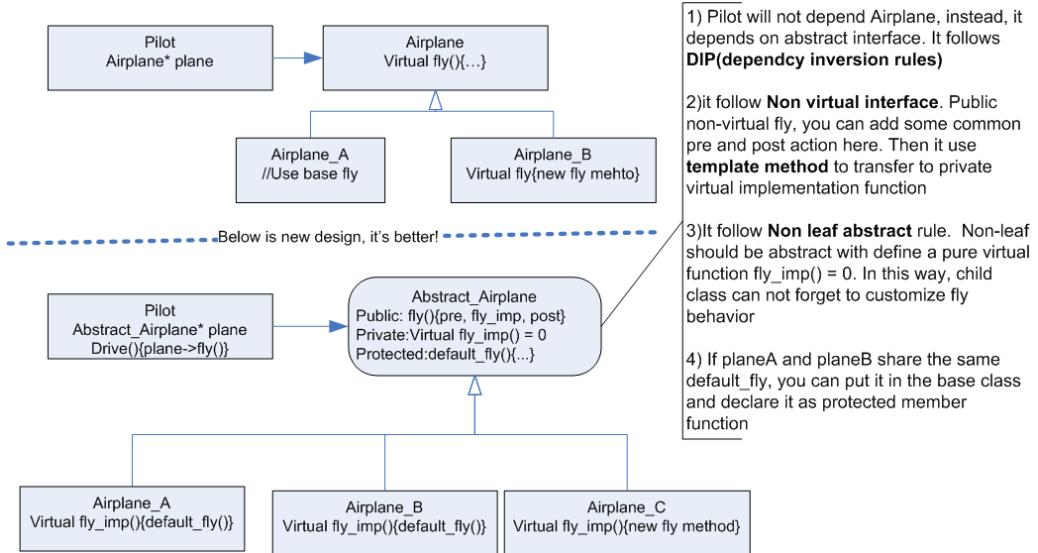


Figure 7.1:

- When implementing RAII, be conscious of copy construction and assignment. the compiler-generated version probably won't be correct. If it's not copyable, use `=delete` , if it's copyable, duplicate the resource. You also can use smart\_pointer in this scenario too.
- The basic idea of RAII is to represent a resource by a local object, so that the local object's destructor will release the resource. That is to say: To prevent resource leaks, use RAII objects that acquire resources in their constructors and release them in their destructors.

```

1. //C version ,
2. File* fp = fopen("/path/to/file");
3. // throw exception here , then resource leaking
4. fclose(fp);
5.
6. //Java version
7. try {
8.     File file = new File("/path/to/file");
9.     // throw exception here , go to finally .
10. } finally {
11.     file.close();
12. }
13.
14. //c++ version
15. fun{
16.     fstream if("path/to/file")
17.     if.getline
18.     // you don't need to if.close().
19. }
20.
21. //c++ smart version .
22. std::unique_ptr<FILE>           // the wrapped raw pointer type: FILE*
23. int(*)(FILE*)>                // the custom deleter type: fclose() prototype
24. myFile( fopen("myfile", "rb") , // resource (FILE*) is returned by fopen()
25. fclose );                      // the deleter function: fclose()

```

- Another good example RAII is `unique_ptr`. The idea of smart pointer is putting `*p` into a local pointer-like object, then when it goes out of scope or unwind-stack when exception is thrown, It will

call destructor, then delete p.

- We should consider resource generically, pointer \*p pointed to a new object is resource, A handle to a file is a resource to. **We wrap handle to a file into ifstream, and wrap pointer \*p into smart\_pointer**
- Just like return value, Exception will skip all the statement below the throw, In C++, It doesn't support finally statement sometimes. At this time, we need to use RAII.

```

1. Int *p = new int;
2. string a    //a is ok, a will be destructed properly
3. //due to the C++ unwind stack.
4. throw exception.
5. delete p; // this will not run.

```

- For this problem, you should use smart pointer to declare a auto object. If you use string object, It's ok. So in previous example, you can use smart pointer, it will help you to avoid memory leakage problem.

```

1. unique_ptr<int> aupr (new int(100));
2. string a    //a is ok, a will be destructed properly
3. //due to the C++ unwind stack.
4. throw exception.
5. // both a and aupr will call their own destructor function .

```

- In you ctor, If you use new and new failed and throw a exception, the destructor will not be called. You can use auto\_ptr as member data and use init list to initialzie it. see more effective C++ exception chapter.
- Don't use C FILE\* and char [] as string. Use iofile class and string object, because they are exception safe

1. Any time when you use new, consider if there are c++ container or object.
  2. If not, use smart\_pointer.
- Another example is when you make program based on Win API.

```

1. class module {
2. public:
3.     explicit module(std::wstring const& name)
4.     : handle{::LoadLibrary(name.c_str())} {}
5.
6.     ~module{
7.         ::FreeLibrary();
8.     }
9. private:
10.    HMODULE handle;
11. };

```

- There are three RAII implementation instances in your practical programming:
  1. Use auto member; You have to keep m\_str and vc are RAII. In this way, you don't need to build dtor manually.

```

1. class RAII {
2.     private:
3.         string m_str;
4.         vector<int> vc;
5. };

```

2. Use pointer and handle; In this way, You have to use pointer, Maybe you need some customized action in runtime , **Use handle is only method to use this resource** or any other reason. And this time, you have to write your own dtor.

```

1. class RAII {
2.     private:
3.         string* m_str;
4.         vector<int*> vc;
5. };

```

3. Use smart point wrap pointer and handle; When you wrap handle, you can custom this delete behavior. See source code below:

```

1. class RAII {
2.     private:
3.         unique_ptr<string> m_str;
4.         vector<unique_ptr<int>> vc;
5. };

```

```

1. class module {
2. public:
3.     explicit module( std::wstring const& name)
4.         : handle { ::LoadLibrary(name.c_str()) } {}
5.     private:
6.         using module_handle = std::unique_ptr<void, decltype(&::FreeLibrary)>;
7.         module_handle handle;
8. };

```

- Another question is ownership of resource:

1. For auto member resource: 1) **Same life duration(RAII)**, 2)**exclusive ownership to a single obj, but it's copyable(A a1 = a2)**. 3)**move with efficiency(A a1 = A() )** . If auto member has it's own copy and move special function, You don't need to write any special function in your class. You follow the "Rule of Zero".
2. For raw pointer and handle: 1) **default copy ctor will cause two pointer or handle refer the same resource**, It's absolutely **BAD SMELL of code** 2) So you have to follow "**Rule of five**" to build your special member function. 3) After you build five special member function, you get **RAII and exclusive ownership to a single object, and copyable and efficient move**

```

1. Class RawPointer{
2.     .....
3.     RawPointer( const RawPointer& rhs) {
4.         pRes = new Resource( *(rhs.pRes));
5.     }
6.
7.     RawPointer( RawPointer&& rhs) {
8.         pRes = rhs.pRes;
9.         rhs.pRes = nullptr;
10.    }

```

```

11. private:
12.     Resource* pRes;
13. }
```

3. For `uniqu_ptr`; 1) **Same life(RAII)** 2)**exclusive ownership but not copyable** 3) **`uniqu_ptr` support move operation.** You still follow "rule of zero"
  4. Even with `uniqu_ptr` member, If you follow "rule of zero", that is to say that you don't provide any customized special member function, then the class is not copyable. But if you build copy ctor by yourself, get raw pointer from origin side, and build a new `uniqu_ptr` member from origin side's raw pointer, you can implement copyable, and code smell better than raw pointer with "Rule of Five". So in this way, **It's not recommended to use raw pointer in RAII and ownership context.**
  5. For `shared_ptr`; 1) **Not a RAII** 2) **shared ownership**, 3) **copyable and moveable.** When you move a `shared_ptr`, origin one is set to `nullptr` and ref count doesn't increase. You still follow "rule of zero".
- **Conclusion, If you consider RAII and ownership at the same time, thing will become complex** so I would like to give you some examples to illustrate them.

1. Prefer to use auto member for most of time! It follows "Rule of Five" and supports copyable and movable. Such as `std::string`
2. For special demand, for example Car class, people can **custom** its engine, and buy **two** at the same time. In this context, your car class should use raw pointer, 1) auto member doesn't support custom 2) `uniqu_ptr` doesn't support copyable. And you have to follow "Rule of Five"

```

1. class Car{
2.     //follow "Rule of Five"
3.     Engine *pEn;
4.     ~Car() {delete pEn} // assure RAII
5. }
```

3. For special context, it doesn't support object copy: for example 1) Person class in semantic; 2) other performance consideration, Class Bigint [30000];; 3) Other implementation constraint, such as iostream class. Under such context, you can use `uniqu_ptr` to manage the resource and implement uncopyable.

```

1. class Person{
2.     uniqu_ptr<Resource> pRes;
3. }
```

4. For special context, shared resource, you can use `shared_ptr`. You still can follow "Rule of zero" and resource will be deleted when ref count is 0.

```

1. class Student{
2.     shared_ptr<SchoolBus> pBus
3. }
```

5. To know semantic of two smart pointers. Don't use them just replace raw pointer.

### 7.5.3 MI or bridge

- Multiple Inheritance will cause one grandson has two copies of grandfathers. A solution is to use virtual keyword: When you use this method, you need to follow New Constructor Rules, Detail can be seen in C++ primer P815. Just look back when you really need MI.

```

1. Father 1: virtual public grandfather
2. Father2: virtual public grandfather
3. son : public Father1, public Father 2

```

- A design discussion can be seen in "C++ FAQ Inheritance– Multiple and Virtual Inheritance"
- Suppose you have land vehicles, water vehicles, air vehicles, and space vehicles. (Forget the whole concept of amphibious vehicles for this example; pretend they don't exist for this illustration.) Suppose we also have different power sources: gas powered, wind powered, nuclear powered, pedal powered, etc. We could use multiple inheritance to tie everything together, but before we do, we should ask a few tough questions:
  1. Will the users of LandVehicle need to have a Vehicle& that refers to a LandVehicle object? In particular, will the users call methods on a Vehicle-reference and expect the actual implementation of those methods to be specific to LandVehicles?
  2. Ditto for GasPoweredVehicles: will the users want a Vehicle reference that refers to a GasPoweredVehicle object, and in particular will they want to call methods on that Vehicle reference and expect the implementations to get overridden by GasPoweredVehicle?

If both answers are "yes," multiple inheritance is probably the best way to go.

- There are at least three choices for the overall design: the bridge pattern, nested generalization, and multiple inheritance. Each has its pros/cons:

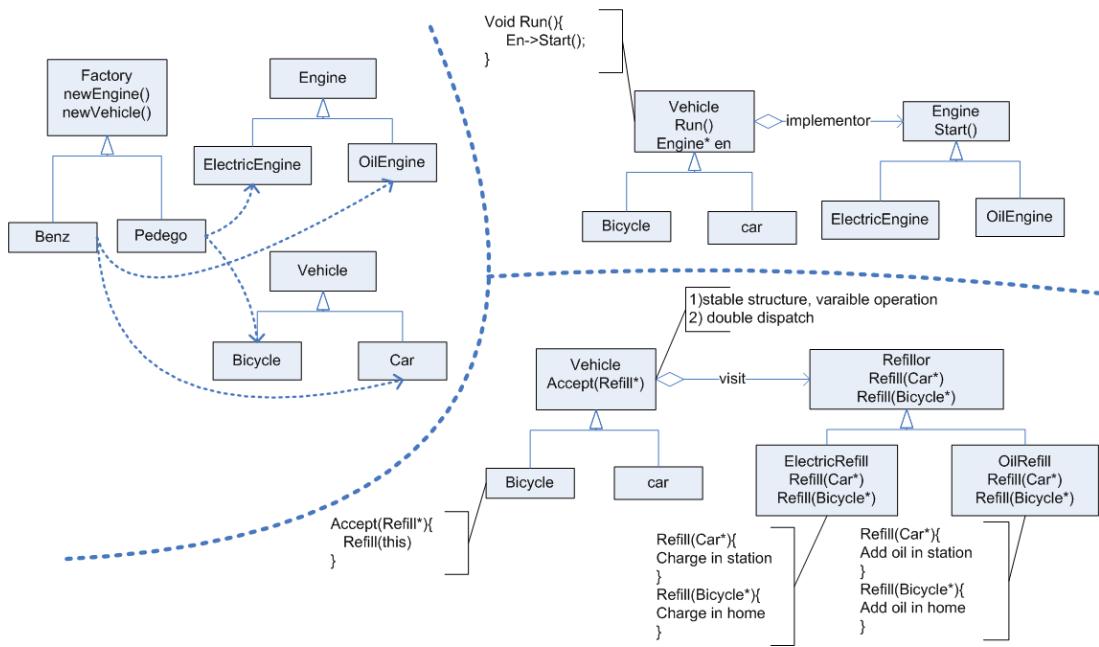
|                              | <b>Grow Gracefully?</b> | <b>Low Code Bulk?</b> | <b>Fine Grained Control?</b> | <b>Static Detect Bad Combos?</b> | <b>Polymorphic on Both Sides?</b> | <b>Share Common Code?</b> |
|------------------------------|-------------------------|-----------------------|------------------------------|----------------------------------|-----------------------------------|---------------------------|
| <b>Bridge</b>                | 😊                       | 😊<br>(N+M chunks)     | –                            | –                                | –                                 | 😊                         |
| <b>Nested generalization</b> | –                       | –<br>(N×M chunks)     | 😊                            | 😊                                | –                                 | –                         |
| <b>Multiple inheritance</b>  | –                       | –<br>(N×M chunks)     | 😊                            | 😊                                | 😊                                 | 😊                         |

- Try especially hard to use ABCs when you use MI. In particular, most classes above the join class (and often the join class itself) should be ABCs. In this context, "ABC" doesn't simply mean "a class with at least one pure virtual function;" it actually means a pure ABC, meaning a class with as little data as possible (often none), and with most (often all) its methods being pure virtual.
- Where in a hierarchy should I use virtual inheritance? Just below the top of the diamond, not at the join-class.

#### 7.5.4 factory, bridge and visitor

- These three design pattern share the same UML structure, so I introduce them together. **Design pattern is based on semantic and concrete application, not based on UML relationship and structure.**

- This is an example with bicycle and car. We use abstract factory, bridge and visit Three pattern.



# Chapter 8

## Generic programming

### 8.1 Template Basic

- A simple way to simulate template is `typedef int Item;` But when you change the data type, you need to change header file, and you can't have int and double to template class at the same programme. so C++ introduces a better method: `template<typename T>`.
- You should use templates if you need functions or container class(act likes) that apply the same algorithm to a variety of types. Templates are frequently used for container classes because the idea of type parameters matches well with the need to apply a common storage plan to a variety of types.
- A good style in template is: Use typename, not class. At the same time. simple names, such as T.

```
1. template<typename T>
2. swap(T& a, T&b) {
3.     .....
4. }
```

#### 8.1.1 template parameter

- You can have several kinds of template parameters.
  1. Type Parameters.
    - (a) Types
    - (b) Templates (only classes and alias templates, no functions or variable templates)
  2. Non-type Parameters
    - (a) Pointers
    - (b) References
    - (c) Integral constant expressions.(That is why `constexpr` is so important sometimes.)

- You can use more than one type parameter, or default type template Parameters

```
1. template <typename T1, typename T2>
2. class Pair{ }
3. Pair<double, int> pair1;
4.
5. template <typename T1, typename T2=int>
6. class Pair{ }
7. Pair<double> pair2;
```

- you can use Non-Type Argument in a template. But it will cause code bloat problem.

```

1. template <typename T, int n>
2. class ArrayTP{
3.     T ar[n];
4.     .....
5. }
6.
7. template <typename T, T n>
8. class Foo{
9. }
```

- type parameter can be another template calss. It's different with "template template parameter" which is introduced below.

```

1. template<typename T> // int T is here
2. class A
3.
4. template<typename T> //A<int> is T here.
5. class B
6.
7. B< A<int> > obj;
8. //Two T has no any relationship .
9. //you can give them any better description name
10. //just like vector< vector<int> >. That is a good example.
```

- What is template template parameter? An article is "Correct usage of C++ template template parameters". Another good one is "C++ Common Knowledge: Template Template Parameters". It gives a stack example. Below is bad way to declare stack template.

```

1. template <typename T, typename Cont>
2. class Stack {
3. public:
4.     ~Stack();
5.     void push( const T & );
6.     //...
7. private:
8.     Cont s_;
9. };
10.
11. Stack<int, List<int> > aStack1; // OK
12. Stack<double, List<int> > aStack2; // legal , not OK
13. Stack<std::string, Deque<char *> > aStack3; // error!
```

- We use template template parameter.

```

1. template <typename T, template <typename ELEM,
2.           typename = std::allocator<ELEM>>
3.         class Cont = std::deque<ELEM>
4. class Stack {
5. //...
6. private:
7.     Cont<T> s_;
8. };
9. //...
10. Stack<int> aStack1; // use default: Cont is Deque
11. Stack<std::string, std::list> aStack2; // Cont is List
```

1. Why you need std::allocator here? because std::deque is template class with two template parameter.
2. You don't need to specify inside template parameter, because `Cont<T>` will help to deduct it.
3. **There is another template keyword inside of the out template arrow brackets.**

### 8.1.2 template instantiation

- There are two confusing words in template domain, let's make them clear. What is **instantiation** and **specification**?

1. C++ use implicit or explicit instantiation to generate a specialized class or function definition from template. For implicit, you have to declare an obj, but for explicit, you don't need to declare an obj; **declare an object or use template keyword**. These two instantiation are all based on existing template implementation.

```

1. template<typename T, int n>
2. class ArrayTP ...
3.
4. ArrayTP<int, 100> stuff //implicit instantiation
5. // you have to declare obj stuff.
6.
7. template ArrayTP<string, 100>;
8. //generate a specialized class even you don't declare an obj

```

2. explicit specification is define a different template implementation for certain type. For example, For bool type, we can use optimize array storage method

```

1. template<typename T, int n>
2. class ArrayTP { ...common implementation }
3.
4. template<>
5. ArrayTP<bool, 100>{... specific implementation}
6. //There is empty<> after template
7. //You should give your own implementation.

```

3. implicit, explicit instantiation and explicit specification are all **specializations**. Because it produces a real function definition that uses specific types, The last result is not template at all
4. partial specification is different with explicit instantiation. Partial specification make generic template a little narrow, but the result is still template.
5. For class template, Explicit specialization include complete specialization and partial specialization.

```

1. template<> class Pair<int,int>{...}; //complete specialization
2. template<typename T1> class Pair<T1, int>{...}; //Partial

```

6. Non type argument and default type argument only define one template body(only one recipe). But Specialization need to define a generic template body(one recipe), For another type, It need to define a different template body(another recipe), because the code will be different with generic one.

**Instantiation is different with specialization. For instantiation, it will use template function to produce function body, but for specialization, you have to redefine your own function body**

```

1. Template<typename T>
2. void sortedArray (T) { ... };
3.
4. template void sortedArray<Person>(Person) // instantiation
5.
6. template<> void sortedArray<Person>(Person){ // specialization.
7. .... //give you own definition of fun body.
8. };
9. //Pay attention there is empty <> after template in specialization .

```

7. For function template, there is no partial specialization, So explicit specialization is complete specialization.

- instantiation happen in compiling time, not running time. When you declare a variable, It will instantiation. (It will make compiling time longer). So all the template definition must be put in head file.
- compiler will parse the template definition before it instantiation.** Why? It's a compiler, not a macro processor. Errors in the template itself won't be detected as long as it is only instantiated with 'friendly' types that don't trigger the errors: for example, if the template assumes that the type always has such-and-such a method.
- C++ Coding standards 65 states customization of point. In order to understand it. You need to understand two basic conceptions: "two phases lookup" and "dependent name".
- two phases lookup can see "Dependent name lookup for C++ templates" in the ref and "Two-Phase or Not Two-Phase: The Story of Dependent Names in Templates" in the ref.

### 8.1.3 template specialization

- Class templates can be partially specialized, and the resulting class is still a template. Partial specialization allows template code to be partially customized for specific types in situations, such as:
  - A template has multiple types and only some of them need to be specialized. The result is a template parameterized on the remaining types.
  - A template has only one type, but a specialization is needed for pointer, reference, pointer to member, or function pointer types. The specialization itself is still a template on the type pointed to or referenced.
  - pointer sometimes need to be deal with differently, at that time we need partial specialization of template class. A very good article is:  
<https://www.learnCPP.com/cpp-tutorial/13-8-partial-template-specialization-for-pointers/>. You can see that there is three level, It become narrower and narrower. 1) base template, 2)partial specialization 3) complete specialization of member.

```

1. template <class T>
2. class Storage{
3.   T m_value;
4. public:
5.   Storage(T value){
6.     m_value = value;
7.   }
8.
9. // Partial-specialization of Storage class for pointers

```

```

10. template <class T>
11. class Storage<T*>{
12.     T* m_value;
13. public:
14.     Storage(T* value){
15.         m_value = new T(*value); //To make deep copy
16.     }
17.
18. // Full specialization of constructor for type char*
19. template <>
20. Storage<char*>::Storage(char* value){
21.     // Figure out how long the string in value is
22.     int length = 0;
23.     while (value[length] != '\0')
24.         ++length;
25. }
```

- For function, there is no template partial specification. Difference between template function overload and partial specializations:

1. partial specializations is just use in template class.
2. partial specializations need put another <type> after class name
3. If you want to have custom implementation of function with same name, use overload function, if you need custom implementation of class with same name, use partial specialization.

```

1. template <typename T1, typename T2>
2. class Pair{} //general one
3.
4. template <typename T1>
5. class Pair<T1, T1> // partial specializations
6.
7. template <typename T1>
8. class Pair<T1, int> // partial specializations
9. //—————
10. template <typename T>
11. class Foo //general one
12.
13. template <typename T>
14. class Foo<T*> //partial specializations.
```

- keep the design of a template in mind and not use it blindly. When you input pointer as typename, you should be on high alert.
- Explicit specification usually need define all the implementation in it. If there is a lot of repetition. There are two helpful options:

1. Add a special function just suitable for certain type.

```

1. template <typename T>
2. class A{
3. public:
4.     void onlyForInts(T t){
5.         static_assert(std::is_same<T, int>::value, "Only ints!");
6.     }
7.
8. protected:
9.     std::vector<T> myVector;
```

```

10.    };
11.
12. int main() {
13.     A<int> i;
14.     i.onlyForInts(1); // works !
15.
16.     A<float> f;
17.     //f.onlyForInts(3.14f); // does not compile !
18. }
```

2. Use type trait and overload, select at the compile time. See enable\_if example below.

- A very good article about specialization is chapter 12 in "C++ template: The complete guide". Remember, you should read each sentence in this chapter.
  1. You can complete(full) specialization of template class.
  2. You also can complete(full) specialization of template function.
  3. You can complete(full) specialization of member in template class.

```

1. template<> int f(int)
2. template<>
3. class S<void>
4. // full specialization must have a empty template<>
```

4. You can partial specialization template class.

```

1. template<T> //1) has template<T>
2. class S<T*> //2) after class name has another <>
```

5. Pay attention the specialization of declaration. it should be after the basic template.
6. **For template function, there is overload and full specialization, for template class, there is full and partial specialization. Template function doesn't support partrial specialization, because it can use overload to reach the same result.**

#### 8.1.4 template and friend

- you can have:
  1. friend class.
  2. friend function.
  3. friend template.

At the same time, they can be bound or non-bound. If it's bound, you can find host class parameter T in the friend declaration.

- There are three different kinds for a template class: non-template friend, Bound-template, unbound-template. Below is non-template friend function counts(); The counts is not invoked by an HasFriend obj and has not object parameters. If counts want to access a HasFriend object, It can access a global one, and use a global pointer to access non-global object.

```

1. template<typename T>
2. class HasFriend{
3.     friend void counts();
4. }
```

- Bound-template friend function. You need to define explicit specialization for the friends you plan to use. **Here, reports is not template function.**

```

1. template<typename T>
2. class HasFriend{
3.     friend void reports(HasFriend<T> &);
4. }
5.
6. void reports(HasFriend<int> &hf){
7.     cout << hf.item << endl;
8. }
9. void reports(HasFriend<short> &hf) {..}
10. void reports(HasFriend<char> &hf) {..}
11. // explicit specialization for the type you plan to use.

```

- A better bound-template, reports has `<>` after it. and you don't need redefine reports many time like previous codes. **reports is a template function here. You can think that is a implicit instantiations.**

```

1. //1)
2. template <typename T> void report(T &);
3.
4. //2)
5. template<typename TT>
6. class HasFriend{
7.     friend void reports<>(HasFriend<TT> &);
8.     // it has <> after reports function name.
9. }
10.
11. //3)
12. template<typename T>
13. void reports(T &hf){
14.     cout << hf.item << endl;
15. }

```

- non-bound friend template.

```

1. template<typename T>
2. class ManyFriend{
3.     template<typename C, typename D>
4.         friend void show(C&, D&);
5.     };
6.
7. template <typename C, typename D> void show2(c& c, D& d){
8.     cout << c.item << d.item << endl;
9. }
10.
11. ManyFriend<int> mi;
12. ManyFriend<double> md;
13. show2(mi, md);

```

- Difference between bound and non-bound:

1. bound friend is specialization of template function, so It has `<>` after the function name.
2. non-bound template use different typename, such as C and D in previous example, and it also use template keyword
3. bound template will produce more function implementation , but non-bound template will only produce ONE function implementation.

### 8.1.5 member function templates

- Use member function templates to accept "all compatible types." detail can be found in effective c++ item 45

```

1. template<typename T>
2. class SmartPtr {
3. public:
4.     template<typename U> // member template
5.     SmartPtr(const SmartPtr<U>& other); //for a generalized copy ctor
6.     ...
7. }
```

- When writing a class template that offers functions related to the template that support implicit type conversions on all parameters, define those functions as friends inside the class template. Detail can be found in effective c++ item 46.
- implicit type conversion functions are never considered during template argument deduction. Never. Such conversions are used during function calls, yes, but before you can call a function, you have to know which functions exist.

```

1. template<typename T>
2. class Rational {
3. public:
4.     ...
5.     friend const Rational operator*(const Rational& lhs,
6.                                     const Rational& rhs){
7.         return Rational(lhs.numerator() * rhs.numerator(),
8.                         lhs.denominator() * rhs.denominator());
9.     }
10. }
```

- TMP has two great strengths. First, it makes some things easy that would otherwise be hard or impossible. Second, because template metaprograms execute during C++ compilation, TMP can be more efficient than a "normal" C++ program, because the traits approach is TMP. Remember, traits enable compile-time if...else computations on types by overload

## 8.2 Type Inference

### 8.2.1 template type deduction

Given below code, We will introduce the basic type deduction rules.

```

1. template<typename T>
2. void f(ParamType param);
3.
4. f(expr); // deduce T and ParamType from expr
```

1. ParamType is a Reference or Pointer, but not a Universal Reference. If expr's type is a reference, **ignore the reference part**. Then pattern-match expr's type against ParamType to determine T.

```

1. template<typename T>
2. void f(T& param); // param is a reference
3.
4. int x = 27; // x is an int
```

```

5. const int cx = x; // cx is a const int
6. const int& rx = x; // rx is a reference to x as a const int
7.
8. f(x); // T is int, param's type is int&
9. f(cx); // T is const int, param's type is const int&
10. f(rx); // T is const int, param's type is const int&

```

## 2. ParamType is a Universal Reference

- (a) If expr is an lvalue, both T and ParamType are deduced to be lvalue references. **although ParamType is declared using the syntax for an rvalue reference, its deduced type is an lvalue reference.**
- (b) If expr is an rvalue, the "normal" (i.e., Case 1) rules apply.
- (c) **Only universal reference deduction distinguishes lvalue and rvalue**

```

1. template<typename T>
2. void f(T&& param); // param is now a universal reference
3. int x = 27; // as before
4. const int cx = x; // as before
5. const int& rx = x; // as before
6.
7. f(x); // x is lvalue, so T is int&, param's type is also int&
8. f(cx); // cx is lvalue, so T and param's type are const int&
9. f(rx); // rx is lvalue, so T and param's type are const int&
10. f(27); // 27 is rvalue, so T is int, param's is therefore int&

```

## 3. ParamType is Neither a Pointer nor a Reference. As before, if expr's type is a reference, **ignore the reference part**. If, after ignoring expr's reference-ness, expr is const, **ignore const too**. If it's volatile, also ignore that.

```

1. template<typename T>
2. void f(T param); // param is now passed by value
3.
4. int x = 27; // as before
5. const int cx = x; // as before
6. const int& rx = x; // as before
7. f(x); // T's and param's types are both int
8. f(cx); // T's and param's types are again both int
9. f(rx); // T's and param's types are still both int

```

## 4. drops const and volatile qualifiers only for by-value parameters. for parameters that are references-to-const pointer or pointers-to-const pointer, we don't skip const.

```

1. template<typename T>
2. void f(T param); // param is now passed by value
3.
4. const int* const p1 = &x;
5. f(p1) // param is const int*, top const has been skipped.
6.
7. const int*& rp = p1;
8. f(p1) // param is const int*, top const is default for reference.
9. // pointer to reference is illegal.

