Drops of knowledge of C++

Yan Zhao

Contents

Chapter 1

CPP Language

1.1 Basic Knowledge

1.1.1 C and C++

1.1.1.1 Basic conception

- 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 subset of C, so any C programs can be compiled by C++. Compared with C language.
- In C++, there are four sub-topics:
    1. Procedural programming. (traditional C programming)
    2. Object-base programming. (class and object)
    3. Object-orient programming. (inheritance)
    4. Generic programming. (template)
- Duration and scope are two different conceptions. there are three kinds of duration:
    1. automatic
    2. static
    3. 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.
- Only statement, which end with semicolon is executed. Most statements are expression statements. Statement and expression are two important conceptions, you can see their definition in cppreference.com to see academic explanation.

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.

- **All expressions yield a value, So expression is a value, statement is an action.**
- And function call is a expression, because it can yield a value.
- 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 a made into a statement just by tacking a semicolon along the end:

```
i //is expression;
x+y //is expression;
x+y; //is statement
j=i; is a statement.
fun(i) //is expression;
```

1.1.1.2 Combine C and C++

- 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 you 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 should use " extern "C" " in head file or function declaration.
- 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".

```
extern "C"{  // method 2
#include "old_C_header.h"
}

#ifdef __cplusplus  //method 3
extern "C" {
#endif
Foo (int a, int b);
#ifdef __cplusplus
}
#endif
```

- When you use g++, __cplusplus will be defined automatically. (you can't undef it in fact.) When you use gcc, __cplusplus is not defined. At the same time, When you use 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 source file name extension**

- 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 the previous trick again. 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 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.

1.1.2 Style

1.1.2.1 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.

1.1.2.2 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.

```
#define ARRAY_NUM 10
bool isVisible;

struct Student{
  name;
};
enum BackgroundColor{
  Red,   //constant
  Green
};
class Teacher{
  name_;   // not m_strName;
```

```
};

typedef struct Student StuStruct ;
StuStruct g_global_varaible;
const int k_DaysInWeek = 7;
main(){
  string teacher_name;  //meaning variable name
}

printTeacherName(const string& name){...} //function name
```

1.1.2.3 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+ doxgen	developer + u
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.

```
/// \brief fooFun does ...
///
///if \p flag is true, when happen
///\param [out] result will be filled
///\return 0 on success.
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.**

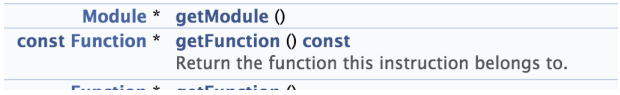
- For class and struct which are declared inside a .h file, Doxygen will show them under Classes tab in the index.html. Even you don't have any comment on it.

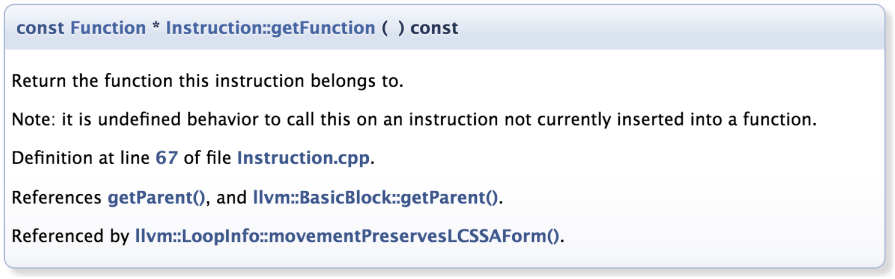
- If you want global functions, variables, enums, typedefs, and defines to be documented, you should document the file in which these contents are located using a comment block containing a \file (or @file) command. Even this global function is inside of .h file. With \file in .h file, You'd better to 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.

```
/// \brief Return the function this instruction belongs to.
///
/// Note: it is undefined behavior to call this on an
/// instruction not currently inserted into a function.
const Function *getFunction() const;
```





1.1.2.4 Code Convention

- Pointer and reference symbol is near to data type, not near to a variable name.

- For forward function, 1) prefer to use reference 2) don't inline f function. Detail can be found in GotW27. When do we need to use forward function.

```
bool f( X& x ){
return g( x );
}
```

- Use `const` as many as possible. Detail can be found in `const` section below.
- 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.

```
int cutoff = 7;
std::find(foo.begin(),foo.end(),[&](const Foo &a ) ->bool{
    return a.blah<cutoff  //use reference to get local var
});
```

- 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.

```
typedef std::map< int, int > IntMap;
typedef IntMap::const_iterator IntMapConstIter;
for( IntMapConstIter it = layout.begin();
                              it != layout.end(); ++it ) {

typedef int(*CB)(int, const char*); //C++11 use function<>
CB callBack;  //decare a functions pointer.

int sort(int, const char*){..}
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 `=`.

```
using UPtrMapSS =
std::unique_ptr<std::unordered_map<std::string, int>>;

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.
  4. Why overload operator « is friend? 1) It can't be member because it need to be written as cout« a; not a«cout. 2) It's need to friend so it can access private member variable. Don't use getter function unless this member variable should be a part of interface. It will break encapsulation

- 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.**

```
if(IsValid){    //bad style
   do something;
} else{
  return;
}

if(!IsValid)  // A better style is
  return;
..do something here.
```

- Turn Predicate Loop into Predicate Function. It makes you code clean and easy to understand.

```
for(v.begin....){    /bad style
  if(it->IsSth){
      flag = true; break;
  }
}
if(flag){...}

// A better style is define isHasSth() return bool.
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.

```
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.

```
#include<cassert>
assert(index>=0 && "index is negative");
//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**

```
//g++ -std=c++11
vector<int> con={1,2,3}; //list initializer
for(auto i = con.begin, e = con.end();i!=e; ++i){
                ...........
}
```

- 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.

```
for(int i = 0;i<10; ++i)
j=++i; // j = i+1;
j=i++; //j = i;  i=i+1;
++*p;  //++(*p) ;
*++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**;

```
++i;  //just like i = i+1 or  i+=1;
//when i++ used in expression, treated as two statements;
*p++; // *(p++);  *p ; p=p+1;

while(*p != '\0'){
p= p+1;
}
//can be written to
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.

```
#pragma warning(push)
#pragma warning(disable:4512);
#include<not_change.h>
#pragma warning(pop) //restore original warning level
```

1.1.3 Declaring and definition

1.1.3.1 Basic conception

- 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.**

```
extern int bar;
extern int g(int, int);
double f(int, double);
// extern can be omitted for function declarations
class Foo;
// 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 calss definition.
```

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

- One very important difference between declarations and definitions is that a symbol may be declared many times, but defined 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**.

- Even you can declare variable, function and class many times, it's not good smell to do that. If you modify it's name, you need to trace back all the declaration. **If a function or class is used by many units, You'd better build a global header file "global.h". and put all the declaration there.**

1.1.3.2 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.

```
class Foo;
Foo* f1 = new Foo //error, you need #include Foo.h
// so compile can know all detail.

fun(Foo* f1){
f1->a;
//error, compiler need to know if there is a
}
fun(Foo f1) //error, compiler need to know the size.
```

- 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 size 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.

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

- About pimpl, you need to know below:

1. In you .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;
Foo* p;
```

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

```
// widget.h file
class Widget { // still in header "widget.h"
private:
struct Impl; // declare implementation struct
std::unique_ptr<Impl> pImpl;
};

//widget.cpp file
#include "Foo.h"  // just include Foo.h in .cpp file.
struct Widget::Impl {
Foo f1;
Gadget g1;
};
```

3. Pimpl Idiom is one of std::unique_ptrs most common use cases. But different with raw pointer, you need to declare or implement some special member functions, such as dtor ctor and move ctor. Explanation can be seen in "effective modern C++ item 22".

```
// widget.h file
class Widget { // still in header "widget.h"

Widget(Widget&& rhs); // declarations
Widget& operator=(Widget&& rhs); // only
Widget::~Widget()
Widget(const Widget& rhs); // declarations
Widget& operator=(const Widget& rhs); // only

private:
struct Impl; // declare implementation struct
std::unique_ptr<Impl> pImpl;
};

//widget.cpp file
#include "Foo.h"  // just include Foo.h in .cpp file.
```

```
Widget::~Widget() = default; // as before
Widget::Widget(Widget&& rhs) = default; // definitions
Widget& Widget::operator=(Widget&& rhs) = default;
Widget::Widget(const Widget& rhs) // copy ctor
: pImpl(std::make_unique<Impl>(*rhs.pImpl))
{}