```

## 5. Conclusion: Take four steps to decide:

- (a) array or function decay to pointer, if they are not used to reference ParamType

- (b) universal reference for lvalue, T is lvalue reference. (keep const)  
 (c) reference is ignored  
 (d) for value-type ParamType, const and volatile are ignored.
6. The last method is manually specify the type.

```

1. template<typename T>
2. void f(T param);
3.
4. int &x = a;
5. f(x) //T is int
6. f<int&>(x) //T is int&
```

### 8.2.2 auto type deduction

- Just like a template type, auto follow the same deduction rule. template type deduction happens when you call a function or build a customize type. **auto deduction happen when you initialize using assignment. Here just think the auto is T in the template deduction.**

```

1. auto x = 27; // x is neither ptr nor reference
2. const auto& rx = x; // rx is a non-universal ref.
3. auto && ax = 27; // ax is universal(forwarding) ref
```

- When combine auto and brace init in assignment, it means `std::initializer_list<T>`

```

1. auto x = { 11, 23, 9 }; // x's type is std::initializer_list<int>
2.
3. template<typename T> // template with parameter
4. void f(T x);
5. f({ 11, 23, 9 }); // error! can't deduce type for T
6.
7. void f(std::initializer_list<T> initList); //it will work
```

- auto in a function return type or a lambda parameter implies template type deduction, not auto type deduction.

```

1. auto createInitList() {
2.     return { 1, 2, 3 };
3.     // error: can't deduce type for { 1, 2, 3 }
4. }
5.
6. std::vector<int> v;
7. auto resetV =
8. [&v](const auto& newValue) { v = newValue; }; // C++14
9.
10. resetV({ 1, 2, 3 }); // error! can't deduce type for { 1, 2, 3 }
```

- When use with auto, you need to be careful. The only situation where = is preferred over {} is when using auto keyword to get the type determined by the initializer.

```

1. auto z1 {99}; // z1 is an initializer_list<int>
2. auto z2 = 99; // z2 is an int
3.
4. auto d = {1.23};
5. // d is a std::initializer_list<double>
6. auto d = double{1.23};
7. auto d = 1.23
8. // Good --- d is a double, not a std::initializer_list.
```

- `auto&&` is a kind of forwarding reference.

```

1. auto&& uref1 = x; // x is int and lvalue ,
2. // so uref1's type is int&
3. auto&& uref2 = cx; // cx is const int and lvalue ,
4. // so uref2's type is const int&
5. auto&& uref3 = 27; // 27 is int and rvalue ,
6. // so uref3's type is int&C3

```

### 8.2.3 decltype deduction

#### 8.2.3.1 basice knowledge

- decltype has two different deduct rules when facing different kinds of expression.

1. expression whose type is to be determined is a plain variable or function parameter, like `x`, or a class member access, like `p->m_x`. In that case, decltype lives up to its name: it determines the type of the expression to be the declared type,

```

1. vector<int> v; // decltype(v) is vector<int>
2. struct S {
3.     int m_x;
4. };
5.
6. int x;
7. const int cx = 42;
8. const int& crx = x;
9. const S* p = new S();
10.
11. decltype(x) a; // a is int, as auto a = x
12.
13. decltype(cx) b; // b is const int
14. auto b = cx; // auto ignore const, b is int
15.
16. decltype(crx) c; // c is const int&.
17. auto c = crx; // ignore reference and const, c is int
18.
19. // p is a pointer to const S. But decltype goes
20. // by the declared type, which is int.
21. decltype(p->m_x) d; // d is int
22. auto d = p->m_x; // auto ignore const, d is int

```

2. If expr is not case 1, then three different rules:
  - If expr is an lvalue, then `decltype(expr)` is `T&`.
  - If expr is an xvalue, then `decltype(expr)` is `T&&`.
  - Otherwise, expr is a prvalue, and `decltype(expr)` is `T`.

- For the case 2, give more examples

```

1. struct S {
2.     int m_x;
3. };
4. int x;
5. const int cx = 42;
6. const int& crx = x;
7. const S* p = new S();
8.
9. typedef decltype((x)) x_with_parens_type;

```

```

10. typedef decltype((cx)) cx_with_parens_type;
11. typedef decltype((crx)) crx_with_parens_type;
12. typedef decltype((p->m_x)) m_x_with_parens_type;

```

1. (x) has type int, and decltype adds references to lvalues. Therefore, `x_with_parens_type` is `int&`.
2. The type of (cx) is const int. Since (cx) is an lvalue, decltype adds a reference to that: `cx_with_parens_type` is `const int&`.
3. The type of (crx) is `const int&`, and it is an lvalue. decltype adds a reference. By the C++11 reference collapsing rules, that makes no difference. Hence, `crx_with_parens_type` is `const int&`.
4. S::m\_x is declared as int. Since p is a pointer to const, the type of (`p->m_x`) is `const int`. Since (`p->m_x`) is an lvalue, decltype adds a reference to that. Therefore, `m_x_with_parens_type` is `const int&`.

- For some complex expression examples:

```

1. const S foo();
2. const int& foobar();
3. std::vector<int> vect = {42, 43};
4.
5. typedef decltype(foo()) foo_type;
6. typedef decltype(foobar()) foobar_type;
7. decltype(vect[0]) first_element = vect[0];
8. double d1, d2;
9. typedef decltype(d1 < d2 ? d1 : d2) cond_type;
10. int x = 0;
11. typedef decltype(x < d2 ? x : d2) cond_type_mixed;

```

1. `foo()` is declared as returning `const S`. The type of `foo()` is `const S`. Since `foo()` is a prvalue, decltype does not add a reference. Therefore, **`foo_type` is `const S`**.
2. The type of `foobar()` is `const int&`, and it is an lvalue. Therefore, decltype adds a reference. By the C++11 reference collapsing rules, that makes no difference. Therefore, **`foobar_type` is `const int&`**.
3. `std::vector<int>`'s `operator[]` is declared to have return type `int&`. Therefore, the type of the expression `vect[0]` is `int&`. Since `vect[0]` is an lvalue, decltype adds a reference. By the C++11 reference collapsing rules, that makes no difference. Therefore, **`first_element` has type `int&`**.
4. The type of the expression is `double`, and the expression is an lvalue. Therefore, a reference is added, and **`cond_type` is `double&`**.
5. The type of the expression is `double`. The expression is a prvalue, because in order to accomodate the promotion of `x` to a `double`, a temporary has to be created. Therefore, no reference is added, and **`cond_type_mixed` is `double`**.

### 8.2.3.2 decltype usage

- **decltype can be used in deducing template function return value.**

```

1. template<typename T, typename U>
2. auto eff(T t U u) -> decltype(T*U) {
3.     ...
4. }

```

- When the function return a lvalue reference, auto will not work here(skip reference), only decltype can success keep reference.

```

1. template<typename Container, typename Index>
2. auto authAndAccess(Container& c, Index i) -> decltype(c[i]) {
3.     authenticateUser();
4.     return c[i];
5. }
6.
7. template<typename Container, typename Index> // C++14; works,
8. decltype(auto) authAndAccess(Container& c, Index i) {
9.     authenticateUser();
10.    return c[i];
11. } // Pay attention to decltype(auto),
12. // it deduct template_type&, not only template_type

```

- make template function support universal reference.

```

1. template<typename Container, typename Index> // final c++14
2. decltype(auto) authAndAccess(Container&& c, Index i) {
3.     authenticateUser();
4.     return std::forward<Container>(c)[i];
5. }
6.
7. template<typename Container, typename Index> // final c++11
8. auto authAndAccess(Container&& c, Index i)
9.     -> decltype(std::forward<Container>(c)[i]) {
10.     authenticateUser();
11.     return std::forward<Container>(c)[i];
12. }
13.
14. auto s = authAndAccess(queue, 5);
15. auto s = authAndAccess(std::move(queue), 5);
16. // Both work here. For the first, it returns a lvalue reference
17. // and for the second, it returns a rvalue reference.

```

- The use of decltype(auto) is not limited to function return types. It can also be convenient for declaring variables when you want to apply the decltype type deduction rules to the initializing expression.

```

1. Widget w;
2. const Widget& cw = w;
3.
4. auto myWidget1 = cw; // auto type deduction:
5. // myWidget1's type is Widget
6.
7. decltype(auto) myWidget2 = cw;
8. // myWidget2's type is const Widget&

```

- An important property of decltype is that its operand never gets evaluated. For example, you can use an out-of-bounds element access to a vector as the operand of decltype with impunity:

```

1. std::vector<int> vect;
2. assert(vect.empty());
3. typedef decltype(vect[0]) integer;
4. // vect is empty now, but we use vect[0] in decltype.

```

- Another property of decltype that is worth pointing out is that when decltype(expr) is the name of a plain user defined type (not a reference or pointer, not a basic or function type), then decltype(expr) is also a class name. This means that you can access nested types directly:

```

1. template<typename R>
2. class SomeFunctor {
3. public:
4.     typedef R result_type;
5.     result_type operator()() {
6.         return R();
7.     }
8.     SomeFunctor() {}
9. };
10.
11. SomeFunctor<int> func;
12. typedef decltype(func)::result_type integer; // access nested type

```

- **If you're declaring variables inside a class.** Instead of typing out the full name of the type of the iterator, you can use decltype:, you can't use auto. This works because decltype doesn't actually execute the expression given as its argument— it is only used by the type checker to determine a type.

```

1. class A {
2.     std::vector<std::pair<int, std::string>> array;
3.     decltype(array.begin()) iter; // You don't need init expression here.
4.     // you can't use auto here, because auto need init expression
5. };

```

- We could have also done the above example with declval. allows you to use decltype without constructing the object. The type doesn't even need a default constructor, and in fact, it can be used with an incomplete type. The above example could be rewritten as:

```

1. template <typename C>
2. decltype(std::declval<const C>().begin())
3. foo(const C& c){
4.     return one iterator of c
5. }
6. //Here no any variable in below expression.

```

- declval is commonly used in templates where acceptable template parameters may have no constructor in common, but have the same member function whose return type is needed.

```

1. struct Default { int foo() const { return 1; } };
2.
3. struct NonDefault{
4.     NonDefault(const NonDefault&) { }
5.     int foo() const { return 1; }
6. };
7.
8. decltype(Default().foo()) n1 = 1; // type of n1 is int
9. //decltype(NonDefault().foo()) n2 = n1; // error: no default ctor
10. decltype(std::declval<NonDefault>().foo()) n2 = n1;
11. //ok now, n2 is int

```

- result\_of usage. result\_of was introduced in Boost, and then included in TR1, and finally in C++0x. Therefore result\_of has an advantage that is backward-compatible (with a suitable

library). decltype is an entirely new thing in C++, does not restrict only to return type of a function, and is a language feature.

```

1. struct foo {
2.     double operator()(char, int);
3. };
4. //same
5. std::result_of<foo(char, int)>::type d1 = 10.0;
6. decltype(foo()('c', 1)) d2 = 10.0;

```

- Basic knowledge of decltype detail can be found "How decltype Deduces the Type of an Expression: Case 1"

#### 8.2.4 check type

- There are three ways to view deducted type

1. use IDE
2. use Compiler

```

1. template<typename T> // declaration only for TD;
2. class TD; // TD == "Type Displayer"
3.
4. TD<decltype(x)> xType; // elicit errors containing x's and y's types
5. TD<decltype(y)> yType;

```

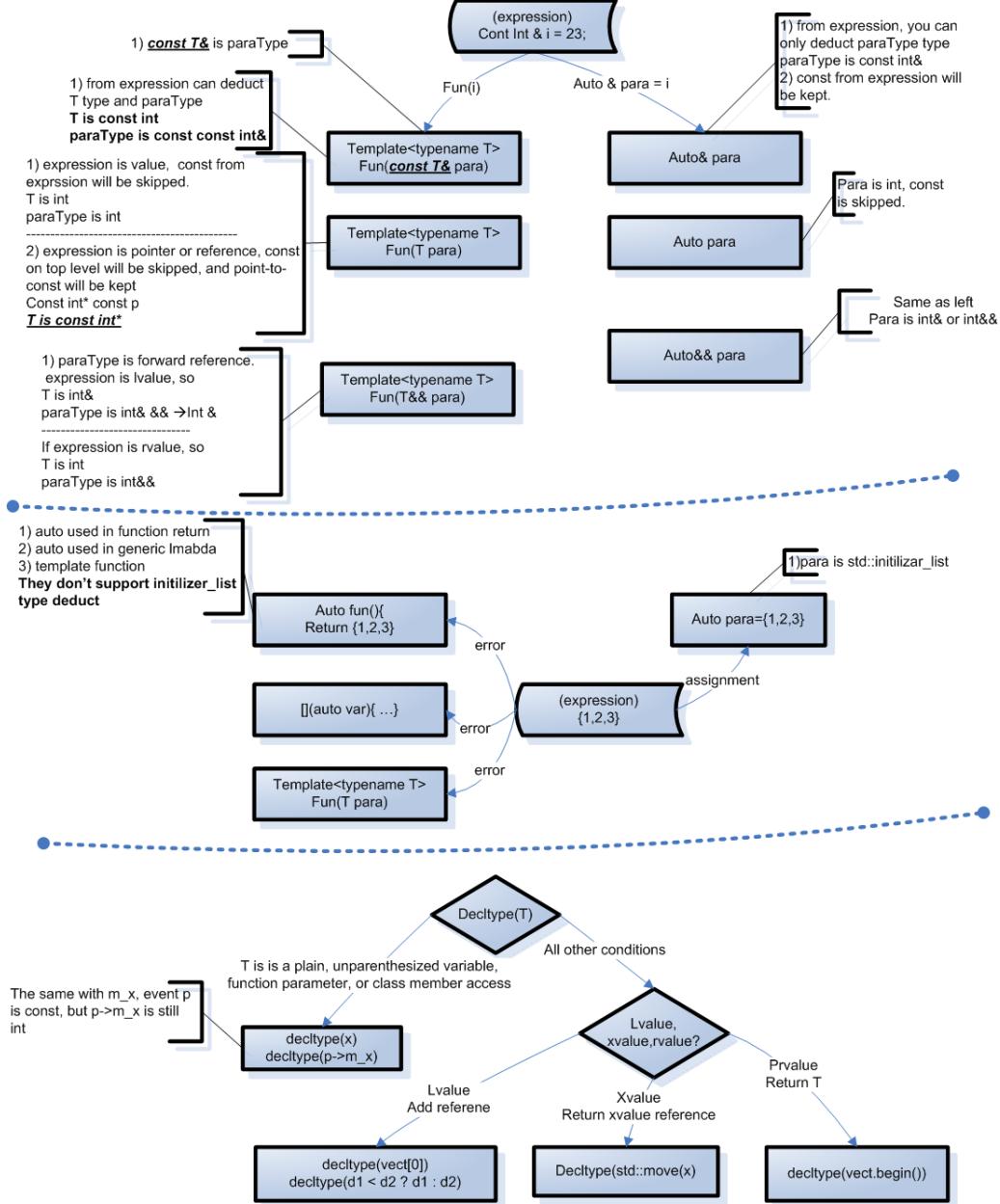
3. use typeid and std::type\_info::name

```
1. std::cout << typeid(x).name() << '\n'; // display types for
```

#### 8.2.5 summary

- **auto**, template T and decltype are three kinds of type deduction context. auto and template T are almost same, beside when we meet initializer\_list. but decltype are quite different with the other two.
- When you use decltype(auto), **auto tell compiler to automatically type deduction, but please use decltype rules.** They are most used when we deal with some expression which return back lvalue, such as a[0], in this way, if you use auto, it will ignore reference. but decltype will keep it.
- **auto** can be used to set the type of a newly declared variable from its initializing expression. It removes the reference, if any, from the expression's type, then removes top-most const and volatile qualifiers.
- decltype can be used in a wider variety of contexts, such as typedefs, function return types, and even in places where a class name is expected. There are two different ways in which decltype(expr) can work, depending on the nature of expr

- Basic idea can be illustrated below:



## 8.3 type traits and policy

### 8.3.1 implementation

- The most common way of implementation is template class specification.

- most common usage of trait is member constants, such as `is_integer<T>::value`
- You use a templated structure, usually named with the type trait you are after. Eg) `is_integer`, `is_pointer`, `is_void`. The structure contains a static const bool named `value` which defaults to a sensible state. You use a type trait by querying its `value`, like: `my_type_trait<T>::value`
- Make default template return default value, then make template specification return what you expect.

```

1. template <typename T>
2. struct is_swapable {
3.     static const bool value = false;
4. };
5.
6. template <>
7. struct is_swapable<short> {
8.     static const bool value = true;
9. };
10.
11. template <>
12. struct is_swapable<int> {
13.     static const bool value = true;
14. };

```

4. A good article is "A simple introduction to type traits"

- Another way is to build new trait based on existing trait.

```

1. template <typename T>
2. struct is_swapable {
3.     static const bool value =
4.         std::is_integral< T >::value && sizeof( T ) >= 2;
5. };

```

- Type trait implementation trick, use `sizeof` and `static_cast`, these two can be executed at compile time. For example, we want to test if a class derived from another class.

```

1. template<typename D, typename B>
2. class IsDerivedFromHelper{
3.     class No { };
4.     class Yes { No no[ 3 ]; };
5.
6.     static Yes Test( B* );
7.     static No Test( ... );
8.     public:
9.     enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };
10.
11. };
12.
13. template <class C, class P>
14. bool IsDerivedFrom() {
15.     return IsDerivedFromHelper<C, P>::Is;
16. }

```

- In C++11, you can use `constexpr` instead of `sizeof`, and `decltype` and `declval` also get type in the compiling time..

1. two overload template test functions. One return true, default return false.
  2. In the funciton which return true, use `decltype` and `declval` to simulate call some functions you want to judge. make these type used in function parameter or return
  3. Pay attention to the usage of `static constexpr bool`. Use test function to initialized `constexpr` value
- Examples1, one is judge if T has member function.

```

1. template <class T>
2. struct IsFunction{
3.     // int (*p)[1] pointer to array(int [1])
4.     // int *p[1], that is array of pointer .
5.     template <typename C>
6.         static constexpr bool test(C (*)[1] ) {
7.             return true;
8.         }
9.
10.    template <typename C>
11.        static constexpr bool test( ... ) {
12.            return false;
13.        }
14.        //use 0 as pretend pointer.
15.        static constexpr bool value = test<T>(0);
16.    };
17.
18. int f();
19. IsFunction<decltype(f)>::value; //value is true

```

1. what is function type?

```

1. typedef int FT(); //FT is type
2.
3. //usage 1: declare function pointer
4. int f(); //f is variable
5. FT* fp= f; //fp is variable and type is function pointer
6.
7. //usage 2: declare a variable,
8. FT fv; //just like int fv(); declare a function
9.           //How to define this variable make it linkable?
10. void fv() {...} //Just like common funciton definition.

```

2. You can't not define function type array.

```

1. typedef int FT(); //FT is type
2. FT fa[3]; //compile error
3.           //error: declaration of 'fa' as array of functions

```

- Another example is to test if T is function. We test if the type has serialize using decltype and declval.

```

1. template <class T>
2. struct hasSerialize{
3.     template <typename C>
4.         static constexpr decltype(std::declval<C>().serialize(), bool()) test(int /* unused */){
5.             return true;
6.         }
7.
8.         template <typename C>
9.             static constexpr bool test( ... ) {
10.                 return false;
11.             }
12.
13.             // int is used to give the precedence!
14.             static constexpr bool value = test<T>(int());
15.         };
16.

```

- last kind of implementation is use category. An example can be found in iterator.

```

1. struct input_iterator_tag {};
2. struct output_iterator_tag {};
3. struct forward_iterator_tag: public input_iterator_tag {};
4. struct bidirectional_iterator_tag: public forward_iterator_tag {};
5. struct random_access_iterator_tag: public bidirectional_iterator_tag {};
6.
7. template < ... > // template params elided
8. class deque {
9.     class iterator {
10.         public:
11.             typedef random_access_iterator_tag iterator_category;
12.             ...
13.     };
14.
15. template<typename IterT>
16. struct iterator_traits {
17.     typedef typename IterT::iterator_category iterator_category;
18.     ...
19. };
20.
21. template<typename T> // partial template specialization
22. struct iterator_traits<T*>{ // for built-in pointer types
23.     typedef random_access_iterator_tag iterator_category;
24.     ...
25. };

```

- How to deal with member function pointer? Given an example code below, if you want to have member function pointer, its type should be: `int (A::*())() const &`

```

1. struct A {
2.     int fun() const&;
3. };
4. typedef int FunSignature() const&;
5. FunSignature A::*p = &A::fun;

```

- When we want to use template class to deduct it, it can be used in `U T::*`.

```

1. struct A {
2.     int fun() const&;
3. };
4.
5. template<class T, class U>
6. struct PM_traits<U T::*> {
7.     using member_type = U;
8. };
9.
10. int main() {
11.     using T = PM_traits<decltype(&A::fun)>::member_type;
12.     // T is int () const&
13. }

```

- Based on previous knowledge, we can implement

```

1. template< class T >
2. struct is_member_function_pointer_helper : std::false_type {};
3.
4. template< class T, class U>

```

```

5. struct is_member_function_pointer_helper<T U::*> :
6.     std::is_function<T> {};
7.
8. template< class T >
9. struct is_member_function_pointer :
10.     is_member_function_pointer_helper< std::remove_cv_t<T> {};;
11.
12. class A {
13. public:
14.     void member() { }
15. };
16.
17.
18. // output 0 if A::member is a data member and not a function
19. cout<<std::is_member_function_pointer<decltype(&A::member)>::value

```

- summary: Common type traits implementation methods
  1. template specification: `is_integer`
  2. use basic trait: `is_swap`
  3. use template parameter deduction : `std::is_function` and `is_member_function`, `is_array`, `is_class(T::*)`
  4. define trait in existing class : `iterator_trait`.
  5. overload template function, deduce return or function parameter based on template parameter T's expression: `has_serialize`, `isIsFunction`
- There are two articles which are good. "An introduction to C++'s SFINAE concept: compile-time introspection of a class member" and "checking expression validity in-place with C++17"

### 8.3.2 Usage

- A type trait is a way for you to get information about the types passed in as template arguments, at compile time, so you can make more intelligent decisions.
- A type trait can return any information or customization about a type. A simple example is `is_integer`, it just return bool value information. A complex example is `char_trait`. It can return a few functions. In this way, you also can think that `char_trait` is a policy.

## 8.4 Template function

- When you use template function with real argument, compiler will deduct type from real argument, then generate real function code for you. So it will make compiling longer.

```

1. swap(i1, i2);
2. swap(f1, f2);
3. // compiler can extract type information from i1
4. // compiler will produce two swap functions bodies.

```

- Why we need a overload template function? Because not all type support the same operation. For example, in your template function, you use = operator, but when you use array for this template function type, compiler will report error.

```

1. template <typename T>
2. void swap(T a[], T b[], int n)
3.
4. template <typename T>
5. void swap(T &a, T &b )

```

- Overload is different with Specializations, Overload means that you have different function signatures. Specialization have the same function signature.
- Some good articles:  
 "Why Not Specialize Function Templates?" .  
 "Why Argument Dependent Lookup doesn't work with function template dynamic\_pointer\_cast".  
 "C++ template function taking template class as parameter".

#### 8.4.1 overload resolution

- Given a function name, you can have regular, template and explicit specialization temple. When pick a function, regular > specialization > template. function picking ranking from best to worst is:
  1. Exact match, regular function.
  2. Template if you have define the same template function name.
  3. Conversion by promotion.
  4. Conversion by standard conversion.
  5. user-defined conversion.

- What is exact match. There are table below:

| Actual argument             | Formal argument                  |
|-----------------------------|----------------------------------|
| type-name                   | type-name &                      |
| type-name &                 | type-name                        |
| type-name [ ]               | type-name*                       |
| type-name ( argument-list ) | ( *type-name ) ( argument-list ) |
| type-name                   | const type-name                  |
| type-name                   | volatile type-name               |
| type-name*                  | const type-name*                 |
| type-name*                  | volatile type-name*              |

- About exact match, there are three result:

1. If there are two exact matches, compiler can't distinguish them, then it will report error.
2. If reference and pointer, even there are two exact match, It will pick up first according to const.

```

1. int i = 2;
2.
3. f(const int& j); //#1
4. f(int& j); //#2, 2 will be selected, because i isn't const

```

3. If there are exact match, it will pick up before template, even template has EXACT specification.
- For template, order is:

1. specialized argument template
2. specialization version inside this specialized argument template.

```

1. template<typename T>
2. f(T t); //#1
3.
4. template<>f<int*>(int* t) #2
5.
6. template<typename T>
7. f(T* t); //#3
8.
9. template<> f<int> f( int* t) //#4
10.
11. int * p;
12. f(p) // #4>#3>#2>#1

```

- For above example, If you omit type inside <> after function name f, It will depends on location.

```

1. template<typename T>
2. f(T t); //#1
3.
4. template<>f<>(int* t) //#2 is specialization of #1
5. /////////////////////////////////
6. template<typename T>
7. f(T* t); //#3
8.
9. template<>f<>(int* t) //#4 put here is specialization of #3
10.
11. int *p;
12. f( p );
13. //If put specialization at #2, it will call #3, wrong!
14. //If put specialization at #4, it will call #4, right!.

```

- You can tell compiler that you prefer template function over overload one.

```

1. template<typename T>
2. void f(T t); //#1
3.
4. void f(int t) //#2
5.
6. f<>(2) // tell compiler use #1, not #2.

```

#### 8.4.2 template function specification

- Explicit Specialization: Because for char \*, you can't use >, but use strcmp. So you need to build explicit Specialization Version. The prototype and definition of an explicit specialization should be preceded by template<> and should mention the specialized type by name.

```

1. Template<typename T>
2.     void sortedArray(T) { ... };
3.
4. template<> void sortedArray<const char *>() { ... }

```

- Some explicit specification syntax.

```

1. template<typename T>
2. void foo(T param);

```

```

3. //syntax 1: not a specialization, it is an overload
4. void foo(int param);
5.
6. //syntax 2: ill-formed, not recommend
7. void foo<int>(int param);
8.
9. //syntax 3: normal explicit specialization, specify T directly
10. template <> void foo<int>(int param);
11.
12. //syntax4: same as above, but use template argument deduction
13. template <> void foo(int param); //explicit specialization
14.
15. //syntax5: but works only if template argument deduction is possible!
16. template void foo(int param); //explicit instantiation
17. template void foo<int>(int param); //explicit instantiation
18. template void foo<>(int param); //explicit instantiation
19.

```

- There seems to be (a lot) of confusion regarding explicit instantiation and specialization. The code I posted above deals with explicit instantiation. The syntax for specialization is different. Here is syntax for specialization: Note that angle brackets after template!

```

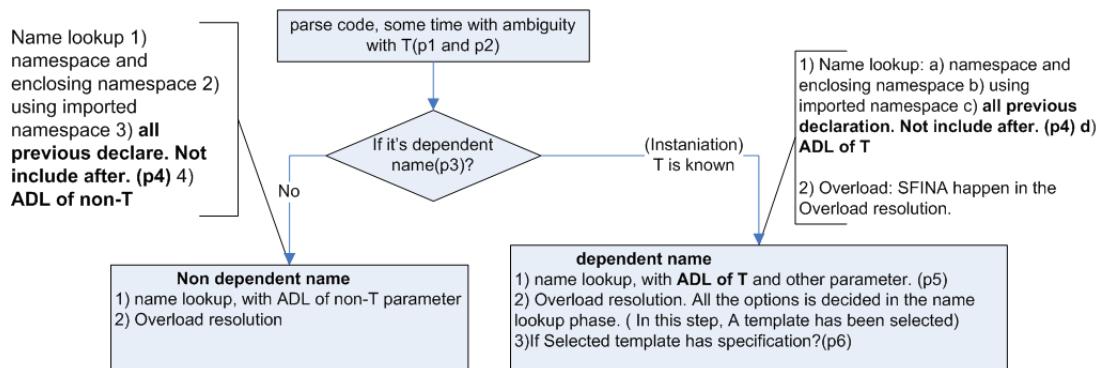
1. template <typename T> void func(T param) {} // definition
2.
3. template void func<int>(int param); // explicit instantiation.
4.
5. template <> void func<int>(int param) {} // specialization

```

- If you want to customize a function base template and want that customization to participate in overload resolution (or, to always be used in the case of exact match), make it a plain old function, not a specialization. And, if you do provide overloads, avoid also providing specializations. Detail you can google "Why Not Specialize Function Templates?" **prefer to use overload than template function specification.**

### 8.4.3 summary

- A good article is "Overload resolution" in Andrzej's C++ blog. It taught you 1) name kookup and 2)overload and 3)instanition three conception very well.



- More explanation about previous figure.

1. p1 and p2,

```

1. template <typename T> struct Base {
2.     typedef int MyType;
3. };
4.
5. template <typename T> struct Derived : Base<T> {
6.     void g() {
7.         // A. error: 'MyType' was not declared in this scope
8.         // MyType k = 2;
9.
10.        // B. error: need 'typename' before 'Base<T>::MyType' because
11.        // 'Base<T>' is a dependent scope
12.        // Base<T>::MyType k = 2;
13.
14.        // C. works!
15.        typename Base<T>::MyType k = 2;
16.    }
17. };

```

```

1. struct Foo {
2.     template<typename U>
3.     static void foo_method() {
4.     }
5. };
6.
7. template<typename T> void func(T* p) {
8.     // A. error: expected primary-expression before '>' token
9.     // T::foo_method<T>();
10.
11.    // B. works!
12.    T::template foo_method<T>();
13. }

```

## 2. p3

```

1. template <typename T> struct Base {
2.     void f() { ... }
3. };
4.
5. template <typename T> struct Derived : Base<T> {
6.     void g() {
7.         // f(); error happen here.
8.         this->f() will resolve problem.
9.     }
10. }

```

3. p4 name look up has two stages. 1) from process function, then it found a) current namespace, b)using namespace and c)all previous declaration. At these time only  $f(T)$  is visible, Then add it into overload options. 2) When  $T$  is knowing, It will search all options in the boost namespace, **At this time, it will not search framework namespace again**

```

1. namespace framework { // library 1
2.     template <typename T>
3.     void f(T) { puts("master"); }
4.
5.     template <typename T>
6.     void process(T v) { f(v); }
7. }
8.
9. namespace boost { // library 2

```

```

10. template <typename T>
11. struct optional {};
12. }
13.
14. namespace framework {
15.     template <typename T>
16.     void f(boost::optional<T>) { puts("optional<T>"); }
17.
18.     inline void f(boost::optional<bool>) { puts("optional<bool>"); }
19. }
20.
21. int main() {
22.     int i = 0;
23.     boost::optional<int> oi;
24.     boost::optional<bool> ob;
25.
26.     framework::process(i); //output three "master"
27.     framework::process(oi);
28.     framework::process(ob);
29. }
```

4. p5, In instantiation, with help of ADL, boost namespace is searched and two other functions are added into overload options.

```

1. namespace framework{ // library 1
2.     template <typename T>
3.     void f(T) { puts("master"); }
4.
5.     template <typename T>
6.     void process(T v) { f(v); }
7. }
8.
9. namespace boost{ // library 2
10.     template <typename T>
11.     struct optional {};
12. }
13.
14. namespace boost { // some glue between 1 and 2
15.     template <typename T>
16.     void f(optional<T>) { puts("optional<T>"); }
17.
18.     inline
19.     void f(optional<bool>) { puts("optional<bool>"); }
20. }
21.
22. int main() {
23.     int i = 0;
24.     boost::optional<int> oi;
25.     boost::optional<bool> ob;
26.
27.     framework::process(i);
28.     framework::process(oi);
29.     framework::process(ob);
30. }
```

5. p6 template specification happen in the end and after overload.

```

1. namespace framework // library 1
2. {
3.     template <typename T>
4.     void f(T) { puts("master"); }
```

```

5.   template <typename T>
6.     void process(T v) { f(v); }
7.   }
8.
9.
10. namespace boost{ // library 2
11.   template <typename T>
12.   struct optional {};
13. }
14.
15. namespace framework{ // some glue between 1 and 2
16.   template <>
17.   void f<boost::optional<bool>>(boost::optional<bool>)
18.   { puts("optional<bool>"); }
19. }
20.
21. int main() {
22.   int i = 0;
23.   boost::optional<bool> ob;
24.
25.   framework::process(i);
26.   framework::process(ob);
27. }
```

## 6. Another interesting example:

```

1 template<typename T>
2 void f(T i){
3   cout<<"template "<<i<<endl;
4 }
5
6 //template<> void f<short>(short s);
7 void g(void){
8   //int s = 5; //output template
9   short s = 5;
10  //uncomment above specification, call specification version.
11
12  // Specification after initialization.
13  f(s);
14 }
15
16 template<> void f<short>(short s){
17   cout<<"specification "<<s<<endl;
18 }
19 void f(short s){
20   cout<<"short "<<s<<endl;
21 }
22
23 int main(){
24   g();
25 }
```

- (a) put line 19 and line 16 two functions before function g, it will call regular exact mathch funciton void f(short).
- (b) put void f(short) after function g, it will can full specification version
- (c) comment line 6(specification decalaration). report error:  
error: specialization of 'void f(T)' [with T = short int] after instantiation. Because it has instantiation f(short) version from generic template, you can't specialize it any more.

- (d) Although exact match and specification have higher order, but you must make it visible by declaring them before the caller.

## 8.5 template and inheritance

- Non-Template class Derived from Template Base class specification

```
1. class u8toU16 : public std::codecvt<wchar_t, char, std::mbstate_t>{
2. ...
3. };
```

- Template class Deriving from the non-template class

```
1. class Base
2.
3. template<typename T>
4. class Derived: public Base
```

- template class can be used as base class and used as a component class. So we can inherit from a template class

### 1. A basic example

```
1. template< typename Type>
2. class SpecialStack: public Stack<Type>{
3.     Array<Type> array;
4. }
```

### 2. If an enum is used by several member functions of the std::codecvt template class, and does not relate to the template parameters. Hence it can exist in a separate base class.

```
1. template<typename T>
2. class codecvt_base{
3. public:
4.     enum result {ok, partial, error, noconv};
5. };
```

### 3. You also can extending the derived class

```
1. template< typename T>
2. class Base
3.
4. template<typename T, typename U>
5. class Derived: public Base<T>
```

### 4. Factor parameter-independent code out of templates. detail can be found in effective item 44. That is to avoid code bloatting.

```
1. template<typename T> // size-independent base class for
2. class SquareMatrixBase { // square matrices
3. protected:
4. ...
5.     void invert(std::size_t matrixSize); // invert matrix of the given size
6. ...
7. };
8.
9. template<typename T, std::size_t n>
10. class SquareMatrix: private SquareMatrixBase<T> {
```

```

11. private:
12.   using SquareMatrixBase<T>::invert; // make base class version of invert
13.   // visible in this class; see Items 33
14.   // and 43
15. public:
16.   ...
17.   void invert() { invert(n); } // make inline call to base class
18. }; // version of invert

```

### 5. Specializing the base class

```

1. template< typename T>
2. class Base
3.
4. template<typename T>
5. class Derived: public Base<int>

```

- Parameterised inheritance

1. Basic example

```

1. template<typename T>
2. class Derived: public T

```

2. Two famous idioms are Mixin and Policy, can be found below sections. In this chapter, we only focus on syntactic level, not semantic level.

- It can be used as private inheritance. An example can be found in policy section.
- public inheritance. A good introduction can be found in "modern c++ design" first chapter, about creator template class.**public inheritance means the derived class want to expose the public interface in base class**
- public inheritance will expose interface to outside, just like Create() function. private inheritance just implement it.

```

1. template <class T>
2. struct OpNewCreator{
3.   static T* Create(){
4.     return new T;
5.   }
6. };
7. template <class T>
8. struct MallocCreator{
9.   static T* Create(){
10.    void* buf = std::malloc(sizeof(T));
11.    if (!buf) return 0;
12.    return new(buf) T;
13.  }
14. };
15. // Library code
16. template <class CreationPolicy>
17. class WidgetManager : public CreationPolicy{
18. ...
19. };
20. // Application code
21. typedef WidgetManager< OpNewCreator<Widget> > MyWidgetMgr;

```

- private inheritance is not the first choice, but the last choice For above example, you don't need private inheritance, you can use previous section "Templates Parameterized by independent class". It's much clear. If policy class has protect member function which we want to use, at this time. private inheritance is our last choice.

## 8.6 template common idiom

- Some common used idioms in std:

1. concept
2. traits
3. tag dispatching
4. adaptors
5. type generators
6. object generators
7. policy classes

### 8.6.1 policy, mixin and CRTP

These three patterns are associated, so I put them together.

#### 8.6.1.1 policy

- The Policy class can be thought as an optional of multi-inheritance.

```

1. template <typename OutputPolicy, typename LanguagePolicy>
2. class HelloWorld : private OutputPolicy, private LanguagePolicy{
3. public:
4.     void run() const{
5.         // Two policy methods
6.         print(message());
7.     }
8. };
9.
10. class OutputPolicyWriteToCout{
11. protected:
12.     template<typename MessageType>
13.     void print(MessageType const &message) const{
14.         std::cout << message << std::endl;
15.     }
16. };
17.
18. class LanguagePolicyEnglish{
19. protected:
20.     std::string message() const{
21.         return "Hello ,_World!";
22.     }
23. };
24.
25. class LanguagePolicyGerman{
26. protected:
27.     std::string message() const{
28.         return "Hallo_Welt!";
29.     }
30. };

```

- Client code is below:

```

1. int main() {
2.     /* Example 1 */
3.     typedef HelloWorld<OutputPolicyWriteToCout,
4.                         LanguagePolicyEnglish> HelloWorldEnglish;
5.
6.     HelloWorldEnglish hello_world;
7.     hello_world.run(); // prints "Hello, World!"
8.
9.     /* Example 2 * Does the same, but uses another language policy */
10.    typedef HelloWorld<OutputPolicyWriteToCout,
11.                         LanguagePolicyGerman> HelloWorldGerman;
12.
13.    HelloWorldGerman hello_world2;
14.    hello_world2.run(); // prints "Hallo Welt!"
15. }
```

- If this polymorphism can be known at compile time, we can use "static polymorphism". Use template parameter inject a class with static method. They can be inline and without any runtime overhead.

```

1. // char_traits :eq
2. #include <iostream>      // std::cout
3. #include <string>        // std::basic_string, std::char_traits
4. #include <cctype>         // std::tolower
5. #include <cstddef>        // std::size_t
6.
7. // traits with case-insensitive eq:
8. struct custom_traits: std::char_traits<char> {
9.     static bool eq (char c, char d) {
10.         return std::tolower(c)==std::tolower(d);
11.     }
12.
13. // some (non-conforming) implementations of basic_string::find
14. // call this instead of eq:
15. static const char* find (const char* s, std::size_t n, char c)
16. { while( n-- && (!eq(*s,c)) ) ++s; return s; }
17. };
18.
19. int main (){
20.     std::basic_string<char,custom_traits> str ("Test");
21.     std::cout << "T found at position " << str.find('t') << '\n';
22. }
```

- This kind of examples can be found a lot in the STL, most of time by a kind of functor. Basic idea is the same.
- All above technologies are part of generic programming. A good and more example about generic programming examples can be found in "Modern C++ design" a book. The first chapter introduce policy pattern. The second chapter introduce a lot of ways to manage Type information at compile time.

### 8.6.1.2 Mixin

- **Mixin refers to the following idiom - a template class that is parameterized on its base class.** Here Mixin classes are template classes that define a generic behaviour, and are designed to inherit from the type you wish to plug their functionality onto.

- For example, `RepeatPrint` is Mixin class, T is the type you wish to plug their functionality onto.

```

1. template<typename T>
2. class RepeatPrint : T{
3.     explicit RepeatPrint(T const& printable) : T(printable) {}
4.     void repeat(unsigned int n) const{
5.         while (n-- > 0){
6.             this->print();
7.             //Make sure printable has print method
8.         }
9.     }
10. };

```

- Supposing that we have an existing type which has print method.

```

1. class Name{
2. public:
3.     Name(std::string firstName, std::string lastName)
4.         : firstName_(std::move(firstName))
5.         , lastName_(std::move(lastName)) {}
6.
7.     void print() const{
8.         std::cout << lastName_ << ", " << firstName_ << '\n';
9.     }
10.
11. private:
12.     std::string firstName_, lastName_;
13. };

```

- We can use it two ways:

```

1. //method 1, we use object generator pattern
2. template<typename T>
3. RepeatPrint<T> rpGen(T const& printable){
4.     return RepeatPrint<T>(printable);
5. }
6. Name ned("Eddard", "Stark");
7. rpGen(ned).repeat(10);
8.
9. //method 2, we need to specify type explicitly
10. RepeatPrint<Name> rp(ned);
11. rp.repeat(10)

```

### 8.6.1.3 CRTP

- The basic syntax is Templates Parameterised by a Derived Class.

```

1. template <class T>
2. struct Base{
3.     void interface(){
4.         // ...
5.         static_cast<T*>(this)->implementation();
6.         // ...
7.     }
8.
9.     static void static_func(){
10.         // ...
11.         T::static_sub_func();
12.         // ...

```

```

13.     }
14. };
15.
16. struct Derived : Base<Derived>{
17.     void implementation();
18.     static void static_sub_func();
19. };

```

- A easy way of "call back" is function pointer, but it can't be used inline. so There is efficiency problem.
- You can use inheritance, It's runtime polymorphism, and has some runtime overhead.
- There are two common usages from CRTP:

1. Static polymorphism. It avoids runtime polymorphism cost.

```

1. template <typename T>
2. class Amount{
3.     public:
4.         double getValue() const {
5.             return static_cast<T const*>(*this).getValue();
6.             //static polymorphism
7.         }
8.     };
9.
10. class Constant42 : public Amount<Constant42>{
11. public:
12.     double getValue() const {return 42;}
13. };
14.
15. template<typename T>
16. void print(Amount<T> const& amount){
17.     //use base class reference
18.     std::cout << amount.getValue() << '\n';
19. }
20.
21. Constant42 c42;
22. print(c42);

```

2. Adding Functionality

```

1. template <typename T>
2. struct counter{
3.     static int objects_created;
4.     counter() { ++objects_created; }
5. };
6.
7. template <typename T> int counter<T>::objects_created( 0 );
8.
9. class X : counter<X>{
10.     // ...
11. };
12.
13. X x;
14. cout << X::objects_created << endl;

```

#### 8.6.1.4 A practical example

- Consider that there is a Task Manager framework. It is expected that there will be some common bits of functionality that should be reused across task implementations. I have selected task execution timing and logging of start and completion messages to serve as examples. We use six different ways to implement it.

- method 1, use Mixin

- The function name (Execute) can be different. Here, we use the same name because we can recursive use it. such as: `LoggingTask< TimingTask< MyTask > > Task;`
- A detail can be found "C++ Mixins - Reuse through inheritance is good... when done the right way" and "Mixin Classes: The Yang of the CRTP"

```

1. class MyTask{
2. public:
3. void Execute() {
4.     std::cout << "This is where the task is executed" << std::endl;
5. }
6. ;
7.
8. template< class T >
9. class LoggingTask : public T{
10. public:
11.     explicit LoggingTask(T const& t) : T(t) {}
12.
13.     void Execute() {
14.         std::cout << "LOG: The task is starting --" << std::endl;
15.         T::Execute();
16.         std::cout << "LOG: The task has completed --" << std::endl;
17.     }
18. };
19.
20. LoggingTask< MyTask > t;
21. t.Execute();

```

- if you have a MyTask value, you can build a helper function to build Mixin class

```

1. template<typename Task>
2. LogTask<Task> getLogTask(Task const& task) {
3.     //Call the below explicit ctor in LogTask class
4.     return LogTask<Task>(task);
5. }
6.
7. getLogTask(task).Execute //with log functions

```

- method 2, use CRTP

- Use base class extend and reuse function(Execute\_imp)
- Expose Execute interface to client by base class
- Client can directly use MyTask.

```

1. template< class Task >
2. class LoggingTask {
3. public:
4.     void Execute() {
5.         std::cout << "LOG: The task is starting --" << std::endl;

```

```

6.     static_cast<Task const&>(*this).Execute_imp() ;
7.     std::cout << "LOG: The task has completed--" << std::endl ;
8.   }
9. };
10.
11. class MyTask: public LoggingTask<MyTask> {
12. public:
13.   void Execute_imp() {
14.     std::cout << "... This is where the task is executed..." << std::endl ;
15.   }
16. };
17.
18. MyTask t ;
19. t . Execute() ;

```

- method 3, template Method

1. No template, just use run time polymorphism
2. Base class define a basic flow of action, some part of action is override in the Child class,  
That is Template Method pattern in design pattern.

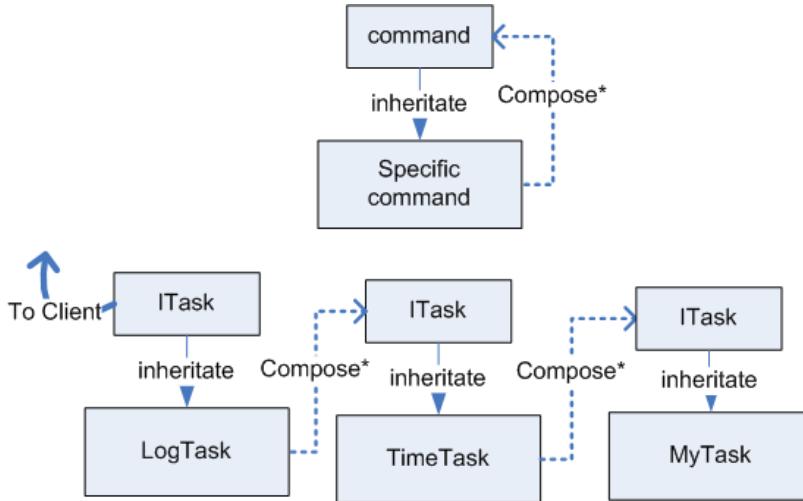
```

1. class BaseLoggingTask{
2. public:
3.   void Execute() {
4.     std::cout << "LOG: The task is starting--" << std::endl ;
5.     OnExecute();
6.     std::cout << "LOG: The task has completed--" << std::endl ;
7.   }
8.   virtual void OnExecute() = 0;
9. };
10.
11. // Concrete Task implementation that
12. //reuses the logging code of the BaseLoggingTask
13. class MyTask : public BaseLoggingTask{
14. public:
15.   virtual void OnExecute() {
16.     std::cout << "... This is where the task is executed..." << std::endl ;
17.   }
18. };

```

- method 4, command pattern

1. That is command pattern.
2. For Logging Task, you can see the class **includes and derived from ITask at the same time**.
3. Command pattern is illustrated by below figure:



```

1. class LoggingTask : public ITask{
2.     ITask* task_;
3. public:
4.     LoggingTask( ITask* task ) : task_( task ){ }
5.
6.     ~LoggingTask() {
7.         delete task_;
8.     }
9.
10.    virtual void Execute(){
11.        std::cout << "LOG: The task is starting--"
12.                    << GetName().c_str() << std::endl;
13.        if( task_ ) task_->Execute();
14.        std::cout << "LOG: The task has completed--"
15.                    << GetName().c_str() << std::endl;
16.    }
17. };
18.
19. class MyTask : public ITask{
20. public:
21.     virtual void Execute(){
22.         std::cout << "... This is where the task is executed..." << std::endl;
23.     }
24. };
25.
26. ITask* t = new LoggingTask(new MyTask());
27. t->Execute();
  
```

- method 5, mixin many mixin

```

1. template< class T >
2. class LoggingTask : public T{
3. public:
4.     void Execute(){
5.         std::cout << "LOG: The task is starting--"
6.                     << GetName().c_str() << std::endl;
7.         T::Execute();
8.         std::cout << "LOG: The task has completed--"
9.                     << GetName().c_str() << std::endl;
10.    }
11. };
12.
  
```

```

13. template< class T >
14. class TimingTask : public T{
15.     Timer timer_;
16. public:
17.     void Execute(){
18.         timer_.Reset();
19.         T::Execute();
20.         double t = timer_.GetElapsedTimeSecs();
21.         std::cout << "Task Duration:" << t << "seconds" << std::endl;
22.     }
23. };
24.
25. class MyTask{
26. public:
27.     void Execute(){
28.         std::cout << "... This is where the task is executed..." << std::endl;
29.     }
30. };
31.
32. typedef LoggingTask< TimingTask< MyTask > > Task;
33. Task t4;
34. t4.Execute();

```

- method 6, policy base

```

1. class coutLog{
2.     void beginLog(){
3.         cout << "LOG: The task is starting--" << endl
4.     }
5.
6.     void endLog(){
7.         cout << "LOG: The task has completed--"
8.     }
9. }
10.
11. template<typename LogPolicy, typename TimePolicy>
12. class Task: private LogPolicy, TimePolicy{
13.     void Execute(){
14.         beginLog();
15.         beginTime();
16.         std::cout << "This is where the task is executed";
17.         endTime();
18.         endLog();}

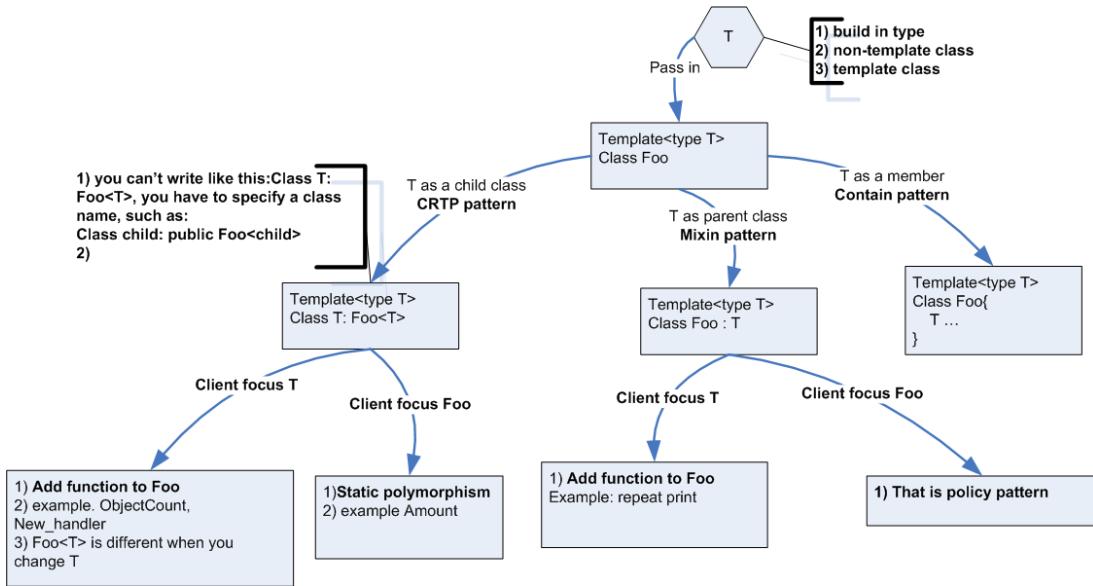
```

- summary:

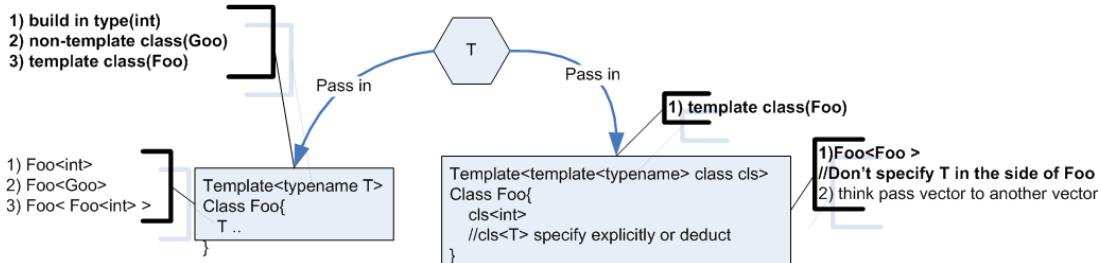
1. Policy can be used when you log and time function change a lot
2. Mixin and command can be composed to form multi-actions.
3. template is more efficient than inheritance.
4. Command pattern can also be thought as composite pattern.
5. Mixin will not change current class, but CRTP changes the current class interface.

#### 8.6.1.5 summary

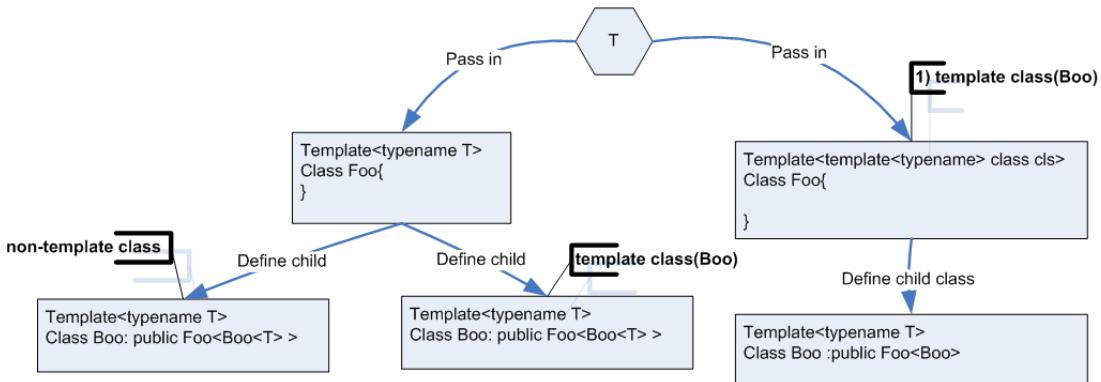
- Basic idea is illustrated below:



- For Contain pattern, we can use template template parameter.



- For CRTP pattern, we can use template template parameter.



## 8.6.2 tag dispatch

- A famous usage of tag dispatch is iterator

- Pay attention to `iterator_traits`. Why do we use it here? It's also a template. With specialization, it also support common pointer type.
- Client side may or may not be template. For advance, it should be template, because it need template parameter `InputIterator`.
- There is some tag type which need to be defined. And you should use these tags in the functions which you want to dispatched by overload resolve in compiling time.

```

1. struct input_iterator_tag { };
2. struct random_access_iterator_tag { };

3.
4. namespace detail {
5.     template <class InputIterator, class Distance>
6.     void advance_dispatch(InputIterator& i, Distance n, input_iterator_tag) {
7.         while (n--) ++i;
8.     }

9.
10.    template <class RandomAccessIterator, class Distance>
11.    void advance_dispatch(RandomAccessIterator& i, Distance n,
12.                          random_access_iterator_tag) {
13.        i += n;
14.    }
15. }

16.
17. template <class InputIterator, class Distance>
18. void advance(InputIterator& i, Distance n) {
19.     typename iterator_traits<InputIterator>::iterator_category category;
20.     detail::advance_dispatch(i, n, category);
21. }
```

- The second implementation of tag dispatch is use integral\_constant

1. std::true\_type use integral\_constant(type generator)to define a new type. is\_arithmetic derived form true\_type or false\_type

```

1. std::integral_constant<bool, true>
2.
3. template< class T >
4. struct is_arithmetic : std::integral_constant<bool,
5. std::is_integral<T>::value ||
6. std::is_floating_point<T>::value> { };
```

2. client side is also template
3. is\_arithmetic is a kind of trait, When you input T, it return a unnamed type, but it inherited from ture\_type, Then use overload resolving.
4. overload funciton can return different types. foo's return type is 'int' if it calls the 'std::true\_type' overload and 'double' if it calls the 'std::false\_type' overload. So here we use auto as function return type.

```

1. template <typename T>
2. int foo_impl(T value, std::true_type) {
3.     // Implementation for arithmetic values
4. }
5.
6. template <typename T>
7. double foo_impl(T value, std::false_type) {
8.     // Implementation for non-arithmetic values
9. }
10.
11. template <typename T>
12. auto foo(T value) {
13.     // Calls the correct implementation function,
14.     return foo_impl(value, std::is_arithmetic<T>{});
15. }
```

- The third method: build your own trait, it define a value
  - build you own trait has\_serialize, it includes a bool member ::value
  - You can use decltype, declval and constexpr to build this trait.
  - From this trait, it return value, not a type. So you need use enable\_if template in the client code. change the value to a type, then use the type to overload resolving.
  - For false, enable\_if is empty define.

```

1. template <bool, typename T = void>
2. struct enable_if
3. {};
4.
5. template <typename T>
6. struct enable_if<true, T> {
7.     typedef T type;
8. };

```

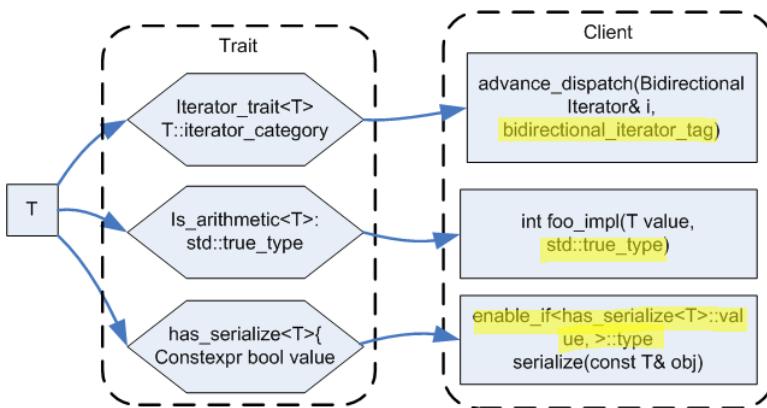
- an example which use enable\_if. has\_serialize trait has been introduced in the type trait section, you can see the detail there.

```

1. template <class T>
2. enable_if<has_serialize<T>::value, std::string>::type
3. serialize(const T& obj){
4.     return obj.serialize();
5. }
6.
7. // Contra-SFINAE to avoid ambiguity
8. template <class T>
9. enable_if<!has_serialize<T>::value, std::string>::type
10. // Watch out for the "!"
11. serialize(const T& obj){
12.     return to_string(obj);
13. }

```

- summary:



### 8.6.2.1 enable\_if

- Note SFINAE. When we make the call do\_stuff(25), the compiler selects the first overload: since the condition std::is\_integral<int> is true, the specialization of struct enable\_if for true is used, and its internal type is set to int. The second overload is omitted because without the true specialization (std::is\_class<int> is false) the general form of struct enable\_if is selected, and it doesn't have a type, so the type of the argument results in a substitution failure.

```

1. template <typename T>
2. void do(typename enable_if<std::is_integral<T>::value, T>::type &t) {
3.     // an implementation for integral types (int, char, unsigned, etc.)
4. }
5.
6. template <typename T>
7. void do(typename enable_if<std::is_class<T>::value, T>::type &t) {
8.     // an implementation for class types
9. }

```

- This technology has been used widely inside of vector.

```

1. template <typename T>
2. class vector {
3.     vector(size_type n, const T val);
4.
5.     template <class InputIterator>
6.     vector(InputIterator first, InputIterator last);
7.     ...
8. }
9.
10. template <class _InputIterator>
11.     vector(_InputIterator __first,
12.             typename enable_if<__is_input_iterator<_InputIterator>::value &&
13.             !__is_forward_iterator<_InputIterator>::value &&
14.             ... more conditions ...
15.             _InputIterator>::type __last);

```

- use a few ways to implement type trait, then use enable\_if to use type trait to get generic programming(static polymorphism); That is generic programming basic idea.