Widget& Widget::operator=(const Widget& rhs) {
*pImpl = *rhs.pImpl;
return *this;
}
```

1.1.3.3 Translation unit

- When you write an implementation file (.cpp, .cxx, etc), your compiler generates a translation unit. This is the object file from your implementation file plus all the headers you #include in it.

- Each .cpp is a translation unit. Basically, you should put each class 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.

- A translation unit roughly consists of a source file after it has been processed by the C 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.

- 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. But in C++, you have to declare function and variable before you use them. Different with C language, In C language, compiler will guess a function prototype from it's usage, It's not good most of time.

- C/C++ use separate compilation. If you need to use a function or variable in two different files, you should put them in the global.h file and include them when necessary. (add extern keyword before variable name). It will keep declaration just once, **DO NOT copy declaration to the other positions in the other .cpp file. It will lead to many duplication, it's bad smell of code.**

- **In your project, you can global search function or variable declaration, if you find more than two, It's strong indication to make a global head file and put it into it.**

- 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 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, 1) You can know what header file should be included, It's helpful to achieve the goal of demand 2. 2) Sometimes, if you have you 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.

1.1.3.4 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 a.c b.c report no error.
//g++ report error
//in b.c, if you g_i= 2; gcc 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 you 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".

- If an inline function appears in foo.cpp (either because it was written in it, or because it #includes a header in which it was written, in which case the preprocessor basically makes it so). Now you compile foo.cpp, and possibly also some other bar.cpp which also contains an inline function with the same signature (possibly the exact same one; probably due to both #includeing the same header). When the linker links the two object files, it will not be considered a violation of the ODR, as the inline directive made each copy of the file local to its translation unit (the object file created by compiling it, effectively). This is not a suggestion, it is binding.

  It is not coincidental that these two things go together. The most common case is for an inline function to appear in a header #included by several source files, probably because the programmer wanted to request fast inline expansion. This requires the translation-unit locality rule, though, so that linker errors shouldn't arise.

- In one word **Put inline function into the header file.**

1.1.3.5 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 (or object files).

- scope is a property handled by compiler, whereas linkage is a property handled by linker.

- Another property of internal linkage is that it is only implemented when the variable has global scope. static has internal linkage. non-const global variable has external linkage. All constants are by default internally linked. So you can put `const int g_num = 10;` into a global header file, but you can't put a static variable into the global header file.

- **const variables internally link by default** unless otherwise declared as `extern`. It means that:

  1. 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 you .cpp and it will not cause redefine linkage error.
  2. You also can put `const int g_num = 10;` in one .cpp file. then declare `extern const int g_num ;` in global.h file.

3. **For const used just in one .cpp, use static and put it in the .cpp file.**

- When internal linkage, it means that 1) you can put two same static variable name in two different .cpp file, no linkage error. 2) you can't put extern before static, in another word, you can't access static variable in another .cpp file.

1.1.4 malloc and remalloc

1.1.4.1 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 you code will crash. And if you add (int*) cast, it will suppress the warning message.(You kill the only clue, it's bad).

```
 // include "stdlib.h"
// with cast, it will NOT produce a (missing protype)warning.
int *sieve = (int *)malloc(sizeof(int)*length);

// without cast, will produce a warning.
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:

```
#define NULL ( (void*) 0)
int *p = NULL;  //legal
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.

```
#define NULL 0
int *p = NULL //legal
-------------------------------------
f(int i)
f(int * p);

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

- 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().

1.1.4.2 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;

```
void *new_ptr = realloc(ptr, new_size);
if (!new_ptr) {
    // deal with error;
}
ptr = new_ptr
```

1.1.5 new operator

1.1.5.1 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)

```
std::set_new_handler(noMem);
MyClass * p1 = new MyClass;  //1: if fail, call noMem

MyClass * p2 = new (std::nothrow) MyClass; //2: no throw

new (p2) MyClass;  //3: placement new

//4: operator new
MyClass * p3 = (MyClass*) ::operator new (sizeof(MyClass));
      // allocates memory by calling: operator new
      // 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.**

1.1.5.2 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.

```
Point3d *origin = new Point3d;

//C++ pseudo code
if(origin = operator new(sizeof(Point3d))){
        try{
                origin = Point3d::Point3d(origin);
        }
        catch(...){
                operator delete(origin);
                throw
        }
}
```

```
delete origin;

//C++ pseudo code
if (origin != NULL) {
origin->~Foo();
operator delete(origin);
}
```

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

```
void * operator new(std::size_t size) throw(std::bad_alloc){
// your operator new might
using namespace std; // take additional params
```

```
if (size == 0) {    // handle 0-byte requests
size = 1;                 // by treating them as
}                   // 1-byte requests
void *last_alloc;
while (true) {
        *ast_alloc = malloc(size)
        if (last_alloc)
                return last_alloc;

        // allocation was unsuccessful; find out what the
        // current new-handling function is (see below)
        $\Hilight{35}$new_handler globalHandler = set_new_handler(0);
        $\Hilight{30}$set_new_handler(globalHandler);

        if (globalHandler)
                (*globalHandler)();
        else
                throw std::bad_alloc();
}
}
```