1. use true\_type or false\_type tag.

```

1. template <typename T>
2. auto get_value(T t, std::true_type) {
3.     return *t;
4. }
5.
6. template <typename T>
7. auto get_value(T t, std::false_type) {
8.     return t;
9. }
10.
11. template <typename T>
12. auto get_value(T t) {
13.     return get_value(t, std::is_pointer<T>{});
14. }

```

2. use SFINAE

```

1. template <typename T,
2.           std::enable_if_t<std::is_pointer_v<T>, int> = 0>
3. auto get_value(T t) {
4.     return *t;
5. }
6.
7. template <typename T,

```

```

8. typename std::enable_if_t<!std::is_pointer_v<T>, int> = 0>
9. auto get_value(T t) {
10.    return t;
11. }
```

3. use `constexpr` if:

```

1. template <typename T>
2. auto get_value(T t) {
3.    if constexpr (std::is_pointer_v<T>) {
4.        return *t;
5.    } else {
6.        return t;
7.    }
8. }
```

- A good article is "SFINAE and enable\_if"

### 8.6.3 type erasure and concept

#### 8.6.3.1 function

- function template support **copyable** and **callable**.

```

1. double add(double a, double b){
2.     return a + b;
3. }
4.
5. struct Sub{
6.     double operator()(double a, double b){
7.         return a - b;
8.     }
9. };
10.
11. double multThree(double a, double b, double c){
12.     return a * b * c;
13. }
14.
15. using namespace std::placeholders;
16. std::map<const char, std::function<double(double, double)>
17. { '+' , add }, // (2)
18. { '-' , Sub() }, // (3)
19. { '*' , std::bind(multThree, _1, _1, _2) }, // (4)
20. { '/' , [](double a, double b){ return a / b; }}}; // (5)
21.
22. cout << dispTable['+'](3.5, 4.5) << dispTable['-'](3.5, 4.5)
23. << dispTable['*'](3.5, 4.5) << dispTable['/'](3.5, 4.5);
```

- If you want to check if a variable of type `std::function` is currently holding a valid function, you can always treat it like a boolean:

```

1. std::function<int ()> func;
2. .....
3. if (func) { // if we did have a function, call it
4.     func();
5. }
```

- function can be used **CALL BACK** as a function argument. At this time, you can use reference, but you'd better pay attention reference dangling problem.

```

1. void run_within_for_each(std::function<void (int)> func){
2.     vector<int> numbers{ 1, 2, 3, 4, 5, 10, 15, 20, 25 };
3.     for_each(numbers.begin(), numbers.end(), func);
4. }
5.
6. void fun(int x){    //1) function pointer
7.     cout << x << endl;
8. }
9.
10. auto lambda1 = [] (int y){ //4) lambda
11.     cout << y << endl;
12. };
13.
14. run_within_for_each(fun);
15. run_within_for_each(lambda1);

```

- One big advantage of std::function over templates is that if you write a template, you need to put the whole function in the header file, whereas std::function does not. This can really help if you're working on code that will change a lot and is included by many source files.
- A good article: google "Should I use std::function or a function pointer in C++?" another is "How is std::function implemented?"
- In this way you can write a type-erased wrapper for any Whatever callbacks, counters, output streams, input generators.

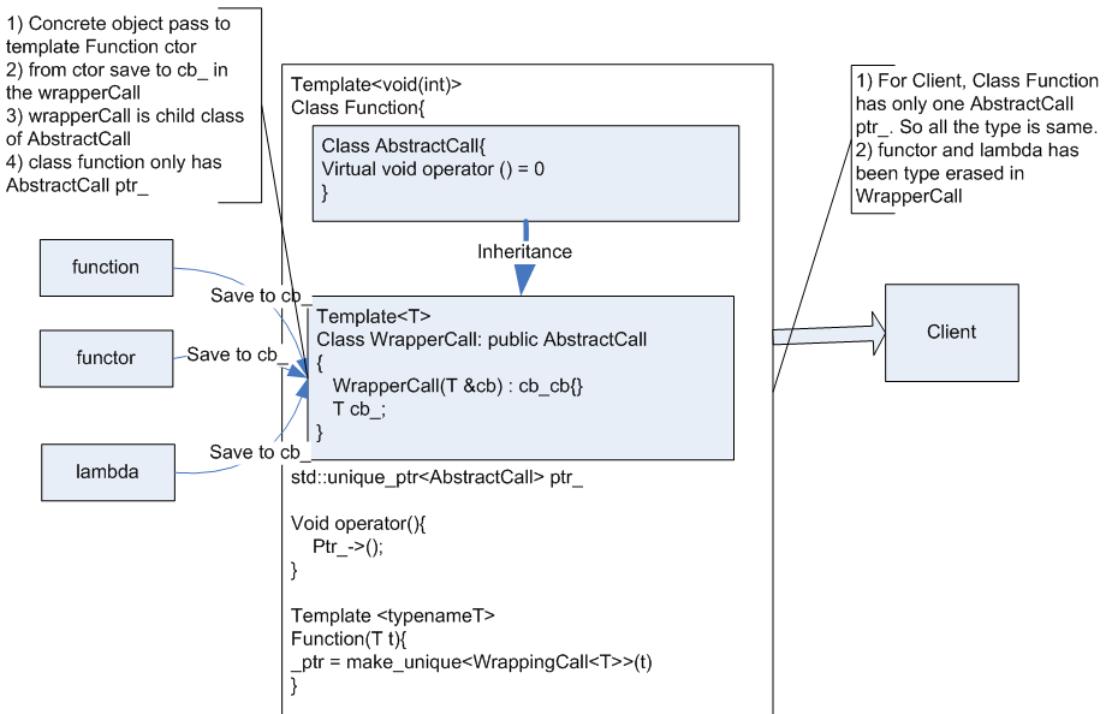
#### 8.6.3.2 type erasure

- When you implement type erasure, it always starts with making a list of the things you want to be able to do with your type-erased object - call it, destroy it, copy it, and so on. For example, A std::function affords copying and calling. A std::any affords copying, but not calling. A unique\_function affords calling and moving, but not copying.
- Each affordance in your list turns into a virtual member function of AbstractWhatever, which will be overridden by WrappingWhatever<T> appropriately for T. Finally, the top-level Whatever will store a unique\_ptr<AbstractWhatever> ptr\_ (and/or an SBO buffer), and provide a clean non-virtual interface implemented completely in terms of calling virtual member functions on \*ptr\_.
- A **concept** is a set of requirements consisting of valid expressions, associated types, invariants, and complexity guarantees. A type that satisfies the requirements is said to **model** the concept. A concept can extend the requirements of another concept, which is called **refinement**.
  1. Valid Expressions are C++ expressions which must compile successfully for the objects involved in the expression to be considered models of the concept. For example, an **Iterator** x is expected to support the expressions `++x` and `*x`.
  2. Associated Types are types that are related to the modeling type in that they participate in one or more of the valid expressions. Typically associated types can be accessed either through typedefs nested within a class definition for the modeling type, or they are accessed through a traits class. For example, an iterator's value type is associated with the iterator through `std::iterator_traits`
  3. Invariants are run-time characteristics of the objects that must always be true, that is, the functions involving the objects must preserve these characteristics. The invariants often

take the form of pre-conditions and post-conditions. For example, Forard iterator is copied, the copy and original must compare equal.

4. Complexity Guarantees are maximum limits on how long the execution of one of the valid expressions will take, or how much of various resources its computation will use.

- A good article is "What is Type Erasure" A detail implementation can be found in <http://blog.bitfoc.us/p>
- An implementation is below:



- An implementation code is below:

```

1. struct Plus1 {
2.     int call(int x) const { return x+1; }
3. };
4.
5. struct AbstractCallback {
6.     virtual int call(int) const = 0;
7.     virtual ~AbstractCallback() = default;
8. };
9.
10. template<class T>
11. struct WrappingCallback : AbstractCallback {
12.     T cb_;
13.     explicit WrappingCallback(T &&cb) : cb_(std::move(cb)) {}
14.     int call(int x) const override { return cb_(x); }
15. };
16.
17. struct Callback {
18.     std::unique_ptr<AbstractCallback> ptr_;
19.
20. template<class T>
21. Callback(T t) {
22.     ptr_ = std::make_unique<WrappingCallback<T>>(std::move(t));
23. }
24. int operator()(int x) const {

```

```
25.     return ptr_->call(x);  
26. }  
27. };  
28.  
29. int run_twice(const Callback& callback) {  
30.     return callback(1) + callback(1);  
31. }  
32.  
33. int y = run_twice([](int x) { return x+1; });  
34. assert(y == 4);
```

# Chapter 9

## STL

### 9.1 Basic

- Effective STL item 48. Always include the proper headers.
  1. Include corresponding header when you use container.
  2. All but four algorithms are declared in <algorithm>. inner\_product, adjacent\_difference, partial\_sum and accumulate are in the <numeric>.
  3. Special kinds of iterators , including istream\_iterators are in the <iterator>
  4. Standard functors, eg less<T> and functor adapter not1, bind2nd are declared in <functional>. They are not recommended to use in C++11. Because C++ 11 introduce **function** and **bind** template.
- Containers, iterators, functions objects, and algorithms four parts. **Use functor customize algorithms, then apply algorithm on Containers by iterators.** Such as algorithm find, you can input two Iterators into the it. Find don't care what container which he will look for a value in it. Find() in fact is a template function, it doesn't use inheritance to make generic programming, but use template. All the idea is explained here:  
[http://www.cplusplus.com/reference/algorithm/for\\_each/](http://www.cplusplus.com/reference/algorithm/for_each/)

```
1. struct Sum{  
2.     Sum(): sum{0} { }  
3.     void operator()(int n) { sum += n; }  
4.     int sum;  
5. };  
6. ////////////  
7. std::vector<int> nums{3, 4, 2, 8, 15, 267};  
8.  
9. std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });  
10. // g++ -std=c++11 compile lambda syntax.  
11.  
12. Sum s = std::for_each(nums.begin(), nums.end(), Sum());  
13. // It return a Sum struct.  
14. cout<<s.sum<<endl;
```

## 9.2 Container

### 9.2.1 Basic knowledge

- For all container, **Copy in, Copy out**. In C++11, you can use `emplace_back` to build object directly on the vector to avoid copy.

```

1. int j = 10;
2. vector<int> vc;
3. vc.push_back(j); // not j, but copy of j to vc
4.
5. int i = vc.pop_back(); //
6. i = -99 // when you modify i, not effect on vc

```

- Even copy in, copy out. vector is better than array in C language. Because it use heap memory, if you want to use stack memory, you can use `std::array`.

```

1. Foo farray[50]; // default ctor has been called 50 times.
2. Foo *parray = new Foo[50]; Same, ctor will be called 50 times.
3.
4. vector<Foo> fvc; // no default ctor has been called.
5. fvc.reserve(50);

```

- Anytime you want to write new [...] just for allocate dynamic array, using a vector or string instead, and using reserve() to allocate enough space.**
- Copy in and copy out can cause efficient problem. Furthermore, It brings another problem. when you inserting a derived class object into a container of base class objects is always error(slicing error). In order to fight this problem, you can use container of pointer. But it will cause resource leaking problem if you forget or an exception is thrown. So a better choice is use smart pointer. But don't use `auto_ptr` in any container, It's prohibited in c++, and produce a compiler error, see effective stl item 8.
- All container is not thread-safe. It's your duty to make thread-safe, and thread-safe is OS depended. Detail can be seen effective STL item 12

```

1. getMutexFor(v);
2. // do something on v.
3. releaseMutexFor(v);

```

- There are three kinds of for-loop container syntax;

```

1. for (auto it = con.begin(), itend != con.end())
2.       it != itend ; ++it) {
3.     foo(*it);
4. }
5.
6. for_each(con.begin(), con.end(), [] (Element& e) {
7.   foo(e);
8. });
9.
10. for (auto & e : con) {
11.   foo(e);
12. }

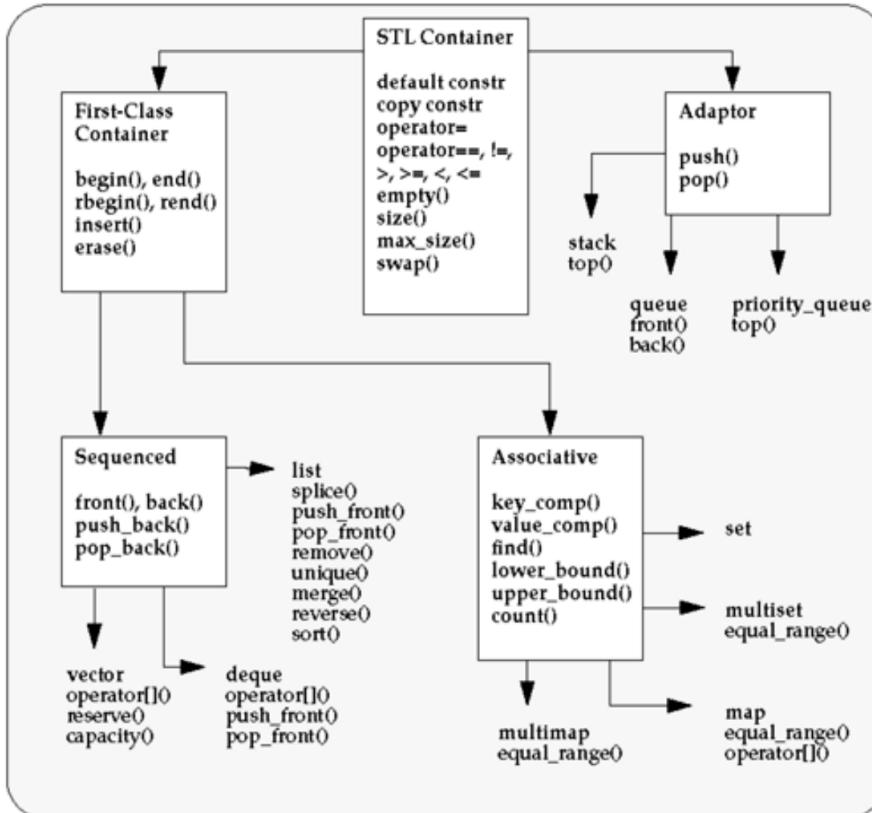
```

### 9.2.2 Basic classifications

#### 9.2.2.1 structure classification

- STL container: sequence container, associative container and containter adapter:

1. sequence: vector, list, string, deque
2. associative: map , multiset , set and multimap
3. adapter: stack, queue and priority\_queue.



1. If you can find same name member functions in container, prefer to use member function than generic algorithm. Such as **sort**, **merge**, **remove**, **reverse**, **unique** in **list**, and **find**, **count**, **low\_bound** in **set** and **map**.
2. all container support empty(), size(), max\_size(), and swap(). max\_size() is just theoretic value. (4 functions)
3. First class container includes begin(), end(), rbegin(), rend(). insert() and erase(). (6 functions)
4. sequence container support pop\_back(), push\_back(), front() and back() (4 functions) while associative don't.
5. vector support operator[], list support push\_front and pop\_front. and deque support both. (3 functions.) By now deque support (4+6+4+3 = 17 functions)**container(4)-> first container(6)->sequence container(4)->deque(3)**
6. vector support reserve, list has reverse(**not reserve**) and splice(). deque support nothing.
7. list support it's own version sort(), remove(), merge(), unique(). For other container, you can use the generic algorithm, Why list has its own? For sort(), list doesn't support random access iterator. For merge(), remove() and unique(). generic algorithm just use copy

method, but list has high efficient pointer implementation, So list offer its own version merge(), remove(), sort(), unique().

8. All first class container support begin() and end(). Only sequence containers support push\_front() or push\_back(). **begin is not equal front, begin can be used for all first class container, but front only be used for sequence container.**
  - (a) begin() and end() return iterator, and all first container support them.
  - (b) front() and back() return reference, and all sequenced container support them.
  - (c) push\_front() and push\_back() add element, and vector only support push\_back. deque and list support both.
9. Associative container support It's own find(), count(), lower\_bound(), upper\_bound() , equal\_range(). They share the same name with generic algorithm, Don't confuse them. When you deal with associative container, just use container member function, don't use generic algorithm. **Associative offer log-time lower\_bound, upper\_bound and equal\_range, but generic algorithm just linear time.**
10. deque is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence. It doesn't guarantee continuity within memory, and higher constant factor cost than vector. Although both offer random access to elements and linear-time insertion and removal from middle of a sequence, the vector is faster. In my evernote, you can see a implementation of deque.
11. difference between vector and deque:
  - (a) vector has relocation problem, If vector has 1000, It probably need  $\log(1000) = 10$  times relocation. that is a little costly. On the contrary, deque just allocate a new place then insert pointer of new place to pointer map. It's relatively cheap.
  - (b) Elements in a deque are not contiguous in memory; vector elements are guaranteed to be. So if you need to interact with a plain C library that needs contiguous arrays, or if you care (a lot) about spatial locality, then you might prefer vector.
  - (c) For stack and queue default use deque as container inside. Why, because for a large amount of element, vector has relocation problem,
12. All the container adapter support push(), pop(). and stack and priority\_queue support top(). They don't have any iterator.
13. If you want strongly error-safe code, such as transnational semantics for inserting and erasing, or need to minimize iterator invalidation, prefer a node-based container.
14. Using a vector for small list is almost always superior to using list. Even though inseriton in the middle of the sequence is a linear-time operation for vector and a constant-time operation for list. Vector usually outperforms list because of its better constant factor. **list's Big-Oh advantage doesn't kick in until data sizes get larger**
15. Store only values and smart pointers(unique\_ptr or shared\_ptr) in container.
  - (a) To contain objects even though they are not copyable or otherwise not value-like (e.g., DatabaseLocks and TcpConnections), prefer containing them indirectly via smart pointers (e.g., container< shared\_ptr<DatabaseLock> > and container<shared\_ptr<TcpConnection> >).
  - (b) Optional values. When you want a map<Thing, Widget>, but some Things have no associated Widget, prefer map<Thing, shared\_ptr<Widget> >.
  - (c) Index containers. To have a main container hold the objects and access them using different sort orders without resorting the main container, you can set up secondary

containers that "point into" the main one and sort the secondary containers in different ways using dereferenced compare predicates. But prefer a container of MainContainer::iterators (which are value-like) instead of a container of pointers.

- (d) To have a container store and own objects of different but related types, such as types derived from a common Base class, prefer container< shared\_ptr<Base> >. An alternative is to store proxy objects whose nonvirtual functions pass through to corresponding virtual functions of the actual object.

### 9.2.2.2 memory classification

- Another classification of container : contiguous-base and node-base.
  - 1. vector, string, and deque
  - 2. list and slist(linked list), set and map(balanced trees)
- Why we have this point of view.
  - 1. Because all the contiguous-base container has invalidation of iterator(pointer, reference) problem.
  - 2. All the node-base container only support bidirectional iterators.
  - 3. All the node-base container don't have reserve() function and need NOT to worry allocation problem. deque doesn't have reserve() function either
- About the iterator invalidation problem, First, you need to know the invalidation rules below:

#### **Insertion** – Sequence containers

1. vector: all iterators and references before the point of insertion are unaffected, unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated)
  2. deque: all iterators and references are invalidated, unless the inserted member is at an end (front or back) of the deque (in which case all iterators are invalidated, but references to elements are unaffected)
  3. list: all iterators and references unaffected
- Associative containers: [multi]set,map all iterators and references unaffected
  - Container adaptors: stack, queue and priority\_queue: inherited from underlying container.(Usually, we don't use iterator in stack, queue and priority\_queue. Maybe you don't need to worry about it. )

```

1. vecArr.insert( it + 2, 1, 200 );
2.
3. // Reinitialize the invalidated iterator to the begining.
4. it = vecArr.begin();
```

#### **Erasure** – Sequence containers

1. vector: every iterator and reference after the point of erase is invalidated
  2. deque: all iterators and references are invalidated, unless the erased members are at an end (front or back) of the deque (in which case only iterators and references to the erased members are invalidated)
  3. list: only the iterators and references to the erased element is invalidated
- Associative containers: [multi]set,map: only iterators and references to the erased elements are invalidated

- Container adaptors: stack, queue and priority\_queue: inherited from underlying container

```

1. auto it = std::find(vecArr.begin(), vecArr.end(), 5);
2. if(it != vecArr.end())
3.     vecArr.erase(it);
4.     it = vecArr.erase(it);
5.
6. // Now iterator 'it' is invalidated because it
7. // still points to old location, which has been deleted.
8. for(; it != vecArr.end(); it++)//Unpredicted Behavior
9.     std::cout<<(*it)<<"\u2022"; //Unpredicted Behavior

```

**swap** no swap() function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped.

**Resizing** vector, deque and list as per insert/erase

### 9.2.3 Usage

#### 9.2.3.1 Search in Container

- Distinguish among count, find, binary\_search, lower\_bound, upper\_bound, and equal\_range.

| <b>What You Want to Know</b>                                                    | <b>Algorithm to Use</b> |                   | <b>Member Function to Use</b> |                                   |
|---------------------------------------------------------------------------------|-------------------------|-------------------|-------------------------------|-----------------------------------|
|                                                                                 | On an Unsorted Range    | On a Sorted Range | With a set or map             | With a multiset or multimap       |
| Does the desired value exist?                                                   | find                    | binary_search     | count                         | find                              |
| Does the desired value exist? If so, where is the first object with that value? | find                    | equal_range       | find                          | find or lower_bound (see article) |
| Where is the first object with a value not preceding the desired value?         | find_if                 | lower_bound       | lower_bound                   | lower_bound                       |
| Where is the first object with a value succeeding the desired value?            | find_if                 | upper_bound       | upper_bound                   | upper_bound                       |
| How many objects have the desired value?                                        | count                   | equal_range       | count                         | count                             |
| Where are all the objects with the desired value?                               | find (iteratively)      | equal_range       | equal_range                   | equal_range                       |

1. For unsorted range, just generic `find()` or `find_if()` algorithm or `count`. return last if no find element.
2. For sorted range, Only four `binary_search`, `lower_bound`, `upper_bound`, and `equal_range`. are used on a sorted range.
3. For set or map, It has own version, `find()`, `count()` and `lower_bound`, `upper_bound`, and `equal_range`
4. Don't use generic `find()` algorithm on a sorted range, use `binary_search()`. `binary_search` just return bool, not position.
5. `find()` will return match iterator, if no match, it will return `end()`(for map or set). or last iterator(for generic algorithm). but `Lower_bound` will return a position anyway, match position or insert position(no match).
6. Lower bound: first element that is greater-or-equal. Upper bound: first element that is strictly greater. `equal_range` return a pair of (`lower_bound`, `Upper_bound`).

```

+- lb(2) == ub(2)      +- lb(6)      +- lb(8)
|      == begin()      | == ub(6)      | +- ub(8) == end()
V          V          V   V
+-----+-----+-----+-----+
| 3 | 4 | 4 | 4 | 5 | 7 | 7 | 7 | 7 | 8 |
+-----+-----+-----+-----+
^           ^           ^
|           |           |
+- lb(4)      +- ub(4)      +- lb(9) == ub(9) == end()

|- eq-range(4) -|

```

7. equal\_range if two iterators equal, (no found.). if two iterators distance  $\geq 1$ , (find one or more match, can replace find() or count() function.) .

```

1. std :: sort ( v.begin() , v.end() );
2. // 10 10 10 20 20 20 30
3. auto p =std :: equal_range ( v.begin() , v.end() , 20 );
4. for ( auto i = p.first ; i != p.second ; ++i )
5.     std :: cout << i->name << ' ';
6.
7. //or code below:
8. if( distance(p.first , p.second) >=1){
9.     cout<<"_found_match"<<endl;
10. }

```

### 9.2.3.2 Range

- Copy algorithm usage: 1) copy v2 to v1. 2) it use loop inside. **Almost all uses of copy where the destination range is specified using an insert iterator should be replaced with calls to range member function.**

```

1. v1. clear();
2. copy(v2.begin() , v2.end() , back_inserter(v1) );
3.
4. //prefer to use below three range memeber function
5. v1.insert(v1.end() , v2.begin() , v2.end() ); //better than copy
6. v1.assign(v2.begin() , v2.end() )
7. // if want to replace all value in v1.
8. vector<int> v1( v2.begin() , v2.end() );
9. // build from sractch.
10. v1.erase( v1.begin() , v1.begin() +5 );
11. //erage range.

```

- A good example to explain why use range member function is below: When you use insert, inside of insert, STL will get distance(n elements) of v2.begin() and v2.end(). Then , It will reserve and move the all element in v1 just once according to the distance. If you use copy, it use loop insert one by one, It will move all element in v1 n times. if no space, it will reallocate and copy old element, just like common vector do. So use range member function, can save you a lot of time in this example.

```

1. //don't use front_inserter() here ,
2. //because vector don't support push_front()
3. copy(v2.begin() , v2.end() ,
4.       inserter(v , v.begin() ) );
5.
6. // a better method is here .
7. v1.insert(v1.begin() , v2.begin() , v2.end());

```

- effective STL Item 5: Prefer range member functions to their single-element counterparts.

1. Range construction:
2. Range insertion:
3. Range erasure:
4. Range assignment:

```

1. container::container(inputIterator1, inputIterator2);
2. container::insert(insertPosition, inputIt1, inputIt2);
3. container::erase(inputIterator1, inputIterator2);
4. container::assign(inputIterator1, inputIterator2);

```

- The main reason for using assign is to copy data from one type of container to another.

1. For example, if you want to migrate the contents of an `std::set<int>` to an `std::vector<int>`, you can't use the assignment operator, but you can use `vector.assign(set.begin(), set.end())`.
2. Another example would be copying the contents of two containers holding different types that are convertible to one or the other; If you try to assign `std::vector<Derived*>` to an `std::vector<Base*>`, the assignment operator is insufficient.
3. different part from one container to another.

### 9.2.3.3 Erasure

- So, in short: generally speaking, you should not delete the items from the list while iterating through it, because the deletion may invalidate the iterator (and the program will possibly crash). If you are however completely sure that the items which you delete are not the values referenced by any of the iterators which you use at the moment of deletion, you may delete.
- Beware that for the other STL containers (e.g. vectors) the constraint is even more strict: deleting from the container invalidates not only iterators pointing to the deleted item, but possibly other iterators, too! So deleting from that containers while iterating through them is even more problematic.
- Choose carefully among erasing options: To eliminate all objects in a container that have a particular value.
  1. For vector, string or deque, use erase-remove. **Don't use for loop, it will invalid the iterator.**
  2. For a list, use its own remove or remove\_if.
  3. For associative container, use its erase member function.

```

1. vect.erase(remove(vect.begin(), vect.end(), 1963), vect.end());
2. list.remove(1963);
3. map.erase(1963);

```

- To eliminate all objects in a container that satisfy a particular predicate, 1)for vector, string or deque, use erase-remove. 2) for a list, use remove\_if, 3) for associative container, write loop being sure to postincrement your iterator. detail can be seen in effective STL item 9;

```

1. bool badValue(int x);
2. vc. erase(remove(vc. begin() , vc. end() , badValue) ,
3.           vc. end()) ;
4.
5. lsit. remove_if(badValue) ;
6. for (auto i = map. begin() ; i!=map. end() ; /* no ++i here*/) {
7.     if(badValue(*i)) map. erase(i++);
8.     // or i = map. erase(i) ;
9.     else ++i;
10. }
```

- To do something inside the loop (in addition to erasing objects); You can't use remove, just write a loop,

```

1. bool badValue(int x);
2. for (vect<int>::iterator i = vect. begin() ; i!=vect. end() ; ) {
3.     if(badValue(*i)) {
4.         i = vect. erase(i);
5.         ... do something else
6.     }
7.     else ++i;
8. }
```

#### 9.2.3.4 type definition in container

- value\_type in container.

```

1. vector<uint> vecs;
2. cout << sizeof(vecs. value_type) // error usage
3. cout<<sizeof(vector<uint>::value_type);
4. //Pay attention to :: because it's static class member.
```

- Having the commonly-used types available as a type on the container is useful when the container's type itself is unknown. For example, someone may want to write library code that works equally well with std::map and std::unordered\_map:

```

1. template<typename TMap>
2. void insert_default_pair(TMap& map)
3. {
4.     map. emplace(typename TMap::key_type() ,
5.                   typename TMap::mapped_type());
6. }
```

- Inside templated code, prefix value\_type with the keyword typename. Why, It depending on whether some identifier designates a type or a variable, e.g. T \* p may be a multiplication or a pointer declaration. Not explicitly marked as type by prefixing it with typename is considered a variable.

```

1. template <typename T>
2. class TSContainer {
3. private:
4.     T container;
5. public:
6.     void push(typename T::value_type& item){
7.             container. push_back(item);
8.     }
```

- summary of type in container
  1. complex type, save typing
  2. change a lot, good maintain
  3. Inside of a function
  4. previous two example, unknow container in template class or support two container at the same time, to reach generic purpose.

```

1. //1) it will save a lot of typing .
2. typedef vector< pair<int, string> > ComVec;
3. ComVec::value_type aaa;
4.
5. //2) int to float , but below code don't need change at all .
6. typedef vector< pair<float, string> > ComVec;
7.
8. //3) inside of funciton .
9. typedef std::vector< std::pair<int, std::string> > Record_t ;
10. // typedef is your good friend , reuse it below
11. Record_t k1;
12.
13. int find_it (std::string value , Record_t const& stuff){
14.   auto fit = std::find_if( stuff.begin() , stuff.end() ,
15.     [value](Record_t::value_type const& vt) -> bool
16.       { return vt.second == value ; });

```

### 9.2.3.5 Sizes

- For container, Call empty() instead of of checking size() against zero. it's very easy to understand it.
- for vector, size(), capacity() and max\_size() can be seen below: capacity is equal or bigger than size. So if you want to avoid allocate the unnecessary space. LLVM give a SmallVecotr<type, N> example, you can use N to specify a smaller vector to avoid waste.

```

1. std::vector<int> myvector ;
2. for (int i=0; i<100; i++) myvector.push_back(i);
3.
4. std::cout << (int) myvector.size() << '\n' ; // 100
5. std::cout << (int) myvector.capacity() << '\n' ; //128
6. std::cout << (int) myvector.max_size() << '\n' ; //1073741823

```

- Four confused conceptions:

```

1. capacity() //how many CAN hold
2. size() //how many are in NOW
3. resize(n) // forces the container to change to n,
4. //if n <size() , object in the end will be lost
5. // if n>size() , default ctor of element will be called .
6. // after resize(n) , size will return n .
7. // but resize don't change capacity
8.
9. reserve(n) //cause the container's capacity() to at lease n .

```

- You want to avoid allocation, 1), if you know the number, then use reserve, 2) if you don't know the number, you can reserve maximum space, then you can trim off any excess capaciy. you can use shrink\_to\_fit() function too. Just remember **string(s).swap(s)**

```

1. vector<class> v1;
2. v1.reserve(1000); v1.size(); // only 5
3.
4. vector<class>(v1).swap(v1);
5. //1) vector<class>(v1) copy ctor create a temp vector
6. //2) temp just copy real object, so it's capacity (maybe 8) is small.
7. //3) temp.swap(v1) then temp has 1000 space, v1 capacity is 8 now
8. //4) in the end of statement, temp destroy.
9. string(s).swap(s) // the same idea behind.

```

- Worthwhile note about vector: It will avoid allocate new space many times.

```

1. std::vector<int> v
2. for () { ...
3.   v.clear
4. } //good smell.
5.
6. for () { vector<int> v ... } //bad smell

```

- resize and reserve difference.

```

1. vector<Foo> vcFoo;
2. vcFoo.reserve(10);
3. vcFoo.resize(10); // will call Foo ctor 10 times
4. vcFoo[2] = foo; // will call Foo ctor and assignment 1 times
5. // if no reserve or resize. vcFoo[2] = foo will fail.

```

#### 9.2.3.6 Usages Tips

- When you use STL Container, you should realize that `typedef` are your best friends.

```

1. typedef vector<Foo, SpecialAllocator<Foo>> FooContainer;
2. typedef FooContainer::iterator FooIt;
3.
4. FooContainer fc1; // make you programming more clearly ,
5. FooContainer fc2; // you should use typedef more
6. FooIt it1;

```

- Never try to expect all the container has the same interface. Even a generic `erase()`, for sequence, It return next iterator,(because it will invalid the iterator). But for associative, it return `void(c++ 98)` and return next iterator(C++ 11). If you just want to change a container in the future, you should put a container into a class: `CustomCollection`. then hide it from the class client. The detail can be seen in Effective STL item 2.
- The standard associative container are implemented as balanced binary search trees. It's optimized for a mixed combination of insertions, erasures and lookup. But if that is dictionary, It can be fall into three distinct phases. setup, lookup, modify. and modify is not happen very often. lookup is very often. At this time, associative container is not best option. sorted vector also support log search time. 1) It use less space. 2) more space cause page fault in memory, then the same log search complex, but sorted vector is faster than associate container. **For dictionary, please use sorted vector.**
- Avoid using `vector<bool>`, use `deque<bool>` or `bitset`
- item 22 Avoid in-place key modification in set and multiset

- you can't change key in map because it's const default.

```
1. map.begin() -> first = 10 //compile error
```

- For set or map, you can modify non-key part.

```
1. iterator i = set.find(employee);
2. if( i != set.end())
3.     i->setTitle("manager") // it's also ok or
4.     const_cast<Employee &>(*i).setTitle("manager")
5.     // you must change it to a reference, then you can modify it.
6.     // if you just use const_cast<Employee>
7.     // this is not right. it will create a temporary obj
8.     //and then modify a temporary obj.
```

- If you want to modify the key part in set or map.

```
// 1) find the one
1. iterator i = se.find(employee);
2. //2) create temp one.
3. if(i != se.end())
4. Employee e(*i);
5. //3) modify
6. e.setKey("new_key")
7. //4) delete the old one and keep ;
8. se.erase(i++);
9. //5) insert new one
10. se.insert(i, e);
```

- Item19 effective STL, Understand the difference between equality and equivalence in associative Containers.

- equality is based on operator ==. equivalence is based on operator<. Because associate container, set, map, they must sort their elements, so they must use operator<. Then it use !if(a<b)&&!if(b<a) to define equivalence, and associate container use equivalence to decide if a object exist in container.
- if you don't have custom compare funciton, most of time equivalence is equal to equality, but if you define you specific compare function, you need to know below:
- It will cause container.find and generic algorithm find has different result

```
1. set<string, case_insensitive_compare> ss;
2. //ss has "AA";
3. ss.find("aa"); //return true;
4. find(ss.begin(), ss.end(), "aa") //return false
5. //find algorithm use operator == .
```

- It will lead to item 21 in effective STL. Always have comparison functions return false for equal values. (strict weak ordering )
- It will lead to item 20 in effective STL. For associative container of pointers, You need to specify comaprison types, You want to order by pointers or want to order by objects pointed by pointer.(Most of time, the second option is what we want)

```
1. struct stringLess : binary_function<const string*,
2.                               const string *, bool>{
3.     bool operator()(const string* s1,
4.                      const string * s2) {
5.     return *s1 < *s2;
```

```

6. }
7. }
8. set<*string ,  stringLess> ss;

```

### 9.2.4 string

- String::npos is the maximum possible length of the string. Equal the maximum value of an unsigned int.
- String is template specialization basic\_string<char>, from this point of view, you will know how to construct a w\_char string.
- There are a lot of ways that can be used construct a string object, you can see the reference , such as:

```

1. string(const char* s);
2. string(const char*, size_type n);
3. string(const string& str, size_type pos, size_type n = npos)
4. .....

```

- The standard containers define size\_type as a typedef to Allocator::size\_type (Allocator is a template parameter), which for std::allocator is defined to be size\_t. So for the standard case, they are the same. However, if you use a custom allocator a different underlying type could be used. So container::size\_type is preferable for maximum portability.
- In C++, we encourage you to use string more, to replace char[] and char \*p = new. Because it offer you more functions, such as **compare, find, erase, replace, insert**.
- About real size of string, string usually manage it's memory by itself, but you can use two functions string::capacity() and string::reserve() to do some simple work. If the size of string is not enough, string will allocates a new block twice the size and copy the old content. You can use capacity to know the real block size. And use reserve to tell string at least you need minimum size of bloc. For example, str.reserve(50), str.capacity will return 63. 63 = 64-1; string need the last char to store '\n' and end of symbol.
- When you use string with some legacy c function, use string.c\_str() function for read only function, string.c\_str() return a const char \*. so You can't modify it. If you Legacy C function want to fill in a string. you need do below:

```

1. Old_c(const char* p); string str;
2. Old_c(str.c_str()); // legacy C function read a string
3.
4. // legacy C write
5. size_t Old_c(char *pArray, size_t arraySize);
6.
7. vector<char> vc(maxNumChars);
8. // 1) create a vector whose size is maxNumChars
9. size_t charsWritten = Old_c(&vc[0], vc.size());
10. // 2) have fillString write into vc
11. string s(vc.begin(), vc.begin() + charsWritten);
12. // 3) copy data from vc to s via range constructor

```

- string method lists:

|                                                                                |                                                                                                                                            |
|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| capacity(), reserve()                                                          | Returns the largest number of elements that could be stored in a string without increasing the memory allocation of the string.            |
| empty(), size()<br>max_size()<br>resize()                                      | maximum number of characters<br>Specifies a new size for a string, appending or erasing elements as required.                              |
| length()                                                                       | Returns the current number of elements in a string.                                                                                        |
| find(), rfind()                                                                | Searches a string in a forward/backward direction for the first occurrence of a substring that matches a specified sequence of characters. |
| substr()                                                                       | Copies a substring of at most some number of characters from a string beginning from a specified position.                                 |
| find_first_not_of()<br>find_first_of()<br>find_last_not_of()<br>find_last_of() | Searches through a string for the first character that is not any element of a specified string.                                           |
| .                                                                              | .                                                                                                                                          |

- find function in the string return position. Vector doesn't have find member function, you can use find function in algorithm category.

```

1. size_t found;
2. found=str.find("haystack");
3. // or found=str.find('.');
4. if (found!=std::string::npos)
5. std::cout << "'haystack' also found at:" << found << '\n';
6.
7. vector<int> myints = { 10, 20, 30, 40 };
8. auto it = std::find (myints.begin(), myints.end(), 30);
9. if(it!=myints.end())
10. cout << "find_30" << endl;

```

|                                                |                                                                                                                                                                                                                                                                    |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| begin(),end()<br>rbegin(), rend()              | Returns an iterator addressing the first element in the string.                                                                                                                                                                                                    |
| clear()<br>insert(), erase()                   | Erases all elements of a string.<br>Inserts an element or a number of elements or a range of elements into the string at a specified position.                                                                                                                     |
| append(), push_back()<br>assign()<br>replace() | Adds characters to the end of a string.<br>Assigns new character values to the contents of a string.                                                                                                                                                               |
| at()<br>c_str(), data()<br>compare()<br>copy() | a reference to the element at a specified location<br><br>Copies at most a specified number of characters from an indexed position in a source string to a target character array. The function does not append a null character at the end of the copied content. |
| get_allocator()<br>swap()                      | Returns a copy of the allocator object used to construct the string.<br>Exchange the contents of two strings.                                                                                                                                                      |

- clear will delete all the characers, and erase will remove character in certain position.
- position in find method can be used in loop

```

1. std::string str ("Please,_replace_the_vowels_with_asterisks.");
2. std::size_t found = str.find_first_of("aeiou");
3. while (found!=std::string::npos)
4. {
5.     str [found]='*';
6.     found=str.find_first_of("aeiou",found+1);
7. }

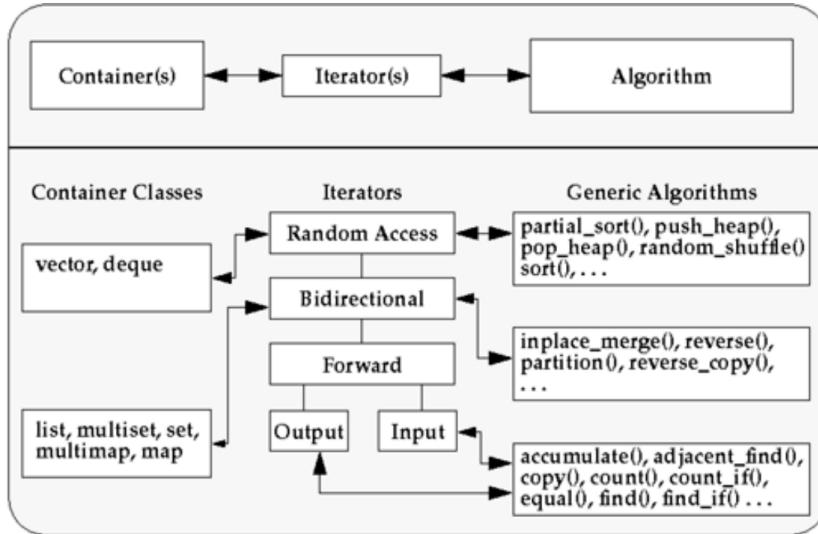
```

### 9.3 Iterator

- There are 5 classification iterators: input, output, forward, bi-direction, random. You need to know these five are not real class, they are just conception, not real implementation.
- As mentioned earlier, each container class defines a class scope typedef name called iterator. So the `vector<int>` class has iterators of type `vector<int>::iterator`. The document will tell you vector iterators is random access iterators.
- In practical point of view: There are four points you need to know:
  1. Five classification and associated supported operation.
  2. Common container's iterator classification.
  3. Common algorithm can accept what kind of iterator. (See next chapter)
  4. Use typedef simply define container iterator.(typedef is good friend when you use stl more and more);
- Now, five classification iterators play important role here: From this figure, you can know what operation each iterator supports.

| category       |         | properties                                                          |                                                                                      | valid expressions                  |
|----------------|---------|---------------------------------------------------------------------|--------------------------------------------------------------------------------------|------------------------------------|
| all categories |         | <i>copy-constructible, copy-assignable and destructible</i>         |                                                                                      | x b(a);<br>b = a;                  |
|                |         | Can be incremented                                                  |                                                                                      | ++a<br>a++                         |
| Bidirectional  | Forward | Input                                                               | Supports equality/inequality comparisons                                             | a == b<br>a != b                   |
|                |         |                                                                     | Can be dereferenced as an <i>rvalue</i>                                              | *a<br>a->m                         |
|                |         | Output                                                              | Can be dereferenced as an <i>lvalue</i><br>(only for <i>mutable iterator types</i> ) | *a = t<br>*a++ = t                 |
|                |         |                                                                     | <i>default-constructible</i>                                                         | X a;<br>X()                        |
|                |         |                                                                     | Multi-pass: neither dereferencing nor incrementing affects dereferenceability        | { b=a; *a++;<br>*b; }              |
|                |         |                                                                     | Can be decremented                                                                   | --a<br>a--<br>*a--                 |
| Random Access  |         | Supports arithmetic operators + and -                               |                                                                                      | a + n<br>n + a<br>a - n<br>a - b   |
|                |         | Supports inequality comparisons (<, >, <= and >=) between iterators |                                                                                      | a < b<br>a > b<br>a <= b<br>a >= b |
|                |         | Supports compound assignment operations += and -=                   |                                                                                      | a += n<br>a -= n                   |
|                |         | Supports offset dereference operator ([])                           |                                                                                      | a[n]                               |

- Basic container iterator is below:



- In each container, iterator of container is implemented by itself. There is no iterator class inheritance hierarchical structure. It means that in vector, there is vectIter: Iterator (vectIter inherit from base iterator class), then you define `++` operator in class vectIter class. It's totally wrong. In fact, In vector, Maybe iterator is defined by:

```

1. template<type T>
2. vector<T> {
3.     ...
4.     typedef T* iterator;
5. };
6.
7. vector<int>::iterator ip; //use it outside.
8. // ++ and -- is done by pointer automatically.

```

and All the member function of vector know the iterator very well, inside, it will use `T*` directly,

```

1. push_back(T x) {
2.     *end++ = x;
3. } // you can see there is no iterator
4. // when you implement container implementation.

```

- When you see the algorithm, you will see the algorithm is based on template, not inheritance. (You don't need to make type can be cast or not. ). **Algorithm don't care what iterator really is, It just make sure each iterator can support what operation.**

```

1. template <class InputIterator, class OutputIterator>
2.     OutputIterator copy (InputIterator first, InputIterator last,
3.                         OutputIterator result);

```

- Common four iterator errors.

- Valid values: Is the iterator dereferenceable? For example, writing `"*e.end()"` is always a programming error.
- Valid lifetimes: Is the iterator still valid when it's being used? Or has it been invalidated by some operation since we obtained it?
- Valid ranges: Is a pair of iterators a valid range? Is first really before (or equal to) last? Do both really point into the same container?

4. Illegal builtin manipulation: For example, is the code trying to modify a temporary of builtin type, as in "`-e.end()`" above? (Fortunately, the compiler can often catch this kind of mistake for you, and for iterators of class type, the library author will often choose to allow this sort of thing for syntactic convenience.)

```

1. int main() {
2.     vector<Date> e;
3.     copy( istream_iterator<Date>( cin ), 
4.           istream_iterator<Date>(), back_inserter( e ) );
5.     vector<Date>::iterator first = find( e.begin(), e.end(), "01/01/95" );
6.     vector<Date>::iterator last = find( e.begin(), e.end(), "12/31/95" );
7.     *last = "12/30/95";
8.     copy( first, last, ostream_iterator<Date>( cout, "\n" ) );
9.     e.insert( --e.end(), TodaysDate() );
10.    copy( first, last, ostream_iterator<Date>( cout, "\n" ) );
11. }
```

- You can inheritate your own iterator from `std::iterator`: yes, that's what it's for. If you mean anything else: no, because none of the STL iterators have virtual destructors. They're not meant for inheritance and a class inheriting from them might not clean up properly. A Good example can be seen link:  
<http://www.cplusplus.com/reference/iterator/iterator/>

- two common iterator operation: `advance()` and `distance()`.

```

1. std::list<int>::iterator first = myList.begin();
2. std::list<int>::iterator last = myList.end();
3.
4. std::advance(first, 3) // 
5. std::cout << std::distance(first, last)
```

- `istream_iterator` and `istreambuf_iterator` topic:

```

1. std::ostream_iterator<int> out_it ( std::cout, ", " );
2. std::copy ( myvector.begin(), myvector.end(), out_it );
3.
4. ifstream inputFile("aa.txt");
5. string fileData((istreambuf_iterator<char>(inputFile)), 
6.                  istreambuf_iterator<char>());
7. // It will just read all the character. (including white character)
8. // I don't need format data,
9.
10. ifstream inputFile("aa.dat");
11. list<int> data((istreambuf_iterator<char>(inputFile)), 
12.                  istreambuf_iterator<char>());
13. // It wil read format data, and use white space as delimiter.
```

### 9.3.1 Insert iterator

- Three common iterator generator: `inserter()`, `back_inserter()`, `front_inserter()`; They will produces three `insert_iterator`:

1. `insert_iterator`
2. `back_insert_iterator`
3. `front_insert_iterator`

- When you use `insert_iterator` in an assignment, `insert_iterator` will call `insert()` function. `back_insert_iterator` will call `push_back()` and `front_insert_iterator` will call `push_front()`. `insert_iterator++` has no any operator inside.
- `std::inserter` is commonly used with sets
- Why do we need it. in some algorithm, such as copy and generator, if you read sth from a container, you can use regular iterator, but when you want to write into a container. You must keep regular iterator is valid. so a method is to use reverse before you write to a container.

```

1. list<State> res2( src.size() );
2. copy( src.begin(), src.end(),
3.       res2.begin() ); // often works ...

```

- Another way is to use `back_insert_iterator`.

```

1. std::vector<int> foo;
2. for (int i=1; i<=5; i++)
3. {
4.     foo.push_back(i); bar.push_back(i*10);
5.     std::copy (bar.begin(), bar.end(), back_inserter(foo));
6.     //foo support push_back
7.
8.     std::deque<int> foo;
9.     // deque or list
10.    for (int i=1; i<=5; i++)
11.    {
12.        foo.push_back(i); bar.push_back(i*10);
13.        std::copy (bar.begin(), bar.end(), front_inserter(foo));
14.        //foo support push_front

```

- Of course, you can use the random access iterators (or any output iterator) in algorithms like `std::copy`, as third argument, but that assumes the iterator is referencing to existing range — `*it` and `++it` are well-defined for the value you passed. You pass them to overwrite the existing elements of the range, whereas `std::back_insert_iterator` adds new elements to the container.
- For copy algorithm, pass `vect.end()` is not meanful, because It's not a valid iterator to support `*it = new_obj.`

### 9.3.2 Reverse iterator

- `reverse_iterator` topic: You need a front-to-back traversal or a back-to-front traversal. The reason for reverse iterators is that the standard algorithms do not know how to iterate over a collection backwards. For example:

```

1. std::find(foo.begin(), foo.end(), L'a');
2.                      // Returns an iterator pointing
3.                      // to the first a character.
4. std::find(foo.rbegin(), foo.rend(), L'a').base() -1;
5.                      // Returns an iterator
6.                      // pointing to the last A.
7. std::find(foo.end(), foo.begin(), L'a');
8.                      //WRONG!! (Buffer overrun)

```

- `reverse_iterator` can use `base()` to change a normal iterator, because all the contain member function just receive normal iterator. such as `insert` and `erase`. why `ri.base()` and `ri` will have step 1 advance. detail can be seen effective stl item 28.

```

1. reverse_iterator ri =
2.     find(foo.rbegin(), foo.rend(), L'a');
3. foo.insert(ri.base()); // you can use ri.base() directly.
4.                                     // when you want to insert.
5. foo.erase((++ri).base()); //you can't use ri.base() directly.
6.                                     //when you erase what you want.
7.     ri
8. 1 2 3 4 5
9.     i = ri.base()

```

## 9.4 Algorithms

### 9.4.1 Basic

- Effective STL item 43, Prefer algorithm calls to hand-written loops.

1. Handing writing is prone to bug

```

1. for( size_t i = 0; i < max; ++i )
2. d.insert( current++, data[i] + 41 ); // do you see the bug?

```

2. You can fix it bu with careful design

```

1. for( size_t i = 0; i < max; ++i ) {
2.     current = d.insert( current, data[i] + 41 );
3.     // be careful to keep current valid
4.     ++current; // then increment it when it's safe
5. }

```

3. but a better way is

```

1. transform( data, data + max, // copy elements from data
2. inserter(d, d.begin()), // to the front of

```

4. With a complex algorithem, just use STL alogrithem+lambda

5. but a better way is

```

1. for( vector<int>::iterator i = v.begin(); i != v.end(); ++i )
2. if( *i > x && *i < y ) break;
3.
4. // This is better version now.
5. vector<int>::iterator i = find_if( v.begin(), v.end(),
6.                                     [x, y](int &i){i>x && i<y} );

```

But if you need a loop that dees something fairly simpler, but would require a confusing tangle of binders and adapters, just use loop.

```

1. list<Widget> lw;
2. type list<Widget>:: iterator WI;
3. for_each(lw.begin(), lw.end(),
4.           mem_fun_ref(&Widget::redraw));
5.
6. transform(data, data+10, inserter(deque, deque.begin()),
7.           bind2nd(plus<double>(), 41));

```

- There are four groups: non modifying sequence , mutating sequence, sorting, generalized numeric operations. Detail can be see c++ primer p1286.

- Note which algorithm expect sorted ranges:

```

1. binary_search lower_bound upper_bound equal_range
2. set_union set_intersection set_difference
3. merge inplace_merge includes
4. unique unique_copy

```

- Make sure destination ranges are big enough effective stl item 30

```

1. vector<int> values, result;
2. int doSth(int x); // function
3. // make destination range big enough
4. result.reserve(result.size() + values.size());
5.
6. transform(values.begin(), values.end(), result.end(), doSth);
7. // It's error. transform writes its result by making
8. // assignment. result.end() has no object at all
9.
10. transform(values.begin(), values.end(),
11. back_inserter(result), doSth) // insert end
12.
13. transform(values.rbegin(), values.rend(),
14. front_insert(result), doSth)
15. // result must be a list
16. // use rbegin make insert in front right order

```

- Know you sorting options: It makes no sense to sort elements in standard associative containers, because such containers use their comparison functions to remain sorted all the time.
- For other sequence container, sorting options is below: (1-3) need random iterator. 4 only need bidirection iterator.(list)

1. full sort on vector, string, deque, or array. use sort or stable\_sort
2. put only the top n elements in order, partial\_sort
3. Identify the elements at position n , you need nth\_element
4. spearate the elements of a standard sequence container, do satisfy some criterion, you use partition or stable\_partition.
5. for list, you should use list.sort() in place of common sort.

```

1. vector<Foo> vf;
2. bool compare(const Foo& f1, const Foo& f2);
3. partial_sort(vf.begin(), vf.begin() + 20, vf.end(), compare);
4. sort(vf.begin(), vf.end(), compare);
5. nth_element(vf.begin(), vf.begin() + 20, vf.end(), compare);
6.
7. bool good(const Foo &f1)
8. partition(vf.begin(), vf.end(), good);

```

- Be wary of remove-like algorithm on container of pointers

```

1. void delAndNULL(Foo*& pf){ // use pointer reference here.
2. if(!pf->isCertified()){ delete pf; pf = nullptr}
3. }

```

```

4. for_each(v.begin(), v.end(), delAndNULL);
5. v.erase(remove(v.begin(), v.end(),
                 static_cast<Foo*>(0)), v.end());
6.

```

- For a algorithms, you need to know three points: 1) what's function for 2) what iterator it accept, 3) what functor it will accept. So please see below summary.

## 9.4.2 STL algorithms

### 9.4.2.1 basic notation

Iterator:

|        |                                                    |
|--------|----------------------------------------------------|
| b, f   | a bidirectional, forward iterator                  |
| i,o,a  | an input, output, random iterator                  |
| (?, ?) | a pair of iterators as a return value, as in (f,f) |

functor:

|              |                                                                                                                            |
|--------------|----------------------------------------------------------------------------------------------------------------------------|
| upred, bpred | a unary or binary predicate (boolean function or function object) (generally used to test a single value from a container) |
| ufunc, bfunc | a unary or binary (value-returning) function or functor                                                                    |
| pfunc        | a "parameterless" (value-returning) function (or function object) (often used to "generate" a value of some kind)          |
| uproc, bproc | a unary or binary procedure (void function or function object)                                                             |
| pproc        | a "parameterless" procedure (void function or functor)                                                                     |

Parameter:

|         |                                                     |
|---------|-----------------------------------------------------|
| n, v, & | a quantity (or size). A value. reference to a value |
|---------|-----------------------------------------------------|

### 9.4.2.2 Applying

|                           |                                                                    |                             |
|---------------------------|--------------------------------------------------------------------|-----------------------------|
| ufunc for_each(i,i,ufunc) | Apply a function to every item in a range and return the function. | ufunc may not return value. |
|---------------------------|--------------------------------------------------------------------|-----------------------------|

```

1. void fun(int n){
2.     cout<<n;
3. } // all function definition end no semicolon
4.
5. struct Sum{
6.     Sum() : sumEven{0} {} //no semicolon here
7.     void operator()(int n) { if(n%2 ==0) ; sumEven += n; }
8.     int sumEven;
9. }; // type definition need semicolon
10.
11. vector<int> nums{3, 4, 2, 8, 15, 267};
12. for_each(nums.begin(), nums.end(), fun); // no () here
13. Sum s = for_each(nums.begin(), nums.end(), Sum()); //have() here
14. for_each(nums.begin(), nums.end(), [sumEven](int n){
15.             if(n%2==0) sumEven+=n;
16.         });

```

- for\_each need a functor, So when to use for\_each equal another question, when use functor.

1. For fun example, a better way is for(auto e: nums){cout<<e<< " ;}. Here just demonstrate that you can input fun.
2. For struct Sum just use once, lambda function is better, because it is cleaner.
3. If functor need to be 1)customized state and 2)reuse many time, then functor is better.

```

1. struct GreatThanX {
2.     GreatThanX(int x): cutoff{x} {} //no semicolon here
3.     bool operator()(int n) { if(n>x) ; return true; return false; }
4.     int cutoff;
5. };
6.
7. vector<int> nums{3, 4, 2, 8, 15, 267};
8. find_if(nums.begin(), nums.end(), GreatThanX(3));
9. copy_if(nums.begin(), nums.end(), GreatThanX(7));
10. .....

```

4. Only functor can be use a argument input to map or set container set. See below examples.

```

1. struct lex_compare {
2.     bool operator() (const int64_t& lhs, const int64_t& rhs) const{
3.         stringstream s1,s2;           s1 << lhs;   s2 << rhs;
4.         return s1.str() < s2.str();
5.     }
6. };
7. set<int64_t, lex_compare> s;

```

5. Lambdas aren't useful for more complex scenarios because they weren't made for them. They provide a short and concise way of creating simple function objects for correspondingly simple situations.
6. with function, bind and name lambda, You can reach above requirement. but it's not as good as functor

```

1. auto f = [](int x, int y){if(y>x) return true; return false;} ;
2. auto f1 = bind(f, 3, placeholders::_1) // GreatThanX(3);
3. if(f1(8)) cout<<"8>3"<<endl;

```

### Transforming

o transform(i1,i1end,o,ufunc)  
o transform(i1,i1end,i2, o, bfunc)

Transform one range of values into  
another.

Ret of ufunc or bfunc wr-  
ited to o.

- If output is associate container, you need use inserter( ) function to get inserter iterator.

```

1. std::transform(a.begin(), a.end(),
                 std::inserter(set, set.begin()), modify);
2.
3. std::transform(a.begin(), a.end(), b.begin(),
                 a.begin(), plus<int>());
4. // 1) you can make in place modification.
5. // 2) plus minus, multiplies, divides, modulus, negate, equal_to
6. // are often used in transform algorithms.