```
operator delete (void *ptr){
        if (ptr)
        free(ptr)
}
```

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.

1.1.5.3 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)**

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

1.1.5.4 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

```
        Foo* p;
        // don't catch exceptions thrown by the allocator itself
        // Return a void type pointer
        void* raw = operator new(sizeof(Foo)*100);

        // catch any exceptions thrown by the ctor
```

```
try {
p = new(raw) Foo();  // call the ctor with raw as this
}
catch (...) {
// oops, ctor throw an exception
operator delete(raw);
throw;  // rethrow the ctor's exception
}
```

- Another example to use placement new.

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

p1->~FOO(); // call dtor directly,
p2->~FOO(); //don't call delete p1.

delete[] buffer;
```

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

```
void operator new(size_t, void* p){
return p;
}
```

- 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.

```
FOO* p1 = new(buffer) FOO();
.....
p1->~FOO();
//Don't use delete p1!!!

//A new object in an existing object.
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

```
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 custimzied placement operator delete. Detail can be found in "effective C++ item 52"

```
void* operator new (std::size_t size, void* pMemeory) throw();
void* operator new (std::size_t size, ostream& logStream) throw();
Widget* pw = new(std::cerr) Widget
//When Widget ctor throw exception, it will call
void operator delete(void* pM, ostream& logStream).
//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.

1.1.5.5 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.

```
Foo pa* = new Foo[10];
delete [] pa;
//system will remember the size corresponding with pa,
//with [], it will iterate with size.
//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

```
vec_new(int elem_count, int size, funptr ctor){
        total_size = size*elem_count;
        ptr_array = new char[total_size];
        regist pair of ptr_array elem_count to system
        while(elem<end of address){
                (*ctor)(elem) //call the ctor
                elem+=size;
        }
}
```

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

```
base *bp = new derived[10]
//always use derived *dp = new derived[10];
bp[2] //dangerous, undefine. size is wrong: bp+sizeof(base)*2

delete [] bp //dangerous, 1)size is wrong, 2) it will call base::~base() destructo
```

- 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.

1.1.5.6 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.

```
void* operator new (std::size_t size) throw (std::bad_alloc);
void* operator new (std::size_t size, const std::nothrow_t& nothrow_value) throw()
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*.

```
//global operator new
void* operator new(size_t size){
cout<<"Yan's own operator new" ;
void* mem = malloc(size);
if(mem)
return mem;
else
throw bad_alloc();
}

class Foo{ //class operator new
public:
static void* operator new(size_t size);
// use static
}

void* Foo::operator new(size_t size){
cout<<"Foo's own operator new";
....
}

int *p = new int[100]; //output Yan's own operator new
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)

```
class C {
static void* operator new(size_t, MemoryPool&);
// hides three normal forms
};
//1) plain new
void* operator new(std::size_t);

//2) nothrow new
void* operator new(std::size_t, std::nothrow_t) throw();

//3) inplace new
void* operator new(std::size_t, void*);
```

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

1.1.6 Numerical

1.1.6.1 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 C and C++, you can use limits.h or <limit> to get the all the type limit information.

```
INT_MAX //use in C
INT_MIN
//use in C++
cout<< std::numeric_limits<int>::lowest() << '\t'
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.

```
unsigned int ui = 1;
int i = -2;
// i+ui will stop in one position in the round clock.
```

```
//how to interpret it depends on the programe context;
int j = i+ui; // as int interpret this position CORRECT
(ui+i)<6 //  as unsigned interpret this position. ERROR!
```

- Most unsigned implicit cast error happen when you compare with a constant number.

- Judge it before calculation.

```
if ((x > 0) && (a > INT_MAX - x)) /* `a + x` would overflow */;
// a is point, x>0 clockwise turn, then it will overflow

if ((x < 0) && (a < INT_MIN - x)) /* `a + x` would underflow */;
// a is point, x<0 anti-clockwise turn,  so it will underflow
//It's easy to understand if you draw a clock figure.

if ((x < 0) && (a > INT_MAX + x)) /* `a - x` would overflow */;
if ((x > 0) && (a < INT_MIN + x)) /* `a - x` would underflow */;

if (a > INT_MAX / x) /* `a * x` would overflow */;
if ((a < INT_MIN / x)) /* `a * x` would underflow */;
// need to check for -1 for two's complement machines
if ((a == -1) && (x == INT_MIN)) /* `a * x` overflow */
if ((x == -1) && (a == INT_MIN)) /* `a * x` (or `a / x`) overflow */
```

- Judge it after calculation

```
uint32 a,b;
//assign values
uint32 result = a + b;
if (result < a) {
    //Overflow
}
```

- Build your own template function.

```
template <class T>
void increment_without_wraparound(T& value) {
   if (value < numeric_limits<T>::max())
     value++;
}
```

- 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.

```
unsigned long b, c, c_test;
if (__builtin_umull_overflow(b, c, &c_test)){
    // returned non-zero: there has been an overflow
}
```

- 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.

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

1.1.6.2 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 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.

```
int i, float f;
i=f;
//1) fragment will be lost,  f= 3.99, i will be 3 (not rounding)
//2) If f is too big.  undefined behave

f = i;
// 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.

```
bool isLucky(int number);

isLucky('a') //i = 'a' , promotion
isLucky(false) //i = false , promotion
isLucky(1.2f) //i = f , standard conversion.
```

- 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).

```
isLucky('a') //promotion, NO warning
isLucky(false) //promotion, NO warning
isLucky(1.2f) //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.

```
int x = 66;
char c1 = {x}; //ok

int x = 666;
char c2 = {x};  // not allowed;
```

1.1.6.3 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.

```
char c1, c2, c //c1 and c2 convert to int first.
c = c1+c2;  // then change int result back to char.
/*  LLVM IR code below
  store i8 97, i8* %c1, align 1
  store i8 2, i8* %c2, align 1
  %0 = load i8, i8* %c1, align 1
  %conv = sext i8 %0 to i32
  %1 = load i8, i8* %c2, align 1
  %conv1 = sext i8 %1 to i32
  %add = add nsw i32 %conv, %conv1
  %conv2 = trunc i32 %add to i8
*/

i+f // i will promoted to f and value keep the same.

float f1, f2, f
f = f1+f2 // whether f1 change to double debpends on compiler
//clang++ has fadd in LLVM IR, so it doesn't change f to double.
```

1.1.6.4 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**

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

1.1.7 Type cast in c++

1.1.7.1 Implicit conversion in C++

- 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.
- When will class implicit convert. 1) single ctor, 2) operator Type.

```
class A {
A(int i); // bad
operator const char*(); //bad
};

A a1, a2;
a2 = a1*2;  //implicit conversion 2 to temp A obj
a2 = 2  the same, then call operator =;
```

- **You should always avoid implicit conversion**
  1. use explicit before single parameter ctor.
  2. 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 temperory obj in a1*A(2) expression.

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

- convert class to basic type, you need operator basicTypeName, no argument, no return value and member function.

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

```
class A{
A(class B&);
};
class B{
operator A()
};

void g(const A&);
B b;
g(b)
// it can call A's ctor in class A
// or it can call B's opeartor A() in class B
//compiler will stop, it meet ambiguity.
```

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

- 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.

```
class A {};

class B {
public:
  // conversion from A (assignment):
  B& operator= (const A& x) {return *this;}
};

  void fun(B x){};
  B b;
  b = a;     // calls constructor
```

1.1.7.2 type cast operator

- There are three operators, `dynamic_cast`, `const_cast` and `static_cast`. You should use them more 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++ langauge, we introduce c++ cast operator, see below section.

```
char c = 10;        // 1 byte
int *p = (int*)&c; // compile ok
*p = 5 //runtime error, stack corruption.

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.

```
//In C++, you have to use cast operator.
int* p = static_cast<int*>(malloc(sizeof(*p)));
//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 you 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.

```
#include <stdio.h>
int main(){
    const int a = 12;
    int* p = const_cast<int*>(&a);
    *p = 66;
}
```

- 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.

```
strlen( char* p);
const char* cp = "hello";

strlen(const_cast<char*>(cp));
```

- `dynamic_cast` should only be used down-cast public inherited relationship. Not for private or protected inherited relationship

- 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.

- If a class doesn't have virtual function, you can't use dynamic_cast on this object.

- `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.

```
struct A {};
struct D : public A {};
int main(){
    D d; // the most derived object
    A& a = d; // upcast, dynamic_cast may be used, but unnecessary
    D& new_d = dynamic_cast<D&>(a); // downcast
```

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

```
struct V {
    virtual void f() {};
    // must be polymorphic to use runtime-checked dynamic_cast
};
struct A : virtual V {};
struct B : virtual V {};
struct D : A, B {};

    D d; // the most derived object
    A& a = d; // upcast, dynamic_cast may be used, but unnecessary
    B& new_b = dynamic_cast<B&>(a); // sidecast
```

- `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.