```



### 9.4.2.3 Bounding

|                                                            |                                                                                                 |                                                                                                           |
|------------------------------------------------------------|-------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| (f,f) equal_range(f,f,&)<br>(f,f) equal_range(f,f,&,bpred) | Find the lower bound and upper bound of a value within a range and return a pair of iterators . | 1) container should be sorted firstly, 2) order by default < or bpred. 3) Sorted by bpred, find by bpred. |
| f lower_bound(f,f,&)<br>f lower_bound(f,f,&,bpred)         | Find the lower bound of a value within a range and return an iterator pointing to it.           |                                                                                                           |
| f upper_bound(f,f,&)<br>f upper_bound(f,f,&,bpred)         | Find the upper bound of a value within a range and return an iterator pointing to it.           |                                                                                                           |

### 9.4.2.4 Comparing

|                                                                                                                                                                                                                                                       |                                                                                                                                                    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| bool equal(i1,i1last, i2)<br>bool equal(i1,i1last,i2,bpred)<br>bool lexicographical_compare<br>(i1,i1last,i2,i2last) bool lexicographical_compare (i1,i1last,i2,i2last,bpred)<br>(i1,i2) mismatch(i1,i1last,i2) (i1,i2) mismatch(i1,i1last,i2, bpred) | Check if the values in two ranges match.                                                                                                           |
|                                                                                                                                                                                                                                                       | Compare two ranges lexicographically, and return true if the first range is less than the second; otherwise return false.                          |
|                                                                                                                                                                                                                                                       | Search two ranges for the first two items in corresponding positions that don't match, and return a pair of iterators pointing to those two items. |

- An example to use equal:

```

1. bool is_palindrome(const std::string& s){
2.     return std::equal(s.begin(), s.begin() + s.size() / 2,
3.                       s.rbegin());
4. }
```

### 9.4.2.5 copy

|                                 |                                                                                                                  |
|---------------------------------|------------------------------------------------------------------------------------------------------------------|
| o copy(i,i,o)                   | Copy a range of items to a destination and return an iterator pointing to the end of the copied range.           |
| b2 copy_backward(b1,b1last, b2) | Copy a range of items backwards to a destination and return an iterator pointing to the end of the copied range. |

Example:

```

1. for (int i=1; i<=5; i++)
2.     myvector.push_back(i*10);
3.     // myvector: 10 20 30 40 50
4.
5. copy(myvector.begin(), myvector.end(), myvector.begin() +3)
6. //error, when source and target is overlap,
7. //you have to use backward copy
8.
9. copy_backward(myvector.begin(), myvector.end(), myvector.begin() +4)
10. // pay attention, copy_backward, *(--last) =
11. //if you want copy to position 3, you need to input position 4
12. //or you can input vector.end() as target,
13. //but for copy, you can't input vector.end()
```

- Copy. Inside Copy, It use assignment operator =. o should be insert iterator or target should be use reserve() to allocate space for assignment operator =.

#### 9.4.2.6 Count

|                       |                                                                            |                                          |
|-----------------------|----------------------------------------------------------------------------|------------------------------------------|
| n count(i,i,& )       | Count the items in a range that match a value and return that count.       | count will not stop when it finds match. |
| n count_if(i,i,upred) | Count the items in a range that satisfy a predicate and return that count. |                                          |

#### 9.4.2.7 Filling and Generating

##### Filling

|                |                                                  |
|----------------|--------------------------------------------------|
| fill(f,f,& )   | Set every item in a range to a particular value. |
| fill_n(o,n,& ) | Set n items to a particular value.               |

##### Generating

|                       |                                        |
|-----------------------|----------------------------------------|
| generate(f,f,pfunc)   | Fill a range with generated values.    |
| generate_n(o,n,pfunc) | Generate a specified number of values. |

- examples:

```

1. std :: vector<int> v(5);
2. std :: generate(v. begin(), v. end(), std :: rand);
   // Using the C function rand()
3.
4.
5. int n = {0};
6. std :: generate(v. begin(), v. end(), [&n]{ return n++; });
7. // 1, 2, 3, 4, 5

```

##### Filtering , should used on sorted container

|                            |                                                                                                                                          |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| f unique(f,f)              | Collapse each group of consecutive duplicate values to a single value, and return an iterator pointing to the end of the modified range. |
| f unique(f,f,bpred)        |                                                                                                                                          |
| o unique_copy(i,i,o)       | Copy a range of values, performing the same action as unique above, and return an iterator pointing to the end of the new range.         |
| o unique_copy(i,i,o,bpred) |                                                                                                                                          |

```

1. int myints[] = {10,20,20,20,30,30,20,20,10};
2. std :: vector<int> myvector (myints, myints+9);
3.
4. // using default comparison:
5. std :: vector<int>::iterator it;
6. it = std :: unique (myvector. begin(), myvector. end());
7. // 10 20 30 20 10 ? ? ? ?

```

##### Heap

|                       |                                               |                                                                                       |
|-----------------------|-----------------------------------------------|---------------------------------------------------------------------------------------|
| make_heap(r,r)        | Make a range of values into a heap.           | It needs random iterator. priority_queue use them inside. You don't use them directly |
| make_heap(r,r,bpred)  |                                               |                                                                                       |
| o pop_heap(r,r)       | Delete the first value from a heap.           |                                                                                       |
| o pop_heap(r,r,bpred) |                                               |                                                                                       |
| push_heap(r,r)        | Insert the last value of a range into a heap. |                                                                                       |
| push_heap(r,r,bpred)  |                                               |                                                                                       |
| sort_heap(r,r)        | Sort a heap.                                  |                                                                                       |
| sort_heap(r,r,bpred)  |                                               |                                                                                       |

#### 9.4.2.8 Math

```
v accumulate(i,i,v)
v accumulate(i,i,v,bfunc)
o adjacent_difference (i,i,o)
o adjacent_difference
(i,i, o, bfunc)
v inner_product
(i1,illast,i2,vInitial) v inner_product
(i1,illast,i2,v,bfunc1,bfunc2)
o partial_sum(i,i,o)
o partial_sum(i,i,o, bfunc)
```

Add an initial value and the values in a range, return sum.

Calculate the difference between adjacent pairs of values, write the differences to an o, and return the end of that output range.

Calculate the inner product of two ranges and return that value plus vInitial.

Fill a range with running totals and return an iterator pointing to.

- An example about partial\_sum

```
1. std :: vector<int>v(10, 2); // new initialize method
2. std :: partial_sum(v. begin(), v. end(),
3.                   std :: ostream_iterator<int>(std :: cout, " "));
4. //2 4 6 8 10 12
5.
6. std :: partial_sum(v. begin(), v. end(), v. begin(),
7.                   std :: multiplies<int>());
8. std :: cout << "The first 10 powers of 2 are: ";
9. for (auto n : v) {
10.   //2 4 8 16 32
11. }
```

- An example about accumulate

```
1. std :: vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2. int sum = std :: accumulate(v. begin(), v. end(), 0);
3. int product = std :: accumulate(v. begin(), v. end(), 1,
4.                                 multiplies<int>());
5.
6. std :: string s = std :: accumulate(v. begin(), v. end(),
7.                                   std :: string{},
8.                                   [] (const std :: string& a, int b) {
9.         return a. empty() ? to_string(b)
10.                      : a + '-' + to_string(b); });
11. // to_string is in C++11
```

- An example about adjacency\_difference

```
1. v = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
2. std :: adjacent_difference(v. begin(), v. end() - 1,
3.                           v. begin() + 1, std :: plus<int>());
4. //fibonacci
```

- difference some algorithms

1. For accumulate, input a init,  $\rightarrow$  calculate(init, element)  $\rightarrow$  return it to init.
2. For adjacent\_difference: first  $\rightarrow$  dfirst, f(first+1, first)  $\rightarrow$  dfirst+1; Pay attention, for first element, assign it directly.
3. For partial\_sum function, you can think it as accumulate, **accumulate just return one value, but partial\_sum return many value, and write them back to target iterators one by one.**

4. accumulate and inner\_product return one value, partial\_sum and adjacent\_difference return a list of value.

#### 9.4.2.9 Merging

|                              |                                                                                  |
|------------------------------|----------------------------------------------------------------------------------|
| inplace_merge(b,b,b)         | Merge two sorted ranges, in place, into a single sorted range.                   |
| inplace_merge(b,b,b,bpred)   | Merge two sorted ranges into a single sorted range.                              |
| o merge(i1,i1,i2,i2,o)       |                                                                                  |
| o merge(i1,i1,i2,i2,o,bpred) |                                                                                  |
| <b>Min/Max</b>               |                                                                                  |
| & min(&,&)                   | Find the minimum of two values and return a reference to that value.             |
| & min(&,&,bpred)             |                                                                                  |
| & max(&,&)                   | Find the maximum of two values and return a reference to that value.             |
| & max(&,&,bpred)             |                                                                                  |
| f min_element(f,f)           | Find the minimum value in a range and return an iterator pointing to that value. |
| f min_element(f,f,bpred)     |                                                                                  |
| f max_element(f,f)           | Find the maximum value in a range and return an iterator pointing to that value. |
| f max_element(f,f,bpred)     |                                                                                  |

#### 9.4.2.10 Partitioning

|                               |                                                                                                                                                                                                        |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| nth_element(r,r,r)            | Partition a range of values so that the value pointed to by the middle r in the parameter list is in its correct sorted position, and no element to its left is greater than any element to its right. |
| nth_element(r,r,r,bpred)      |                                                                                                                                                                                                        |
| b partition(b,b,upred)        | Partition a range of values using a predicate, and return an iterator pointing to the first value for which upred returns false.                                                                       |
| b stable_partition(b,b,upred) | Partition a range using a predicate without altering the relative order of the values, and return an iterator pointing to the first value for which upred returns false.                               |

- An example of partition:

```

1. std::vector<int> v = {0,1,2,3,4,5,6,7,8,9};
2. auto it = std::partition(v.begin(), v.end(),
3.                           [] (int i) {return i % 2 == 0;});
4. std::copy(std::begin(v), it,
5.           std::ostream_iterator<int>(std::cout, " "));
6. std::copy(it, std::end(v),
7.           std::ostream_iterator<int>(std::cout, " "));
8. \\output: 0 8 2 6 4      5 3 7 1 9

```

- **Partition return: Iterator to the first element of the second group.**

- All of the elements before this new nth element are less than or equal to the elements after the new nth element. Who are my top 20 salespeople?" For example, nth\_element( s.begin(), s.begin() + 19, s.end(), SalesRating ); puts the 20 best elements at the front.
- If you use nth\_element on most of the range, It may be slower than a full sort.

#### 9.4.2.11 Permuting

```
bool next_permutation(b,b)
bool next_permutation
(b,b,bpred)
bool prev_permutation (b,b)
bool prev_permutation
(b,b, bpred)
```

A example:

```
1. int a[] = {1,2,3};
2. do {
3.     cout << a[0] << ' ' << a[1] << ' ' << a[2] << '\n';
4. } while ( std::next_permutation(a,a+3) );
5. 1 2 3 // 1 3 2 // 2 1 3
6. 2 3 1 // 3 1 2 // 3 2 1
```

Change a range of values to the next lexicographic permutation of those values, and return true, or false if no next permutation exists.

Change a range of values to the previous lexicographic permutation of those values, and return true, or return false if no previous permutation exists.

#### 9.4.2.12 Random/shuffling

```
random_shuffle(r,r)
random_shuffle(r,r,ranGen)
```

Randomize a range of values, and use the random generator function ranGen, if supplied, rather than an internal random generator.

#### 9.4.2.13 Removing

```
remove(f,f,&)
remove_if(f,f,upred)
remove_copy(i,i,o,&)
remove_copy_if(i,i,o,upred)
```

Remove from a range of values all values that match a given value or satisfy a predicate

Copy a range of values, removing all values that match a given value.

- These algorithms cannot be used with associative containers such as std::set and std::map because ForwardIt does not dereference to a MoveAssignable type (the keys in these containers are not modifiable)
- list has its own remove
- Then, remove only can be used in erase remove idiom in vector and string.

#### 9.4.2.14 Replacing

```
replace(f,f,&,&)
replace_if(f,f,upred,&)
replace_copy(i,i,o,&,&)
replace_copy_if(i,i,o,upred,&)
```

Replace, within a range of values, one specified value (satisfies a predicate) with another value.

Copy replacing one specified value with another specified value.

#### 9.4.2.15 Reverse

```
reverse(b,b)
reverse_copy(b,b,o)
```

Reverse the order of all values in a range of values.

Reverse and copy

#### 9.4.2.16 Rotating

```
rotate(f,f,f)
rotate_copy(f,f,f,o)
```

Rotate a range of values by n positions.

Copy and rotating it by n position.

- Rotates the order of the elements in the range [first,last), in such a way that the element pointed by middle becomes the new first element.

```

1. for (int i=1; i<10; ++i) myvector.push_back(i);
2. // 1 2 3 4 5 6 7 8 9
3. rotate(myvector.begin(), myvector.begin() +3,
4.                                myvector.end());
5. // 4 5 6 7 8 9 1 2 3

```

#### 9.4.2.17 Searching

##### 1:Sorted range Searching

bool binary\_search(f,f,&)

| sorted range of values and return bool.

##### 2: linear time Searching

f adjacent\_find(f,f)

| the first pair of equal adjacent values in a range and return an iterator pointing to the first value of the pair.

f adjacent\_find(f,f, bpred)

| return an iterator pointing to the value or end

i find(i,i,&)

| satisfies a predicate and return an iterator pointing to the first such value, or to the end

i find\_if(i,i,upred)

##### 3: Searches for a single element from a range

f1 find\_first\_of(f1,f1,f2,f2)

| std::find\_first\_of searches for a single element from a range within another range.

f1 find\_first\_of(f1,f1,f2,f2, bpred)

##### 4: Below 3 algorithms search for a whole range of elements within another range

f1 search(f1,f1,f2,f2)

| first occurrence of a second range of values within a first range . return an iterator pointing to the first value of that first match. or end of the first range

f1 search(f1,f1,f2,f2)

| the last occurrence of a second range of values in a first range of values and return an iterator pointing to the first value of that last match within the first range, or pointing to the end of the first range(not find)

f1 find\_end(f1,f1,f2,f2)

| For a contiguous sequence of n values each equal to &, return iterator to the first of those values, or the end of the range

f1 find\_end(f1,f1,f2,f2, bpred)

#### 9.4.2.18 set

bool includes(i1,i1,i2,i2)  
 bool includes(i1,i1,i2,i2, bpred)  
 o set\_difference(i1,i1,i2,i2,o)  
 o set\_difference  
 (i1,i1,i2,i2, bpred)  
 o set\_intersection(i1,i1,i2,i2,o)  
 o set\_intersection  
 (i1,i1,i2,i2, bpred)  
 o set\_union(i1,i1,i2,i2,o)  
 o set\_union(i1,i1,i2,i2, bpred)  
 o set\_symmetric\_difference  
 (i1,i1,i2,i2,o)  
 o set\_symmetric\_difference  
 (i1,i1,i2,i2, bpred)

| Search for all values from the second range in the first range and return true if found, or false  
 in the first range but not in the second range and return the end of that output range.

| in the first range and also in the second range and return the end of that output range.

| either in the first range or in the second range and return the end of that output range.

| not common to both ranges and return the end of that output range.

- All algorithm need two ranges should be sorted.

### 9.4.2.19 swapping

|                             |                                                                                           |
|-----------------------------|-------------------------------------------------------------------------------------------|
| iter_swap(f,f)              | Swap the values pointed to by the two iterators.                                          |
| swap(&,&)                   | Swap the two values.                                                                      |
| f2 swap_ranges(f1,f1end,f2) | Swap two ranges of values and return an iterator pointing to the end of the second range. |

### 9.4.2.20 sort

|                                    |                                                                                                               |
|------------------------------------|---------------------------------------------------------------------------------------------------------------|
| partial_sort(r,r,r)                | Sort all values till first part of range is in sorted order.                                                  |
| partial_sort(r,r,r,bpred)          | Partially sort a range of values (as above) and copy <b>as many values as will fit into an output range</b> . |
| r partial_sort_copy(i,i,r,r)       |                                                                                                               |
| r partial_sort_copy(i,i,r,r,bpred) |                                                                                                               |
| sort(r,r)                          | Sort a range of values.                                                                                       |
| sort(r,r,bpred)                    |                                                                                                               |
| stable_sort(r,r)                   | Sort and maintaining the same relative order of duplicate values.                                             |
| stable_sort(r,r,bpred)             |                                                                                                               |

## 9.5 Function object

### 9.5.1 Basic

1. In C language, we use function pointer.

```

1. int(*pf)(int , int); //declare a pointer function
2. int fun1(int i , int j);
3. pf = fun1 // or pf=&fun1; we usually skip &.
4. (*pf)(1,2) //or pf(1,2);

```

2. In C++, Some generic algorithms can also accept function by template.

```

1. void fun(int i) {
2.     //do stuff
3. }
4. //here just input fun name, it's function pointer usage
5. for_each(a.begin() , a.end() , fun);

```

3. But passing function has shortcomings:

- (a) Can't be inline.
- (b) Sometimes It can't be compile due to different compiler implementation.
- (c) You can't adapt or custom it.

So STL invented a functor(function object). It is class or structure objects for which the () operator is overloaded.

```

1. class functor { // also call function objects
2. public:
3.     void operator()(int i);
4. };
5. //here, use functor, (class object);
6. //overload operator();
7.
8. for_each(a.begin() , a.end() , functor());

```

```

9. // function accept variable, not type
10. // You must use functor() to produce a temporary functor obj.

```

4. You can use struct or class. If you want to have a private customized value, you have to use class to build a functor. Such as cutoff value in below code.

```

1. // you can use class instead struct, but you need to make
2. // operator() public, in struct, default is public, so struct is better!
3. struct less_than_7 : std::unary_function<int, bool>{
4.     bool operator()(int i) const { return i < 7; }
5. };
6.
7. class less_than_value : std::unary_function<int, bool>{
8.     less_than_value(int x) : value(x) {}
9.     bool operator()(int i) const { return i < value; }
10.    private:
11.        int value;
12. };
13.
14. std::count_if(v.begin(), v.end(), std::not1(less_than_7()));
15. std::count_if(v.begin(), v.end(), std::not1(less_than_value(7)));

```

5. In previous example, you can see advantage of usage less\_than\_value.

- (a) You can inherit from template unary\_function when you declare a functor, then your functor is adaptable by std::not1.
- (b) you can change value when you build a less\_than\_value functor.
- (c) set or map are template class. So it only accept type, not function, If you want to give set or map a customized compare function, you have to use functor to define a type.

```

1. class yanCompare{
2.     bool operator()(string &s1, string &s2) {....;}
3. };
4.
5. set<string, yanCompare> setDic;
6. //just yanCompare, no() follow it. pass into a type, not obj
7.
8.
9. remove_if(...yanComare());
10. //yanComare(), to pass a function obj.

```

6. Based on previous example, I would like to say something about **type**, **variable**, **expression**, **value**.

- (a) type is build-in type, custom type(class, struct), and pointer, reference type.
- (b) variable has a name, value and type.
- (c) expression has no name but has value and type.
- (d) value can be divided by three categories.

7. template function is a function, we have to input value, so we pass variable or expression. in previous example, yanCompare() produce a obj variable.
8. template container need type. so we have to input type, so we input yanCompare, It's a class type.

9. Given a variable or expression, we need to know its type, we can use auto, T in template and decltype. detail can be seen "type inference" section.
10. Given a container, we can get value\_type by predefined type information in container.
11. given a type, we need to define an variable, we can use typedef or using alias to replace complex type in C++(such as: vect<pair<string, int>>). In template,
12. In template class, If you define depended type, use using alias, detail can be seen in using alias part in the last chapter.
13. Sometimes, you don't want to reuse this functor which will cause you write clutter code, so C++14 introduce lambda. Detail can be seen in C++ 11 New features.

```

1. []->bool(int){return x<7;};
2. // if only return , you can omit ->bool (return type);
3. std::count_if(v.begin(), v.end(), [](int x){return x<7;});

```

14. Why you use unary\_funciton and binary\_function template, then inherit from it. It can make you functor adaptable, see below section:

## 9.5.2 Adaptable

### 9.5.2.1 Before c++11

- Four adapter is not1, not2, bind1st and bind2nd. It can change your current functor. An example is below: equal\_to is binary function, but count\_if only need unary function. bind1st or bind2nd can change binary function to unary function. another function is not1, used for unary function, not2 used for binary function.

```

1. int array[] = {10,20,30,40,50,10}; int cx;
2. cx = count_if (array, array+6, bind1st(equal_to<int>(),10));

```

- In order to use these four adapters. Your functor should be inherited from unary\_funciton or binary\_function. It makes your function object has typedef information, which not1 or not2 will use them.
- Usage of ptr\_fun, If you want to adapt a current function, (not function object), because function doesn't have any typedef information, so you have to use ptr\_fun firstly.

```

1. bool IsBad(Foo& f);
2.
3. find_if(vect.begin(), vect.end(),
4.     not1(IsBad)); //error! don't compile;
5.
6. find_if(vect.begin(), vect.end(),
7.     not1(ptr_fun(IsBad))); //OK, now:
8.
9. //recommend to use c++11 new feature
10. // std::not1(std::cref(isvowel)));
11. // std::not1(std::function<bool(char)>(isvowel)));

```

- Usage of mem\_fun, mem\_fun\_ref

```

1. list<Foo*> lpf
2. for_each(lpf.begin(), lpf.end(), mem_fun_ref(&Foo::testFun));
3.
4. list<Foo> lf
5. for_each(lf.begin(), lf.end(), mem_fun(&Foo::testFun));

```

- With variadic templates, a lot of general function composing can be expressed much more simply and consistently, so all of the old cruft is no longer necessary: deprecated in C++ 11. (but support) **Do use:** `std::function`, `std::bind`, `std::mem_fn`, `std::result_of`, `lambdas`. **Don't use:** `std::unary_function`, `std::binary_function` `std::mem_fun`, `std::bind1st`, `std::bind2nd`

### 9.5.2.2 After c++11

- `std::mem_fn` is deprecated. `std::mem_fn` can do everything it does, and it does it more conveniently. The relation between the two is the same as the relation between `std::bind1st`/`std::bind2nd` and the C++11 `std::bind`.
- `std::bind` use variadic template parameter.

```

1. struct Foo {
2.     void display_greeting() {
3.         std::cout << "Hello, world.\n";
4.     }
5.     void display_number(int i) {
6.         std::cout << "number: " << i << '\n';
7.     }
8.     int data = 7;
9. };
10.
11. Foo f; // use & here to get address
12. auto greet = std::mem_fn(&Foo::display_greeting);
13. greet(f); // have to input a object f
14.
15. auto print_num = std::mem_fn(&Foo::display_number);
16. print_num(f, 42);
17.
18. auto access_data = std::mem_fn(&Foo::data);
19. std::cout << "data: " << access_data(f) << '\n';

```

### 9.5.2.3 member function

If you called for `_each()` with `&Item::Foo`, the code try to call `(&Item::Foo)(x)`, which is ill-formed since for pointers to members you have to write `(x.*&Item::Foo)()`. It's that syntactical difference that `mem_fn` is meant to solve: `mem_fn` deals with the invocation syntax of pointers to members so that you can use all the algorithms with pointers to members as well as functions and function objects. You cannot have `for_each(v.begin(), v.end(), &Item::Foo)` but you can have `for_each(v.begin(), v.end(), mem_fn(&Item::Foo))`.

```

1. template<class InputIt, class UnaryFunction>
2. UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f){
3.     for (; first != last; ++first) {
4.         f(*first); // == N.B. f(*first)
5.     }
6. }

```

- In order to resolve this problem. Three ways to use member function:

1. use std::function:

```

1. struct Foo {
2.     Foo(int num) : num_(num) {}
3.     void print_add(int i) const { std::cout << num_ + i << '\n'; }
4.     int num_;
5. };
6.
7. std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
8. const Foo foo(3);
9. f_add_display(foo, 1);

```

2. use mem\_fn:

```

1. struct Foo {
2.     void display_number(int i) {
3.         std::cout << "number:" << i << '\n';
4.     }
5.     int data = 7;
6. };
7.
8. Foo f;
9. auto print_num = std::mem_fn(&Foo::display_number);
10. print_num(f, 42);
11. auto access_data = std::mem_fn(&Foo::data);
12. std::cout << "data:" << access_data(f) << '\n';

```

3. use lambda:

```

1. typedef std::function<void(int)> fp;
2.
3. void test(fp my_func) {
4.     my_func(5);
5. }
6.
7. Foo foo;
8. test([&foo](int i){ foo.print_add(i); });

```

- Prefer mem\_fn than bind because it's verbose, Perfer mem\_fn than std::function, because it lightweight.

- You can't really compare std::function with std::mem\_fn. The former is a class template whose type you specify, and the latter is a function template with unspecified return type. There really isn't a situation in which you'd actually consider one versus the other.

A better comparison might be between mem\_fn and std::bind. There, for the specific use-case of a pointer-to-member, mem\_fn is going to a lot less verbose if all you want to do is pass-through all the arguments. Given this simple type:

```

1. struct A {
2.     int x;
3.     int add(int y) { return x+y; }
4. };
5.
6. A a{2};
7. auto add1 = std::mem_fn(&A::add);
8. auto add2 = std::bind(&A::add, _1, _2);
9. add1(a, 5); // yields 7
10. add2(a, 5); // same

```

- `mem_fn(C++11)` and `mem_fun`. `mem_fn` is new, don't use `mem_fun` any longer
  1. `mem_fn` is faster than `std::bind`. so prefer `mem_fn` first.
  2. `std::mem_fn` can only deal with member functions that take one or no argument. `std::mem_fn` is variadic and can deal with members that take any number of arguments.
  3. You also need to pick between `std::mem_fn` and `std::mem_fn_ref` depending on whether you want to deal with pointers or references for the class object (respectively). `std::mem_fn` alone can deal with either, and even provides support for smart pointers.
- **bind is deprecated in C++11, and will be given up in C++14. prefer to use lambda than bind, detail can be found in effective modern c++ item**

### 9.5.3 functor tips

- prefer to use lambda than `std::bind`, detail can be found in "Effective modern C++" item 34.
- Functor can be template.

```

1. template<typename Type>
2. class TooBig{
3.     Type cutoff;
4.     bool operator()(const T&v){ return v>cutoff; }
5. }
```

- For template functor, You can get value type from `iterator_traits`

```

1. template<typename T>
2. struct Average{
3.     T operator()(T t1, T t2){t1+t2/2;}
4. };
5.
6. transform(beg1, end1, beg1,
7. Average<typename iterator_traits<beg1>::value_type>());
8. // 1) I also can extract value_type information from container.
9. // 2) This information can be used in template function object.
```

- Make predicates pure function, see effective STL item 39. That means that there is no side effect in side the fucntion. 1) no I/O, 2) no change any state.

```

1. class Predicate: public unary_function<Foo, bool>{
2. public:
3.     bool operator()(const Foo& f) const{ // two const
4.         cutoff++; // this statement will not compile.
5.     }
6. private:
7.     int cutoff;
```

- More effective STL item 42. Make sure `less<T>` means `operator<`. Don't specialization of `std::less`. If you need to another compare, just define your own compare function.

#### 9.5.3.1 when to use `std::function`

- About `std::function` and template? In general, if you are facing a design situation that gives you a choice, use templates. I stressed the word design because I think what you need to focus on is the distinction between the use cases of `std::function` and templates, which are pretty different.