```
struct V {
    virtual void f() {};
    // must be polymorphic to use runtime-checked dynamic_cast
};
struct A : virtual V {};
struct B : virtual V {};
A a;
V& v = a;

B& b = static_cast<B&>(v); //ok
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.**

1.1.7.3 RTTI

- It's a relatively new conception in C++, you should avoid using it on old C++ compiler. It inclueds two methods: `typeid`, `dynamic_cast`.

- **It only work with class hierarchy that has virtual functions.**
- `typeid` operator will return a `type_info` class. You need to include typeinfo.h head file. Typeid operator receive pointer or class name.
- 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.
- Just like exception, It has some loss in performance, you can use flag "-fno-rtti" to turn off it.

1.1.8 cv-qualifier

1.1.8.1 const and mutable

- **Just like assert, Use `const` aggressively.**
- `const` must be initialized when you declare it. `static` will be initialzied 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 only low-level const.**

```
int i = 0;
int * const p1 = &i; // const is top-level
const int *p2 = &ci; // const is low-level

const int *const p3 = p2;
// right most const is top-level, left-most is low-level
const int &r = ci;
// 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)
  3. member 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.

```
// value parameter: top-level const is not part of function signature
int f( int );
int f( const int );     // redeclares f(int): this is the same function

// non-value parameter: top-level const is part of function signature
int g( int& );
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.

```
double cube (const double side){
    return side * side * side;
}
```

- 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:

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

- **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 all.**

```
const int foo() {
   return 3;
}
int x = foo();  // copies happily
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.**

```
const time operator+(const time &t){
   time temp;
   return temp.bla = bla+t.bla;
}

//because + return const, so below just copy
//not use time move ctor, low efficient.
time t3(t1+t2);
```

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

```
class String{
const char& operator[](int position);
}
```

- 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.

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

- When **const** used in member function. 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).

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

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

     ```
     class Fred {
     public:
       void inspect() const;
       // This member promises NOT to change *this
       void mutate();
       // This member function might change *this
     };

     void userCode(Fred& changeable, const Fred& unchangeable){
       changeable.inspect();    // Okay: doesn't change
       changeable.mutate();     // Okay: changes
       unchangeable.inspect(); // Okay: doesn't change
       unchangeable.mutate();   //// ERROR:
     }
     ```

  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.**

     ```
     class Person {
     public:
       const std::string& name_good() const;
       // Right: the caller can't change the Person's name

       std::string& name_evil() const;
       // Wrong: the caller can change the Person's name

       int age() const;
       // Also right: the caller can't change the Person's age
       // ...
     };

     void myCode(const Person& p){
      // const p means not to change the Person object
       p.name_evil() = "Igor";  // But changed....!!
     }
     ```

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.**

```
class Fred { /*...*/ };
class MyFredList {
public:
  const Fred& operator[] (unsigned index) const;
  // Subscript operators often come in pairs

  Fred&  operator[] (unsigned index);
  // Subscript operators often come in pairs
};
```

- **If your member functions doesn't change member varaible, 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 RAM instead of using old value.**

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

- "mutable" allows you to modify a member variable in a class by a const method.

- 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.

```
class TextBook  {
private:
    mutable int length_;
    mutable bool isValid;
public:
    void getLength() const {
    if(isValide == false){
       length_ = strlen(*p);
       isValid = true;
     }
     else return length_
};
```

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

- 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, because if you don't do it. const obj can't call getLength() function at all.

- **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.

```
   iterator begin() noexcept;
const_iterator begin() const noexcept;

const vector<int> cvi;
vector<int>::iterator vi = cvi.begin(); //Compile error
//cvi will call the second overload function anyway.
//but second just return const_iterator.
//and const_iterator can 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**. Because getArea is const funciton, when it is called by a const obj, points will become a const member variable, so error will happen just like previous example.

```
class A{
getArea() const{
   vector<int>::iterator vi = points.begin();
   // This can't be compiled
}

calArea(){
   vector<int>::iterator vi = points.begin();
    // This can't be compiled
}

vector<int> points
};

const A ca;
aa.calArea(); // not compile

A a;
a.calArea(); // ok.
a.getArea(); // only const obj can access const member fun.
```

- 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.**

- 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.

```
class A{
        getArea() const{
        vector<int>::const_iterator vi = points.begin();
        }
        .....
```

- another method is using auto

```
class A{
        getArea() const{
        auto vi = points.begin();
        }
        .....
```

- A good reference article about const is GotW 6.
- 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.

```
iterator insert (iterator position, const value_type& val); //c++98

iterator insert (const_iterator position, const value_type& val); //c++11

vector<int> values:
auto it = std::find(values.cbegin(),values.cend(), 1983); // and cend
values.insert(it, 1998);
```

- 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

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

1.1.8.2 constexpr

- constexpr has two advantages: 1) improve efficiencies(calculate at compile time). 2) expand usage scope (initialze constexpr obj and use in integer constexpr context)

```
    constexpr int fun(int a, int b){return a+b;}

    constexpr int const foo = fun(2,3) ;
    int a[fun(2,3)];
    // without constexpr in function declaration,
    // the next two statmentments can't be compiled.
```

- The first conception is **constant expressions**.

    1. A constant expression is more than merely constant: It can be used in places that require compile-time evaluation, for example, template parameters and array-size specifiers.
    2. 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.

- It can be used on object and function.The basic difference when applied to objects is this:

    1. const declares an object as constant. This implies a guarantee that, once initialized, the value of that object won't change, and the compiler can make use of this fact for optimizations. It also helps prevent the programmer from writing code that modifies objects that were not meant to be modified after initialization.
    2. All constexpr objects are const, but not all const object are constexpr.
    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.

- constexpr can be used on functions.

    1. 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.
    2. const can only be used for non-static member functions, not functions in general. It gives a guarantee that the member function does not modify any of the non-static data members.
    3. 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. **constexpr functions must be able to return compile-time results when called with compile-time values.** The compiler will only accept it if the function meets certain criteria (7.1.5/3,4), most importantly:
        (a) 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.)
        (b) As of C++14 the rules are more relaxed, what is allowed since then inside a constexpr function: 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.

(c) The arguments and the return type must be **literal types** (i.e., generally speaking, very simple types, typically scalars or aggregates)

(d) constexpr function can call only other constexpr function not simple function.

(e) in C++11, constexpr member function is implicit const; C++14 lift it up.

- constexpr and const

1. 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

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

- when do you need to use constexpr function

1. 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.
2. 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.
3. Where compile-time evaluation is not necessary, using inline functions or functions with internal linkage would seem more appropriate that constexpr. All of these variants have in common that the function body is âĂIJpublicâĂİ and is available in the same compilation unit as the call location.
4. a few example;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.

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

5. 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.

```
template< typename Type > constexpr Type max( Type a, Type b ) { return a
```

6. 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:

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

1.1.8.3 static

- In C++, **global and static variables initialized to default values.** But auto variable is random value, 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.

```
class{  //in .h file
  static int obj_num;  // you can't initialize
  const static int months = 12; // you can  initialize
};

//In .cpp file
Int class::obj_num = 0; // no static keyword anymore.
// 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.
- static can be used restrain the scope.

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

- Summary: static uses in three ways:
  1. use it inside a function. unvisible outside of function, valid untill 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,

1.1.9 Namespace

1.1.9.1 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 instead of 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.

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

namespace ns{
int zy;
}
using namespace ns;

int main(){
int zy = 0; //it will hide ns::zy
cin>>zy; //read into local zy
}
```

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

```
#include <iostream>
#include <string>
using std::string;
int main(){
    string str = "Example";
    using std::cout;
    cout << str;
}
```

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

```
//a.h
namespace Yan{
 Class Foo{
     void mem_fun();
 };
}

//a.cpp
//mehtod 1. then use name, using declaration
```

```
using Yan::Foo;
void Foo::mem_fun(){.......}

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

// method 3.
//Good style
void Yan::Foo::mem_fun(){.........}

//method 4.
 //Also good, when you have a lot of function need to be define,
 //compared with method3, save your typing.
namespace Yan{
    void Foo::mem_fun(){......}
}
```

- 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!**

     ```
     #include <iostream>
     using std::cout;
     using std::endl;

     Int main(){
     cout<<"Hello world"<<endl;
     }
     ```

  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

```
// a.cpp file
static int count;

// A better method to use namespace.
namespace{
int count;
}
```

1.1.9.2 Name lookup

- 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.

```
namespace NS{
class T { };
void f(T);
}

NS::T parm;
int main() {
f(parm); // OK, calls NS::f
}
```

- 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.

```
namespace NS { // some header T.h
class T { };
void f( T ); //  number 1, add new function
}
void f( NS::T ); //number 2

int main(){
NS::T parm;
f(parm); // ambiguous: NS::f  or global f?
}
```

- 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.

```
namespace A{
class X { };
void f( X ); // <-- new function
}

namespace B{
void f( A::X );
void g( A::X parm ){
f(parm); // ambiguous: A::f or B::f?
}
}
```