- In general, the choice of templates is just an instance of a wider principle: try to specify as many constraints as possible at compile-time. The rationale is simple: if you can catch an error, or a type mismatch, even before your program is generated, you won't ship a buggy program to your customer.
- Moreover, as you correctly pointed out, calls to template functions are resolved statically (i.e. at compile time), so the compiler has all the necessary information to optimize and possibly inline the code (which would not be possible if the call were performed through a vtable).
- When use `std::function`? One such use case arises when you need to resolve a call at run-time by invoking a callable object that adheres to a specific signature, but whose concrete type is unknown at compile-time. This is typically the case when you have a collection of callbacks of potentially different types, but which you need to invoke uniformly; the type and number of the registered callbacks is determined at run-time based on the state of your program and the application logic. Some of those callbacks could be functors, some could be plain functions, some could be the result of binding other functions to certain arguments.

|                                      | function ptr      | <code>std::function</code> | template param |  |
|--------------------------------------|-------------------|----------------------------|----------------|--|
| can capture context variables        | no(1)             | yes                        | yes            |  |
| no call overhead (see comments)      | yes               | no                         | yes            |  |
| can be inlined (see comments)        | no                | no                         | yes            |  |
| can be stored in class member        | yes               | yes                        | no(2)          |  |
| can be implemented outside of header | yes               | yes                        | no             |  |
| supported without C++11 standard     | yes               | no(3)                      | yes            |  |
| nicely readable (my opinion)         | no<br>(ugly type) | yes                        | (yes)          |  |

# Chapter 10

## Exception and error

### 10.1 End application

- In C language, You can call `abort()`, `exit()`, `quick_exit` and `_exit` to end you problem any-time. they are both declared in `<cstdlib>` head file.
- `exit( status )` terminates the process normally. a status value of 0 or `EXIT_SUCCESS` indicates success, and any other value or the constant `EXIT_FAILURE` is used to indicate an error. `exit()` performs following operations:
  1. Flushes unwritten buffered data.
  2. Closes all open files.
  3. Removes temporary files.
- `atexit()` Registers the function pointed to by `func` to be called on normal program termination (via `std::exit()` or returning from the main function)

```
1. void myProgramIsTerminating1( void ) {
2.     cout<<"exit function _1"<<endl;
3. }
4.
5. int main( int argc , char**argv ){
6.     atexit ( myProgramIsTerminating1 );
7.     //abort(); if you uncomment it , myProgramIsTerm will not be called .
8.     return 0;
9. }
```

- C++11 introduce `quick_exit`. It was added to specifically deal with the difficulty of ending a program cleanly when you use threads. `std::quick_exit()` is similar to `_exit()` but with still the option to execute some code, whatever was registered with `at_quick_exit`.
- `_exit()` is called without performing any of the regular cleanup tasks for terminating processes
- `abort()` is called without destroying any object and without calling any of the functions passed to `atexit` or `at_quick_exit`. But It will dump core, if the user has core dumps enabled. Using `abort` to debug by analysing a core dump.
- **When you use gdb, `abort` can list stack frame information for you.** It's very helpful for you debug information. Exit just end the application. When you use `gdb`, it show nothing.
- assert just call `abort`. You can use `assert` in this way.

```
1. assert (! "You should not reach here");
```

- `std::terminate` is what is automatically called in a C++ program when there is an unhandled C++ exception. **This is essentially the C++ equivalent to abort.** This calls a handler that is set by the `std::set_terminate` function, which by default simply calls `abort`.
- Don't use `exit` in `main`, It will not destroy local object in `main` function. Catch the exceptions you can't handle in `main()` and simply return from there. This means that you are guaranteed that stack unwinding happens correctly and all destructors are called.

```
1. int main() {
2.     try {
3.         // your stuff
4.     }
5.     catch( ... ) { // catch all exceptions.
6.         return EXIT_FAILURE;
7.     }
8. }
```

- according to previous `main`, if you want to end application in the other fun, you need to throw a exception, then leave it un-handle or re-throw it until it reach `main`, in this way, stack unwinding will make sure all the destructor will be called. **Don't use exit, it's C-style function and will not perform any stack unwinding**

```
1. try{
2.     fun(){
3.         throw end_exception();
4.     }
5. }
6. catch(end_exception& ex){
7.     //do something here
8.     throw;
9. }
10. // or just skip the whole catch, end_exception will reach
11. //main function all clean all the local object,
12. //when it return from main, it will do the same work as exit();
```

## 10.2 Bug and assert

### 10.2.1 Use assert

- **Bugs should be found as early as possible.** There are two basic methods to find bugs early: assert and unit test.
- Idea of assert is to make "an unnormal" can be spotted immediately, or this "an unnormal" will cause error in other place, then It's a little difficult to trace back source.

```
1. fun(char* p){
2.     assert(p!=nullptr);
3.     ..... // a lot of codes here
4.     strcpy(p) //error happen here,
5.     // but You don't know the source is the beginning of fun.
6.     fun1(p); It will cause error in other place.
7. }
```

- Although when to use assert depends on context, Use assert to its fullest. **precondition assertion** to test the validity of the arguments passed to a method. and use **postcondition assertion** to test the validity of the results produced by the method. In my whirl2llvm project, I have used them a lot. It really gave me a lot of benefits.

```

1. #include <cassert>
2. assert(I<5 && "I is more than 5");
3. //string literal is always true;
4. //when I<5 is false, the whole condition will be printed out.

```

- Assert is just if() + abort(): Difference of assert and return error(throw exception) lies in two sides:
  1. It will abort your application, and you can use GDB to trace back source easily.
  2. **Do you think that is a bug or exceptions(error) ?** Detail can be seen conclusion section.
- A practical example is dead battery in cell phone, It's an exception. But if you have diarrhea, It's a unnormal, It's not exception. For example, opening file failure is exception, You should (throw exception). but age<0 is a indication of a bug.

```

1. FILE *f = fopen("hr.dat", ...);
2. if(f==NULL){ // It's an exception, so don't use assert here.
3.     return -1;
4.     throw runtime_error();
5. }
6.
7. assert(age>=0); //Here use assert.
8. fprintf(f, "%d", age);

```

## 10.2.2 Trace

- you can implement trace, such as it in the MFC

```

1. #if defined NDEBUG
2. #define TRACE( format, ... )
3. #else
4. #define TRACE( format, ... ) printf("%s::%s(%d)" 
5. format, __FILE__, __FUNCTION__, __LINE__, __VA_ARGS__ )
6. #endif

```

## 10.3 Handling exceptions

### 10.3.1 errno in C

- There are three common error methods:

1. global error code(C) or error state(C++).
2. return value.
3. Exception.

```

1. //1) global error code , C language method
2. errno() and strerror()
3. //1) status in object , C++ method
4. cin.fail()
5.
6. //2) return value by parameter or return value
7. //3) Exception.

```

- `errno()` and `strerror()` is a typical C language style.
- Besides above, you can use some custom function pointer to do some custom error handling behavior, such as `set_new_handler` function for new operator
- 1. In general, you should detect errors by checking return values, and use `errno` or `perror()` only to distinguish among the various causes of an error, such as "File not found" or "Permission denied."

```

1. FILE * pFile = fopen ("unexist.ent","rb");
2. if (pFile==NULL)
3. perror ("The following error occurred");

```

- 2. It's only necessary to detect errors with `errno` when a function does not have a unique, unambiguous, out-of-band error return (that is, because all of its possible return values are valid; one example is `atoi()`). In these cases (and in these cases only; check the documentation to be sure whether a function allows this),

```

1. #include <cerrno>
2. errno = 0
3. // set it zero before call any math library function.
4. double not_a_number = std::log(-1.0);
5. if (errno == EDOM) {
6.     std::cout << "log(-1) failed: " <<
7.     std::strerror(errno) << '\n';

```

- 3. You can detect errors by setting `errno` to 0, calling the function, and then testing `errno`. (Setting `errno` to 0 first is important, as no library function ever does that for you.)

- C11 provide `fenv.h` file to expand `errno`

```

1. #include <math.h>           /* math_errhandling */
2. #include <errno.h>          /* errno, EDOM */
3. #include <fenv.h>
4. /* feclearexcept , fetestexcept , FE_ALL_EXCEPT, FE_INVALID */
5.
6. #pragma STDC FENV_ACCESS on
7. errno = 0;
8. if (math_errhandling & MATH_ERREXCEPT)
9. feclearexcept(FE_ALL_EXCEPT);
10.
11. sqrt (-1);
12. if (math_errhandling & MATH_ERRNO) {
13.     if (errno==EDOM) printf("errno set to EDOM\n");
14. }
15.
16. if (math_errhandling &MATH_ERREXCEPT) {
17.     if (fetestexcept(FE_INVALID)) printf("FE_INVALID raised\n");
18. }

```

### 10.3.2 exceptions in C++

- Exception mainly deal with runtime error: At the same time , these potentially recoverable error. Such as open an unavailable file re request more memory than is available, they can be exceptions.
  1. A file write operation failed or file access operation because failed file change or non-exist.
  2. No enough memory
  3. invalid value, which come from user input, not come from you error code logic.
  4. System communication software invalid protocol, format, or no response.
- unwinding the stack has cost problems, it make programme 10% larger and slowlier. So you can use `-fno-exceptions` to stop it.
- Exception specification is add throw in the end of function, no throw means that it will throw any exceptions, and throw() means it will not throw any exceptions. Throw( e1, e1) means that it will throw two kinds of exceptions. In C++, this feature has been unsupported and only one left is use throw() to indicates that it will not throw any exceptions. At the same time, exception specification doesn't use very well with template.
- A simple exception can be a char string, than use "const char\* c" to catch it. A more complicate example is exception class, and you can define the exception class, and throw exception\_class, than use exception\_class & ec to catch it. You don't need explicit define exception\_class object and throw this object; throw exception\_class(); will call constructor and throw this unnamed object, it is ok.
- You can build exception class inside the C++ standard exception system. It need to derive you class from exception, and redefine function what(); if you exception class has very tight relationship with you real class, it can be declared as nested class.

```

1. Class my_ex :public std::exception{
2.     const char* what(){return "my_ex_reason_is_here";}
3. }
```

- Arranging the catch blocks in inverse order of derivation.
- Standard exception includes `logic_error`, `domain_error`, `runtime_error`, `invalid_argument`, `out_of_bounds`, `range_error`, `overflow_error`.
- **Catch use reference,** 1) It will support polymorphic exception, and catch exception from specify to generic.2) It will avoid extra coping
- Empty throw means that you throw present exception again.

```

1. catch(my_base_ex &me){
2.     .....
3.     throw;
4.     //not use throw me,
5.     //because maybe me is child class of my_base_ex;
6. }
```

- In you destructor, don't throw any exception, or catch all the exception in side of it. Or it will call stop the application. see more effective C++ exception chapter.
- Compared with return error code, exception has advatnage:

1. Can catch deeper called function exceptions. If you want to use return value, deeper called function is hard to deal with.
2. Difficult to return value, 1) no return value, such as class constructor, 2) all return value is normal value, such as atoi().
3. Make happy path and error-handle path clearly.
4. You can't omit exception, Any unhandled exception will terminate in the end.

## 10.4 Conclusion

- Consciously specify, and conscientiously apply, what so many projects leave to adhoc (mis)judgment: Develop a practical, consistent, and rational error handling policy early in design, and then stick to it. Ensure that it includes:
  1. Identification: What conditions are errors.
  2. Severity: How important or urgent each error is.
  3. Detection: Which code is responsible for detecting the error.
  4. Propagation: What mechanisms are used to report and propagate error notifications in each module.
  5. Handling: What code is responsible for doing something about the error.
  6. Reporting: How the error will be logged or users notified
- For different runtime errors, you can take different actions. And It's depends on you context of application.
  1. error is just warning, log or show it to user, then continue;
  2. error can be resolved inside a function, the whole application can be continue after you resolve or retry, such as input error.
  3. error is serious, application can't continue, stop (call exit). please debug me (call abort)
  4. set global error code or return error code or throw exception. then caller decide what to do (return a error code or use exception)
- Pay attention, exception and return a error code has the same philosophy. **let caller decide what to do next.**
- Exception handling is highly dependent on your application context. So it should be designed into a program rather than just added on.
- Confusing logical error with runtime error. **For logical error, you need to rewrite code to fix it. For runtime error, you need to throw it and catch it to respond.** logical error should use assert or unit test to find the as early as possible. For example, in f(Foo\* p), p is nullptr. There are two possibilities:
  1. they are passing nullptr because they got bad data from an external user (for example, the user forgot to fill in a field and database connection fail) you should throw an exception since it is a runtime situation (**i.e., something you can't detect by a careful code-review; it is not a bug**).
  2. they just plain made a mistake in their own code. you should definitely fix the bug in the caller's code. but you must not merely change the code within f(Foo\* p); you must, must, MUST fix the code in the caller(s) of f(Foo\* p).

- In previous example, You can use assert first, after you code-review, found that It's not bug, but cause by an exception, such as net disconnect. so you can change assert to throw. THAT IS A GOOD STRATEGY!

```

1. fun(int* pi, int j){
2.     assert(j < 10);
3.     if (pi == nullptr)
4.         //don't throw it, It's logical error. you need to check
5.         //caller of fun why pass a nullptr into fun
6.
7.     if(file.open())
8.         throw runtime_error();
9.         // file maybe deleted. It's not your logic error but runtime error.
10. }
```

- Use all throw and catch to replace return-code. For a simple function, just return one error code and user will not forget to test return code. this time, return-code method maybe is better.
- For object constructor, always use exception and try block.
- When to use assert ?

1. your problem comes from your own bad code, it's better to use ASSERTs. Including you coding error or addition overflow.
2. bugs in your program are not something the user can handle, User can do nothing when he face "age should be not negative" unless age is inputted by himself(at this time, you should use return error code)
3. you have to stop your application. If you write negative age back to database, It may cause futuristic error for other user later, and It's very difficult to trace back.

- When to throw a exception( or return error code)?
  1. As a general rule of thumb, throw an exception when your program can identify an **external problem** that prevents execution.
  2. identify problems that program cannot handle and tell them about the user, because user can handle them.
  3. For example, no memory space, no file exit, no net connection , no object construct and get invalid data from sever etc.

- When to use try... catch block?
  1. Catch an exception where you can do something useful with it.
    - (a) You can actually handle the exception. your catch clause deals with the error and continues execution without throwing any additional exceptions. My caller never knows that the exception occurred.
    - (b) So I can have a catch clause that does blah blah blah, after which I will rethrow the exception. In this case, consider changing the try block into an object whose destructor does blah blah blah. For instance, if you have a try block whose catch clause closes a file then rethrows the exception, consider replacing the whole thing with a File object whose destructor closes the file. This is commonly called RAII
    - (c) Show some message or log exception, or give user a list options to select. then rethrow.
  2. For an atomic operation, there is only one try block. That is to say, inside one function, just include one try block

```
1. atomic_fun() {
2.     try{
3.         //complex operation .
4.     }
5.     catch (...) {
6.         cout<<"I can't do it "<<endl;
7.         throw;
8.     }
```

- Exception in C++ is a tool; use it properly and it will help you; **but don't blame the tool if you use it improperly.** "Wrong exception-handling mindsets" in c++ FAQ website section 17 is good topic. You need to read it again if you have time.

# Chapter 11

## functional programming



# Chapter 12

## concurrent

### 12.1 data race

- data race and race condition, you can see a good reference page: "Race Condition vs. Data Race", you can google it.
- Even one thread read, another thread write, It will cause data race. A good reference page is "Benign data races: what could possibly go wrong?".
- Another good paper about data race is "Fun with Concurrency Problems".
- Even increment is not atomic operation. It will change it to three assemble statements. So If there are two threads, thread 1 just move to reg, then another thread 2 finish ++, next, thread1 change back, The value will be overwrite mem. The result is: op is incremented 1 although two thread call op++.

```
1. op++;
2.
3. move mem reg
4. // Thread1 change to thread 2 here.
5. add reg 1
6. move reg mem
```

- In visual studio. You can use debug view assemble language produced by compiler.

### 12.2 synchronization



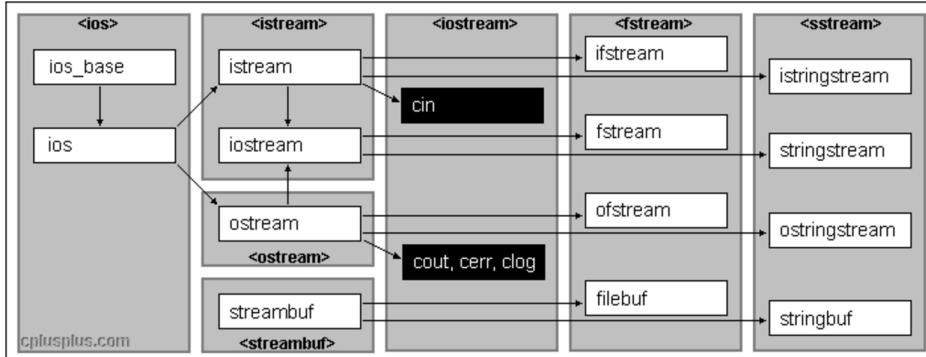
# Chapter 13

## I/O

### 13.1 I/O basic

#### 13.1.1 I/O basic knowledge

- Neither C++ nor C has built input and output in the language. They use functions(C) or other I/O objects(C++) in language library.
- C++ I/O class and head file



1. **iofs** stand for three head files `<iostream>` `<fstream>` and `<sstream>`. `<iostream>` includes `<ios>` automatically. They are three main header file you should include in your C++ application.
2. three classes: `iostream`(`streambuf`), `fstream`(`filebuf`), `stringstream`(`stringbuf`) and four pre-defined object: `cin`, `cout`, `clog` and `cerr`
3. `clog` is just like `cerr`, but it buffers its output.
4. C++ normally flushes the input buffer when you press enter. For output to the display, C++ program normally flushes the output buffer when you transmit a newline character, or reaches an input statement.
5. `>>` and `<<` don't need to format strings, C++ will automatically convert them, it's better than `printf` and `scanf` in C language.
6. through inheritance, `fstream` and `cin`(`cout`) share the same usage. **All the knowledge can be used directly in fstream.** I like it the most.

## 13.2 Input

### 13.2.1 Input basic knowledge

- For Input, you need to master **One basic idea, Two languages, Three data type.**
- One basic idea: In order to make continuously input, you need to use while(inputMethod), When two things happens:
  1. user want to end input(ctrl+D) or read the end of File;
  2. read fail (for example, `cin>> int`, but input letter 'a'),

InputMethod will return false. Then you need use some flag or status to tell the difference between EOF or error inside of while loop.

- Two languages is c and c++, they use the different inputMethod. three data types are: number and word(no space in middle), character(white), and string(include space in middle)

- two languages common used input method

|     | Number and non-white character word<br><code>scanf("%d %f %c %s",&amp;i, &amp;f,&amp;c);</code> | Character (including white-character)<br><code>int a = getchar();</code> | string(line)<br><code>fgets(stdin, char*p, n )</code>                                                             |
|-----|-------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| C   |                                                                                                 |                                                                          |                                                                                                                   |
| C++ | <code>cin&gt;&gt;i&gt;&gt;f&gt;&gt;c&gt;&gt;w;</code>                                           | <code>cin.get(char &amp; c);</code><br><code>Ch = cin.get();</code>      | <code>cin.get( char *p, n);</code><br><code>cin.getline(char *p, n);</code><br><code>getline(cin, string);</code> |

- In scanf, You need to specify exact data type when you read.
  1. h: short int or short unsigned. Example: `%hd` or `%hu`.
  2. l: a long int or long unsigned, or double (for `%f` conversions.) Example: `%ld`, `%lu`, or `%lf`.
  3. L: The value to be parsed is a long long for integer types or long double for float types. Example: `%Ld`, `%Lu`, or `%Lf`.
  4. \*: Tells `scanf()` do to the conversion specified, but not store it anywhere. This is what you use if you want `scanf()` to eat some data but you don't want to store it anywhere; you don't give `scanf()` an argument for this conversion. Example: `%*d`.

- `scanf("%c" &c)` will read any character, including whitespace character. If you want to skip any whitespace, you can use space before `%c`.

```

1. while(true){
2.   scanf ("%c",&c);
3.   //scanf(" %c",&c);
4.   printf("you_input: %d" c);
5. }
6. // When you input a(enter)
7. //output will be like
8. you input: 97(a value)
9. you input: 10(enter value)

```

- `cin>>c` will not read white character(tab, space , newline), If you want to read them from input buffer, You should use `getchar()` or `cin.get()`; If you want the user to input his or her name

```

1. while(true){
2.   cin>>c;
3.   cout<<"you input : "<<c<<endl;
4. }
5. // When you input "      a(enter)"
6. // output will be like
7. you input: a
8. then cursor wait for here.

```

- Using `cin>>` or `scanf` will terminate the string after it reads the first space. The best way to handle this situation is to use the function to read a line;

- Read word and line:

```

1. //read a word until reach white character .
2. scanf("%s",char_array) //c
3. cin>>char_array or ; //c++
4. cin>> str;
5.
6. //line
7. gets(char_array) //c
8. fgets(char_array, n , FILE *) //recommend to use this for safety .
9.
10. cin.getline(char *,int n) //c++ read and discard newline
11. cin.get(char *,int n) //not read newline
12. std::getline(istream& is , string& str)

```

- `cin.read` function has the same interface with `cin.get`, but it doesn't append a null character to input, It's not intend for keyboard input, but for binary format of file

- **For C++, three get, two getline, other use >>;**

- Difference between `cin.get(char)` and `int = cin.get()`

```

1. while(cin.get(c))
2.   // use cin.get(char) in reading loop
3.
4.   cin.get() != '\n'
5.   //use cin.get() return character to test sth .
6.
7.   cin.get()!=EOF
8.   //When used in EOF, you have to use int
9.   //because EOF may not be expressed by char type

```

- Three confused functions: `cin.get` and `cin.getline` are almost the same thing.

```

1. cin.get ( char* s, streamsize n, char delim );
2. cin.getline (char* s, streamsize n, char delim );
3. istream& getline (istream& is , string& str);

```

1. For each line, if you don't know the max length, just use `getline(cin, string)`. You don't need to input any length.(you can reserve length of string if you want to avoid allocation of memory)
2. `cin.get()` doesn't discard `delim` from input stream. However, `cin.getline()` will read and discard newline. (It's easy for you to remember, because, `line` is defined by newline character)

3. In `cin.getline(char* s, int n)` The failbit flag is set if the function extracts no characters(newline is a character), Or if the delimiting character is not found once (n-1) characters have already been written to s. Note that if the character that follows those (n-1) characters in the input sequence is precisely the delimiting character, it is also extracted and the failbit flag is not set.
4. In `cin.get(char* s, int n)`. The failbit flag is set only if the function extracts no characters.
5. If you are talking about the newline character from a console input,it makes perfectly sense to discard it, So use `cin.getline()`. Or you don't want to customized flexible reading method, just read a line from a file. please use `cin.getline()`.
6. `cin.get(char* s, n)` is more flexible than `cin.getline`. Because when it read to the array is full, It doesn't set failbit. At this time you can use `gcount()` or `peek()` to see if next character is new line. It's more customized than `cin.getline()`;

### 13.2.2 Input error

- **For the error of input: Just remember C language return EOF and NULL, and C++ return all false.**

|   |                                                           |                                                            |                                                         |
|---|-----------------------------------------------------------|------------------------------------------------------------|---------------------------------------------------------|
| C | Number, non-white<br>scanf return EOF<br>cin return false | character, white<br>getchar return EOF<br>cin return false | string<br>fgets return NULL<br>cin.getline return false |
|---|-----------------------------------------------------------|------------------------------------------------------------|---------------------------------------------------------|

- When you can't continue input, you need to know if it's end of file or fail. In C language, you can use `feof` and `ferror` function, in C++, you can use `cin.eof()`, `cin.fail()` or `cin.bad()` three functions. `cin.bad()` means serious system error happens, and input buffer can't be consistent and can't be recovered anymore.
- When `failbit` or `badbit` are set. `fail()` return false, So you need to judge it by `bad()` again if this information is important for you. Usually, `bad()` is not use very often.
- `eofbit` is interesting topic. when you reach(not read) the "logical end-of-file", it will not be set, **It's set by a read function, not by position** Instead, You can use `std::ifstream::peek()` to check for the "logical end-of-file".
- **eof is set by last read EOF position** When you read EOF position, `failbit` is set and `eof` is set too. So If EOF is set, `failbit` must set too. If only `failbit` is set, it means that input error happen. Under these two conditions, `istream` will return false by operator `bool`.
- In [http://en.cppreference.com/w/cpp/io/basic\\_ios/operator\\_bool](http://en.cppreference.com/w/cpp/io/basic_ios/operator_bool) . You can see `eofbit` are true and `failbit` are false, operator `bool` return true. I don't know when it will happen. **eof() function returns "true" after the program attempts to read past the end of the file.** But when read action set `eofbit`, it will set `failbit` at the sametime, so `bool` operator will return false because `failbit` are set.
- How do you know if you read an incorrect input word, or if you are at the end of the file? This is when you use `cin.eof()` or `feof()` in C language.
- you can use `clear()` and `setstate()` to set the states, Why? It's just depends on what the program is trying to accomplish. It provides a means for you to change the state. `setstate()` is different with `clear()`, `clear()` clear all the status bit. but `setstate` just set specific bit.
- You can use `exceptions()` method to control if exceptions will throw, when the `eofbit`, `failbit` and `badbit` is set.

- setstate() is to provide a means for input and output functions to change the state. For end user, we don't use it very often. We just use clear() more.
- for rdstate() it will return all the bit value, then you can use & to certain bit( such as failbit) to check if the bit has been set. But here, you can use fail() directly, so we don't use this function very often.

```

1. std::ifstream is;
2. is.open ("test.txt");
3. if ( (is.rdstate() & std::ifstream::failbit) != 0 )
4. std::cerr << "Error opening 'test.txt'\n";

```

- **clear() function is very important, when cin.fail() return true, you have to use clear(), otherwise all he below cin operation will not work.** See below example.

```

1. char ch, str[20];
2. cin.getline(str, 5);
3. cout<<"flag1:"<<cin.good()<<endl;      // check good bit
4. cin.clear();    //without, cin>>ch below will not work at all
5. cout<<"flag1:"<<cin.good()<<endl;      // check good bit again
6. cin>>ch;
7. cout<<"str :"<<str<< " ch :" <<ch<<endl;
8.
9. input:
10. 12345[Enter]
11. output:
12. flag1:0 // good() return false
13. flag2:1 // good() return true after clear().
14. str:1234 ch:5

```

### 13.2.3 Input Pattern

- It is bad idea to test the stream on the outside and then read/write to it inside the body of the conditional/loop statement. This is because the act of reading may make the stream bad. It is usually better to do the read as part of the test.

```

1. while(!cin.fail()){ // that is bad programming style
2.   cin>>i; //here may make stream fail().
3.   .... //then i is not valid value
4. }

```

- If you just want to input, You don't want to know eof or deal with failbit. You can use below:

```

1. while(scanf ("%d",&i) != 1)
2. while((int c = getchar())!=EOF)
3. while( fgets(line, sizeof(line), fp) != NULL )
//in c language, use these to exit loop!
5.
6. while(cin>>input)
7. while(cin.get(p, 20) )
8. while(getline(ifstream, string)) {
//in C++ language, use bool operator to exit loop.
9. //do some useful things. //input is right.
10. }
11.

```

- If you want to know eof or deal with error. You can use below code. When you press enter, the read will end. When you input letter, it will get rid of these letter until you input number.

```

1. While(true) // use break to exit loop;
2. {
3.     cin>>i // or getline(ifstream, string);
4.     if(cin.eof()) { //If it's EOF
5.         cout<<"EOF encountered"<<endl
6.         //break;
7.     }
8.
9.     if(cin.fail()) // Invalid input
10.    {
11.        cin.clear(); //Important. clear state and buffer
12.        while(cin.get()!='\n') //get rid rest of line,
13.            continue;
14.        cout<<"please input a number"<<endl;
15.        continue;
16.    }
17.    ... //do some useful things. //input is right.
18.

```

- In the previous example, Why do we need to distinguish eof and fail? When fail happen, maybe there are invalid character in buffer. After clean the buffer, I can continue to read input from input buffer. Three methods to clean invalid character in the buffer

```

1.     cin.clear(); //Important. clear state and buffer
2.     while(cin.get()!='\n')
3.         continue; // method 1
4.
5.     while(!isSpace(cin.get()))
6.         continue; //method 2
7.
8.     basic_istream& ignore(streamsize Count = 1, \
9.     int_type _Delim = traits_type::eof()); //method 3
10.    cin.ignore(5, 'a');
11.    cin.ignore(numeric_limits<streamsize>::max(), '\n');

```

- You can use istream\_iterator, It can save you some trouble to judge EOF.

```

1. class A{
2. private:
3.     int x;
4.     int y;
5.     friend istream& operator>>(istream& in, A& a);
6.     friend ostream& operator<<(ostream& out, const A& a);
7. };
8.
9. istream& operator>>(istream& in, A& a){
10.    in>>a.x>>a.y;
11.    return in;
12. }
13.
14. ostream& operator<<(ostream& out, const A& a){
15.    out<<a.x<<" "

```

```

21. copy(istream_iterator<A>(cin), istream_iterator<A>(),
22.       back_inserter(v));
23. copy(v.begin(), v.end(), ostream_iterator<A>(cout, "\n"));
24. }
```

- In summary, It's better just use judgment to exit end loop. If you need to different specific action to take to deal with EOF or error. Use while(true), then use eof() of feof() fail() or clear() functions in c++ and c to deal with and break the loop;

### 13.3 output

- For cout, It can recognize type automatically, and It is extensible, You can redefine << operator so that cout can recognize your data type.

```

1. class Foo{
2.     friend ostream & operator<<(ostream& s, const Foo &r);
3. }
4.
5. ostream & operator<<(ostream& s, const Foo &r){
6.     s<<Foo.a<<Foo.b<<endl;
7. }
```

- How to print pointer address in C and C++?

```

1. char* amount = "dozen";
2. cout<< amount ; //print "dozen" string
3.
4. cout<<(void*) amount; //prints the address of pointer
5. printf("%p", (void*) p);
```

- Format is key point for Output. you need to know the common manipulator to control the output format.
- Number base manipulators: hex, oct and dec; Field widths: Width, fill, precision Setf()
- You don't need to know the details, just name of them. When you want to use them go to reference website to look. A detail format manipulators can be seen on C++ primer P1090. You need to include <iomanip>
- << is a bitwise left-shift operator in C language, but in C++, we overloaded it in ostream class, cout is an object of ostream,
- You can use cout<<flush to force flushing the output buffer
- cout.write can be used to output a string, It will output certain length string, even reach the end of string.

### 13.4 other stream

#### 13.4.1 file

- When you studied cin and cout very well, you will find that file operation is so easy. just change cin or cout to your ifstream , ofstream or fstream object.

```

1. std::fstream ifs;
2. ifs.open ("test.txt", ios_base::in | ios_base::binary);
3. if (!ifs.is_open())
4.     exit(1);
5. char c = ifs.get();
6. // use all previous input methods
7. ifs.close();

```

- Only read ifstream; Only write ofstream; write and read fstream.
- For ios\_base::binary mode, use write() and read() function.
- For writing, pay attention to the difference between ios\_base::trunc and ios\_base::app
- Random access is used mostly on binary file. Because the position can be pinpointed exactly. For seekg() for input, and seekp() for output.(p is put, g is get) It move the pointer. tellp() and tellg() function. It tell the position of pointer.

### 13.4.2 buffer and string buffer

- stringstream is a convenient way to manipulate strings like an independent I/O device. Sometimes it is very convenient to use stringstream to convert between strings and other numerical types. The usage of stringstream are much the same with iostream, so it is not a burden to learn.
- You need to build a stringstream from a string, or convert a stringstream back to a string.

```

1. stringstream outstr;
2. //change number to a text
3. outstr<<"salary_value"<<123333.00<<endl;
4. string str = outstr.str() //change to string
5.
6. //change text to number
7. istringstream Instr(str);
8. while(instr>>number)
9. cout<<number<<endl
10.
11. //////////below is C language/////////
12. char sentence []="Yan is 41 years old";
13. char str[20];
14. int i;
15. sscanf (sentence, "%s %*s %d", str, &i);
16. /* means input will be ignored. So str = Yan, i = 41.
17. sprintf (....);

```

## 13.5 manipulate stream

1. peek return the next character from input without extracting from the input stream. For example, you want to read input up to the first newline or period.

```

1. char input[80];
2. int i = 0;
3. while( (ch=cin.peek()) != '.' && ch != '\n')
4.     cin.get(input[i++]);
5. input[i] = '\0';

```

2. **gcount()** method returns the number of characters read by the last unfromatted extraction method. That means character read by the a `get()`, `getline()`, `ignore()`, or `read()`, but not extraction operator.
3. **putback()** function inserts a character back in the input buffer.



# Chapter 14

## New Feature in modern C++

### 14.1 New Type

#### 14.1.1 New int

- Add three new data types " long long", "char16\_t" and "char32\_t". standard define:
  1. int must be at least 16 bits
  2. long must be at least 32 bits
  3. long long must be at least 64 bits
- If you need a specific integer size for a particular application, rather than trusting the compiler to pick the size you want, #include <stdint.h> (or <cstdint>) so you can use these types:**All these types have \_t in the end.**

```
1. int8_t and uint8_t  
2. int16_t and uint16_t  
3. int32_t and uint32_t  
4. int64_t and uint64_t
```

- The fixed-width integers have two downsides: First, they may not be supported on architectures where those types can't be represented. They may also be less performant than the built-in types on some architectures. To help address the above downsides, C++11 also defines two alternative sets of integers. They are int\_least and int\_fast
- The 8-bit fixed-width integer types. Any compiler-specific fixed width integers – for example, Visual Studio defines \_\_int8, \_\_int16, etc...
- integer best practices:
  1. int should be preferred when the size of the integer doesn't matter (e.g. the number will always fit within the range of a 2 byte signed integer). For example, if you're asking the user to enter their age, or counting from 1 to 10, it doesn't matter whether int is 16 or 32 bits (the numbers will fit either way). This will cover the vast majority of the cases you're likely to run across.
  2. If you need a variable guaranteed to be a particular size and want to favor performance, use std::int\_fast\_t.
  3. If you need a variable guaranteed to be a particular size and want to favor memory conservation over performance, use std::int\_least\_t. This is used most often when allocating lots of variables. Avoid the following if possible:

4. Don't use unsigned types, unless you have a compelling reason.

- nullptr is always a pointer type, Don't use NULL any more. NULL is just 0 In C language, but in strong type C++ language, NULL can be ambiguity in f(int); and f(foo \*);
- Syntactic improvement of C++ 14:
  1. Prefix 0b and are followed by binary digits.
  2. To use the single quote character, making the million dollar value render in C++ as: 1'000'000.00.

```
1. int val = 0b11110000;
2. std::cout << 0b1000'0001'1000'0000;
3. std::cout << 300'000.00;
```

- Note that the syntax of C++ attribute-tokens might seem a bit unfamiliar. The list of attributes, including [[deprecated]], comes after keywords like class or enum, and before the entity name. A good article can be found if you google "Marking as deprecated in C++14". This feature is useful for a large and long term project. You can't just delete class **flaky**, because other people maybe is using it in their code.

```
1. class [[deprecated]] flaky {
2. };
3. [[deprecated("Consider using something other than cranky")]]
4. int cranky() {
5.     return 0;
6. }
7. // below two statements produce compiler warning.
8. flaky f;
9. return cranky();
```

### 14.1.2 new container

- New container: forward\_list, unordered\_map unordered\_set. The latter two support implementing hash tables. std library provide basic hash function for some basic type, such as int and string. If you have you own class, you can use boost::hash\_combine.
- To support user-defined key types in std::unordered\_set<Key> and std::unordered\_map<Key, Value> one has to provide operator==(Key, Key) and a hash functor

```
1. struct X { int id; /* ... */ };
2. bool operator==(X a, X b) { return a.id == b.id; }
3.
4. struct MyHash {
5.     size_t operator()(const X& x) const { return std::hash<int>()(x.id); }
6. };
7.
8. std::unordered_set<X, MyHash> s;
```

- You are expressly allowed and encouraged to add specializations to namespace std\*. The correct (and basically only) way to add a hash function is this:

```
1. namespace std {
2. template <> struct hash<Foo>{
3.     size_t operator()(const Foo & x) const{
4.         /* your code here, e.g. "return hash<int>()(x.value); */
```

```

5. }
6. } ;
7. }
8.
9. std::unordered_set<Foo> s;

```

- A good std::hash example can be found in [cppreference.com](http://cppreference.com)
- std::array is just a class version of the classic C array. Its size is fixed at compile time and it will be allocated on the stack.
- std::vector is a small class containing pointers into the heap. (So when you allocate a std::vector, it always calls new.) They are slightly slower to access because those pointers have to be chased to get to the arrayed data... But in exchange for that, they can be resized and they only take a trivial amount of stack space no matter how large they are.
- Because it does bounds checking, at() is slower (but safer) than operator [].
- array support copy and assignment between the same size.

```

1. //Making a new array via copy
2. auto a3 = a2;
3.
4. //This works too:
5. auto a4(a2);
6.
7. a5 = a2 // a2 and a5 must have the same size.

```

- Fixed arrays decay into pointers, losing the array length information when you pass it to a function.
- Like std::vector, std::array doesn't implicitly decay into a raw pointer. If you want to use the underlying std::array pointer, you must use the data() member function. For example, let's assume you are using an API with a C-style buffer interface: For other new function, you can use std::array& as a function parameter.

```

1. void carr_func(int * arr, size_t size){
2.     std::cout << "carr_func->arr:" << arr << std::endl;
3. }
4.
5. //Error:
6. //carr_func(a2, a2.size());
7.
8. //OK:
9. carr_func(a2.data(), a2.size());

```

- Use std::vector unless (a) your profiler tells you that you have a problem and (b) the array is tiny.

```

1. #include <iostream>
2. #include <array>
3. void printLength(const std::array<double, 5> &myarray) {
4.     myarray.sort();
5.     std::cout << "length:" << myarray.size();
6. }
7.
8. std::array<double, 5> myarray { 9.0, 7.2, 5.4, 3.6, 1.8 };
9. printLength(myarray);

```

- For most of time, you should use `std::array` and `std::forward_list` firstly if you don't have any strong reason against using them.

## 14.2 range base

- A range-based for loop is introduced in C++11. Use `auto&` to avoid copying each element. Below are highly recommended when yo use STL container.

1. For observing the elements, use the following syntax. For observing the elements in generic code. since we can't make assumptions about generic type T being cheap to copy, it's safe to always use `for (const auto& elem : container)`. (This won't trigger potentially expensive useless copies, will work just fine also for cheap-to-copy types like `int`, and also for containers using proxy-iterators, like `std::vector<bool>`.)

```
1. for (const auto& elem : container)
2. // capture by const reference
```

2. If the objects are cheap to copy (like `ints`, `doubles`, etc.), it's possible to use a slightly simplified form.

```
1. for (const auto elem : container)
2. // capture by value
```

3. Of course, if there is a need to make a local copy of the element inside the loop body, capturing by value (`for (auto elem : container)`) is a good choice.

4. For modifying the elements in place, use:

```
1. for (auto& elem : container)
2. // capture by (non-const) reference
```

5. If there are pointer inside the container. You don't need star symbol explicitly, but it give more information to the variable, so I recommend using it.

```
1. //make sure Container contain pointer
2. for (auto *ptr : Container){ ptr->change(); }
3. for (const auto *ptr : Container){ ptr->read(); }
```

- For `vector<bool>`, below code doesn't work, because `std::vector` template is specialized for `bool`, with an implementation that packs the bools to optimize space (each boolean value is stored in one bit, eight "boolean" bits in a byte). Because of that (since it's not possible to return a reference to a single bit), `vector<bool>` uses a so called "proxy iterator" pattern. A "proxy iterator" is an iterator that, **when dereferenced, does not yield an ordinary `bool &`, but instead returns (by value) a temporary object**, which is a proxy class convertible to `bool`.

```
1. int fun() { ... };
2. int& a = fun(); // reference can't bind to temporary prvalue.
3.
4. vector<bool> v = {true, false, false, true};
5. for (auto& x : v)
6.     x = !x;
```

- If you want to read, you can use two below methods:

```

1. for (auto i : bv)
2.
3. for (auto const & i : bv)
4. cout<<i<<endl;

```

- If we want generic code to work also in case of proxy-iterators, the best option is for (auto&& elem : container). This will work just fine also for containers using ordinary non-proxy-iterators, like std::vector<int> or std::vector<string>. It also can modify vector<bool>

```

1. for (auto&& elem : container)

```

- The range-based version is for: You want to do something with every element in the container, without mutating the container(insert or erase). If you want to mutate the container, Follow idea introduced in Container-Erasure section below.

1. For any container, you can **NOT** mutating container by calling insert or erase inside loop.
2. If you really want to change a container, for a sequential container, you can use range-based function to change elements.
3. For any associate container, only visit each element. If you want to change a key, delete it first, then modify, in the end insert new key again.

## 14.3 lambda

- lambda basic syntax and an example.

```

1. [ captures ] (parameters) -> returnTypesDeclaration { ..; }
2. int sum = 0, divisor = 3;
3. vector<int> vc { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45 };
4. for_each(vc.begin(), vc.end(), [divisor, &sum] (int y){
5.     if (y % divisor == 0){
6.         cout << y << endl;
7.         sum += y;
8.     }
9. });
10. cout << sum << endl;

```

- Lambda is used on complex type container. You can use typedef.

```

1. typedef std::vector< std::pair<int, std::string> > Record_t;
2. // typedef is your good friend, reuse it below
3. Record_t k1;
4.
5. int find_it(std::string value, Record_t const& stuff){
6.     auto fit = std::find_if(stuff.begin(), stuff.end(),
7.                            [value](Record_t::value_type const& vt) -> bool
8.                                { return vt.second == value; });
9.     if (fit != stuff.end())
10.         return fit->first;
11. }

```

- This auto-declaration defines a closure type named factorial that you can call later instead of typing the entire lambda expression (a closure type is in fact a compiler-generated function class):

```

1. auto factorial = [](int i, int j) {return i * j;};
2.
3. int arr{1,2,3,4,5,6,7,8,9,10,11,12};
4. long res = std::accumulate(arr, arr+12, 1, factorial);
5. cout<<"12! = "<<res<<endl; // 479001600

```

- Under the final C++11 spec, if you have a lambda with an empty capture specification, then it can be treated like a regular function and assigned to a function pointer. Here's an example of using a function pointer with a capture-less lambda:

```

1. typedef int (*func)();
2. func f = [] () -> int { return 2; };
3. f();

```

- A delegate. When you call a normal function, all you need is the function itself. When you call a method on an object, you need two things: the function and the object itself. It's the difference between `func()` and `obj.method()`. Starting with some code that again expects a function as an argument, into which we'll pass a delegate. **lambda is like a glue to combine for\_each and class MessageSizeStore()**.

```

1. class MessageSizeStore{
2. public:
3.     MessageSizeStore () : _max_size( 0 ) {}
4.     void getMaxMessage (const std::string& message) { ... }
5.     int getSize () { ... }
6. private:
7.     int _max_size;
8. };
9.
10. MessageSizeStore size_store;
11. for_each(s.begin(), s.end(), [&] (const std::string& message) {
12.     size_store.checkMessage( message );
13. });

```

- Why functor is better than function pointer? because functor can be inline.

- Why lambda is better than functor?

1. lambda will produce a unnamed functor class. Not pollute name space. Frequently (not in this example, though) the name of the functor class is much less expressive than its actual code.
2. It can be inline too.
3. It can access all the automatic variable.
4. It's much less verbose.
5. Placing the code closer to where it's called improves code clarity.

```

1. int count3 = 0;
2. std::count_if(v.begin(), v.end(),
3. [&count3](int x){count3 += (x%3 == 0)} );
4. //how many number can be divided by 3.

```

## 14.4 Other New Feature

### 14.4.1 decltype

- decltype and Trailing Return type(C++11 new feature)

```

1. template<typename T1, typename T2>
2. void f(T1 x, T2 y) {
3.     decltype(x+y) xpy = x+y;
4. }
5.
6. //with return value;
7. template<typename T1, typename T2>
8. auto f(T1 x, T2 y) ->decltype(x+y) {
9.     x+y;
10. }
```

- How to make your template class has more generic use for client, "exceptional C++" item 5 introduces more idea in this topics.

### 14.4.2 alias declaration

- Alias declaration is better than typedef when you declare a function pointer, It's more verbose.

```

1. // FP is a synonym for a pointer to a function taking an int
2. // and a const std::string& and returning nothing
3. typedef void (*FP)(int, const std::string&);
4.
5. using FP = void (*)(int, const std::string&);
6. // alias declaration
```

- C++11 doesn't support typedef template directly, only can be put inside a struct.

```

1. template<typename T> // MyAllocList<T>
2. using MyAllocList = std::list<T, MyAlloc<T>>;
3. // is synonym for std::list<T, MyAlloc<T>>
4. MyAllocList<Widget> lw; // client code
5.
6. template<typename T>
7.     struct MyAllocList {
8.     typedef std::list<T, MyAlloc<T>> type;
9. };
10. // MyAllocList<T>::type is synonym for std::list<T, MyAlloc<T>>
11. MyAllocList<Widget>::type lw; // client code
```

- MyAllocList<T>::type is thus a dependent type, and one of C++'s many endearing rules is that the names of dependent types must be preceded by typename.

```

1. template<typename T>
2. class Widget { // Widget<T> contains
3. private: // a MyAllocList<T>
4.     typename MyAllocList<T>::type list; // as a data member
5. };
```

- A better way is to use Alias name

```

1. template<typename T>
2. using MyAllocList = std::list<T, MyAlloc<T>>; // as before
3.
4. template<typename T>
5. class Widget {
6. private:
7.     MyAllocList<T> list; // no "typename" and no "::type"
8. };

```

- Another advantage of Alias Name is below:

```

1. template<typename T>
2. struct type{ typedef std::vector<T> sometype; }
3.
4. template<typename TT>
5. void someFunction( typename type<TT>::sometype& myArg );
6. // You need typename here, because sometype is depended T.
7. // typename indicates type<TT>::sometype is a type.
8.
9. std::vector<int> a;
10. someFunction(a); // error, cannot deduce 'TT'
11. someFunction<int>(a);
12.
13.
14. template<typename T>
15. using sometype = std::vector<T>;
16.
17. template<typename T>
18. void someFunction(sometype<T> &myArg );
19.
20. std::vector<int> a;
21. someFunction(a);

```

- C++14 offers alias templates for all the C++11 type traits transformations.

```

1. template <class T>
2. using remove_const_t = typename remove_const<T>::type;
3.
4. template <class T>
5. using remove_reference_t = typename remove_reference<T>::type;
6.
7. template <class T>
8. using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;

```

### 14.4.3 scoped enums

- C++98-style enums are now known as unscoped enums.

```

1. enum Color { black, white, red };
2. // black, white, red are in same scope as Color
3. auto white = false; // error! white already
4.
5. enum class Color { black, white, red };
6. // black, white, red are scoped to Color
7. auto white = false; // fine, no other
8.
9. Color c = white; // error!
10. Color c = Color::white; // fine

```

```

11. auto c = Color::white; // also fine (and in accord
12. // with Item 5's advice)

```

2. Enumerators of scoped enums are visible only within the enum. They convert to other types only with a cast.

```

1. enum class Color { black, white, red }; // enum is now scoped
2. Color c = Color::red; // as before, but
3. ... // with scope qualifier
4. if (c < 14.5) { // error! can't compare
5. // Color and double

```

3. Both scoped and unscoped enums support specification of the underlying type. The default underlying type for scoped enums is int. Unscoped enums have no default underlying type.
4. Scoped enums may always be forward-declared. Unscoped enums may be forward-declared only if their declaration specifies an underlying type.
5. Prefer deleted functions to private undefined ones. because Any function may be deleted, including non-member functions and template instantiations.

```

1. template <class charT, class traits = char_traits<charT> >
2. class basic_ios : public ios_base {
3. public:
4.     basic_ios(const basic_ios&) = delete;
5.     basic_ios& operator=(const basic_ios&) = delete;
6. };
7.
8. bool isLucky(int number); // original function
9. bool isLucky(char) = delete; // reject chars
10. bool isLucky(bool) = delete; // reject bools
11.
12. template<typename T>
13. void processPointer(T* ptr);
14.
15. template<>
16. void processPointer<void>(void*) = delete;
// you can't not use void to instantiate

```

#### 14.4.4 noexcept

- When an exception is thrown from an noexcept function, std::terminate gets triggered. compiler will not check it for you.
- noexcept is part of a function's interface, and that means that callers may depend on it.
- noexcept functions are more optimizable than non-noexcept functions.
- noexcept is particularly valuable for the move operations, swap, memory deallocation functions, and destructors.
- Most functions are exception-neutral rather than noexcept.

#### 14.4.5 Variadic Templates

- Finally, there's a way to write functions that take an arbitrary number of arguments in a type-safe way and have all the argument handling logic resolved at compile-time, rather than run-time.

```
1. template<typename T> //Basic condition
2. T adder(T v) {
3.     return v;
4. }
5.
6. template<typename T, typename... Args>
7. T adder(T first, Args... args) {
8.     return first + adder(args...);
9. }
10. /////////////////////////////////
11. template<typename T>
12. show_list(const T& value){cout<<value<<endl;}
13.
14. template<typename T, typename... Args>
15. show_list(const T& value, const Args&... args){
16.     cout<<value<<endl;
17.     show_list(args...);
18. }
```

# Chapter 15

## Style and Guideline

### 15.1 efficiency

- lazy evaluation includes:
  1. reference counting (to avoid extra copy).
  2. distinguish read from write.
  3. lazy fetching.
  4. lazy expression evaluation.
- Detail see in "More Effective C++". Usually, implementation will be more complex with lazy evaluation.

### 15.2 Style

#### 15.2.1 Basic Principles

- **Keep consistent with your style!** Don't change it very often.
- **Don't sweat on the small stuff.** Such as how many spaces to indent? space or tab? Or must have comment etc. In any interview, style is not an important question.
- **Use descriptive function and variable name. Even it's a little longer.** Most code is read a lot more than typed. With good function and variable name, comment is unnecessary.
- **Don't use abbreviation name unless it's very common, such as CPU and TCP/IP.**
- **Don't need to use type indicator for all variable.** Hungarian notation offer no benefit for object-oriented language, especially it's impossible to use in generic programming. But for some generic concept, such as reference, pointer and STL container, you can use it such as: "ptr\_map\_dic". Modern IDE(such as Visual Studio and Understand) support pop up message when you hover your mouse over a variable.

#### 15.2.2 Naming

- For global scope, use g\_ prefix;
- For member variable in class, use tailing underscore. Why?
  1. Prefix underscore uses for reserved word mostly.

2. When you use trailing underscore, you can use auto completion better. For example for variable name `test_`, When you type t, the name will appear, but for `_test`, you have to type \_t two characters, and options are much more than `test_`.

- For member variable in struct, just like an ordinary variable name.
- For constant name, use `k_` prefix and upperCamelCase;
- Using upper-case and underscores for pre-processor Macro;
- **Using upperCamelCase for Classes, Structures, Enumerations, Typedef and Constants;**
- **Using lowerCamelCase and verb for function. Such as `getSth` and `doSth`**
- **Using lowercase and underscore and noun for variable and parameter name. Because for long variable name, It's easier to read. Such as `sth_for_dinner`**
- Using `other` or `rhs` as name for copy ctor and assignment operator.
- The prefix `is` should be used for boolean variables and methods which return bool.

```

1. #define ARRAY_NUM 10
2. bool isVisible;
3.
4. struct Student{
5.     name;
6. };
7. enum BackgroundColor{
8.     Red,    //constant
9.     Green
10. };
11. class Teacher{
12.     name_;   // not m_strName;
13. };
14.
15. typedef struct Student StuStruct ;
16. StuStruct g_global_varaible;
17. const int k_DaysInWeek = 7;
18. main(){
19.     string teacher_name; //meaning variable name
20. }
21.
22. printTeacherName(const string& name) {...} //function name

```

### 15.2.3 Comment and Document

- There are five levels of comment:

| type                 | tool                 | user             |
|----------------------|----------------------|------------------|
| good name convention | no                   | developer        |
| source code comment  | with source code     | developer        |
| API                  | source code+ doxygen | developer + user |
| developer document   | latex                | developer        |
| End user             | Word, power point    | end users        |

- Only use comment when it's very necessary. If you give function and variable good name, don't need to comment them at all.

- source code comment are most inside of a function. API comments are most before function, class. You can use doxygen to produce a html from it.
- Use C++ and doxygen style comment more, use C style comment less.

```

1.  /// brief fooFun does ...
2.  ///
3.  /// if |p flag is true, when happen
4.  ///|param [out] result will be filled
5.  ///|return 0 on success.
6.  bool fooFun(bool flag, int& result);

```

- When you want to comment a large block of code out, use `#if 0 ... #endif`, It's better than using `/* ... */`.
- Don't duplicate the function or class name in comment. Maybe you will change name later, this inconsistency will confuse comment readers in the future.
- Install Doxygen in linux, then run `$ doxygen -g ↴` in the terminal to produce configure file, the name is Doxyfile. In Doxyfile, modify two items:

```
GENERATE_LATEX = NO
INPUT = ./src
```

then run `$ doxygen Doxyfile ↴`, A html directory will be built.

- Install doxygen on Windows, you can run doxywizard application, or you can use GUI to set Doxyfile configuration file.
- Usually you have a standard tree structure of project, that is to say every .cpp has .h file. Under this circumstance, you should put comment command in .h file. Because .h file is an interface to customer.
- If a .cpp file include a .h file, doxygen will not parse the .cpp file automatically, It only parse all .h file in certain directory and extract all types, such as class and struct information. **No function and global variable information is extracted.**
- For class and struct which are declared inside a .h file, Doxygen will show them under Classes tab in the index.html. Even you don't have any comment on it.
- If you want global functions, variables, enums, typedefs, and defines to be documented, you should document the file in which these contents are located using a comment block containing a \file (or @file) command. Even this global function is inside of .h file. With \file in .h file, You'd better to use HIDE\_UNDOC\_MEMBERS to ignore all the other global function members without comment inside this .h file. It will make last result looks clean.
- In order to make Doxygen to parse a .cpp file , you need to put Doxygen command \file in a separate line in the .cpp file, then Doxygen will parse this .cpp file and produce the corresponding html page.
- Given a C++ source code section, Doxygen will produce below html page. **References** and **Referenced** can be turn on in the configure files.

```

1.  /// \brief Return the function this instruction belongs to.
2.  ///
3.  /// Note: it is undefined behavior to call this on an
4.  /// instruction not currently inserted into a function.
5.  const Function *getFunction() const;

```

- Doxygen produce html pages looks like this:

**Module \* getModule()**

**const Function \* getFunction() const**

Return the function this instruction belongs to.

**const Function \* Instruction::getFunction() const**

Return the function this instruction belongs to.

Note: it is undefined behavior to call this on an instruction not currently inserted into a function.

Definition at line 67 of file [Instruction.cpp](#).

References [getParent\(\)](#), and [llvm::BasicBlock::getParent\(\)](#).

Referenced by [llvm::LoopInfo::movementPreservesLCSSAForm\(\)](#).

#### 15.2.4 Code Convention

- Pointer and reference symbol is near to data type, not near to a variable name.
- Declare variables as locally as possible and minimize usage of global variables.
- Avoid macro, use inline function; Avoid magic numbers, instead of using `const` and `enum`
- Format lambdas like blocks of code.

```

1. int cutoff = 7;
2. std::find(foo.begin(), foo.end(), [&](const Foo &a) -> bool{
3.     return a.blah < cutoff //use reference to get local var
4. });

```

- Use `_FILE_` and `_LINE_` to capture the current file name and line number. `assert` just uses this kind of macro inside.
- Use `const` replace `#define` to define global constants. `static const` can be used to define constant class member. (`const` means that it will not change, so I don't need keep multi copy in multi objs, so I use static.)
- Use more `typedef` or `alias` to simplify Complicated Type Expressions, It's helpful especially in STL.

```

1. typedef std::map<int, int> IntMap;
2. typedef IntMap::const_iterator IntMapConstIter;
3. for( IntMapConstIter it = layout.begin();
4.      it != layout.end(); ++it ) {
5.
6.     typedef int(*CB)(int, const char*); //C++11 use function<>
7.     CB callBack; //decare a functions pointer.
8.
9.     int sort(int, const char*){...}
10.    callBack = sort; // & operator is optional here!

```

- Using alias(C++11) is better than typedef.
  1. First word is `using`, There are no sharp symbol before `using`.
  2. Second is TypeName.
  3. Third is assignment `=`.

```

1. using UPtrMapSS =
2. std :: unique_ptr<std :: unordered_map<std :: string , int>>;
3.
4. using CB = int(*)(int , const char* );

```

- Create a zero-valued enumerator to indicate an invalid or default state and make it the first item.
- **Declare all member variables private, then use getter and setter functions to access them.**
  1. Declare all member variables private firstly.
  2. If Outside need to access it's member variables, just build a public interface function. There are two advantages: 1) Inside set-function, You can test valid range. 2) Inside get-function, even type of member variable change, you just need to change get-function, and get all customer code unmodified. In this way, It's de-coupling.
  3. If child want to access its parent member, declare it as protected.
- Don't use multi-thread unless you really need it. Simultaneously respond to many events. Just remember, synchronization has some overhead. Below are some typical scenarios.
  1. A web server.
  2. Interface and working thread.
  3. Take advantage of with multiple processor.

- **Uses early exits to simplify code.**

```

1. if( IsValid){ //bad style
2.   do something;
3. } else{
4.   return;
5. }
6.
7. if(! IsValid) // A better style is
8.   return;
9.   ..do something here .

```

- Turn Predicate Loop into Predicate Function. It makes you code clean and easy to understand.

```

1. for(v. begin ....) { /bad style
2.   if(it ->IsSth){
3.     flag = true; break;
4.   }
5. }
6. if(flag) {...}
7.
8. // A better style is define isHasSth() return bool.
9. if(isHasSth()) {...}

```

- Use `static_assert` to test some compile-time boolean conditions. It supports a message parameter. It doesn't need to build execution application. It will stop when you are compiling your code.

```
1. static_assert(sizeof(int) > 4, "int_is_too_small");
```

- Use assert more in your project. You don't need to include file name and line number in the string. assert will output these hint messages.

```
1. #include<cassert>
2. assert(index>=0 && "index_is_negative");
3. //Assertion failed: expression, file name, line num
```

- If you don't modify container inside loop, don't need to evaluate end() every time through for loop. Please remember below code block. Just remember `auto i,e ++i`

```
1. //g++ -std=c++11
2. vector<int> con={1,2,3}; //list initializer
3. for(auto i = con.begin(), e = con.end(); i!=e; ++i){
4.     .....
5. }
```

- The C++11 standard (23.2.1) mandates that end has O(1) complexity, so previous item has no efficiency meaning in C++11.

- Prefer Preincrement (`++i`), especially in a for loop. When you use `++i` or `i++` inside expression, there are differences. Otherwise, there is no differences.

```
1. for(int i = 0; i<10; ++i)
2. j=++i; // j = i+1;
3. j=i++; //j = i; i=i+1;
4. ++*p; //++(*p) ;
5. *++p; /*(++p);
```

- How to understand `*p++` ?

1. Precedence of prefix `++` and `*` is same. Associativity of both is right to left.
2. Precedence of postfix `++` is higher than both `*` and prefix `++`. Associativity of postfix `++` is left to right.
3. Prefix and postfix are both syntax sugar, in order to reduce typing. But prefix represents **one statement**, postfix means **two statements**;

```
1. ++i; //just like i = i+1 or i+=1;
2. //when i++ used in expression, treated as two statements;
3. *p++; /*(p++); *p ; p=p+1;
4.
5. while(*p != '\0'){
6.     p= p+1;
7. }
8. //can be written to
9. while(*p++ != '\0')
```

- Always turn on all warning options in your compiler with `-Wall`. Once you get warning, Eliminate warning before you go further.

- Sometimes, this warning is what you have to, or if warning comes from a header file you can't change. You can use #pragma warning compile directive to temporary disable it, then restore it later.

```
1. #pragma warning( push )
2. #pragma warning( disable :4512 );
3. #include<not_change.h>
4. #pragma warning( pop ) //restore original warning level
```