- 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.

```
namespace A{
class X { };
void f( X );
}

class B{ // <-- class, not namespace
void f( A::X );
void g( A::X parm ){
f(parm); // OK: B::f, not ambiguous
```

```
}
};
```

- 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.

```
namespace N{
class X {};

X operator+( const X&, const X& );
}

x3 = x1+x2;
```

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

```
#include <vector>
namespace N {
struct X {};

// this template should not be put in the namespace N
template<typename T>
int* operator+( T , unsigned ) {/* do something */}
}
```

1.1.10 size_t and ptrdiff_t

1.1.10.1 unsigned int

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

```
    unsigned int ui = 0xfffffffe  // -2 or UNIT_MAX-1
    int i = 1;
    //ui+i will be expreseed 0xffffffff in memeory.
    printf("%d", ui+i); //print -1
    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.

```
        unsigned int ui = -2 // or UNIT_MAX-1
        int i = 1;
        ui+i< 6  // greater than 6
        (ui+i)/4  //a bit positive number
        int* p;
```

```
p+(ui+i);  //on 32 bits, this ok,
           //but on 64 bits, ui+i will be promote to 64 bits first.
           // 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 zext 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.

1.1.10.2 size_t

- Type size_t is a stypedef 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. 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.

```
int A = -2;   // should use ptrdiff_t here
unsigned B = 1; // should use ptrdiff_t here.
int array[5] = { 1, 2, 3, 4, 5 };
int *ptr = array + 3;
ptr = ptr + (A + B); //Error
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!

1.1.11 POD

- An aggregate is an array or a class (clause 9) with no user-declared constructors (12.1), no private or protected non-static data members (clause 11), no base classes (clause 10), and no virtual functions (10.3).

- An aggregate can have a not Aggregate1 data member.

```
class NotAggregate1{
  virtual void f() {} //remember? no virtual functions
};

class NotAggregate2{
  int x; //x is private by default and non-static
};

class NotAggregate3{
public:
  NotAggregate3(int) {} //oops, user-defined constructor
};

class Aggregate1{
public:
  NotAggregate1 member1;   //ok, public member
  Aggregate1& operator=(Aggregate1 const & rhs) {/* */} //ok, copy-assignment
private:
  void f() {} // ok, just a private function
};
```

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

```
struct X{
  int i1;
  int i2;
};

struct Y{
  char c;
  X x;
  int i[2];
  float f;
protected:
  static double d;
private:
  void g(){}
};

Y y = {'a', {10, 20}, {20, 30}};
```

- 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:

```
struct Aggregate {
    Aggregate() = default; // asks the compiler to generate the default implement
```

```
};
```

- 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:

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

- 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 :).

- 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.

```
struct POD{
  int x;
  char y;
  void f() {} //no harm if there's a function
  static std::vector<char> v; //static members do not matter
};

struct AggregateButNotPOD1{
  int x;
  ~AggregateButNotPOD1() {} //user-defined destructor
};

struct AggregateButNotPOD2{
  AggregateButNotPOD1 arrOfNonPod[3]; //array of non-POD class
};
```

- 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:1) It supports static initialization, 2) and 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.

- 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.

```
// empty classes are trivial
struct Trivial1 {};

// all special members are implicit
struct Trivial2 {
    int x;
};

struct Trivial3 : Trivial2 { // base class is trivial
    Trivial3() = default; // not a user-provided ctor
    int y;
};

struct Trivial4 {
public:
    int a;
private: // no restrictions on access modifiers
    int b;
};

struct Trivial5 {
    Trivial1 a;
    Trivial2 b;
    Trivial3 c;
    Trivial4 d;
};

struct Trivial6 {
    Trivial2 a[23];
};

struct Trivial7 {
    Trivial6 c;
    void f(); // it's okay to have non-virtual functions
```

```
};

struct Trivial8 {
     int x;
     static NonTrivial1 y; // no restrictions on static members
};

struct Trivial9 {
     Trivial9() = default; // not user-provided
      // a regular constructor is okay because we still have default ctor
     Trivial9(int x) : x(x) {};
     int x;
};

struct NonTrivial1 : Trivial3 {
    virtual void f(); // virtual members make non-trivial ctors
};

struct NonTrivial2 {
    NonTrivial2() : z(42) {} // user-provided ctor
    int z;
};

struct NonTrivial3 {
    NonTrivial3(); // user-provided ctor
    int w;
};
NonTrivial3::NonTrivial3() = default; // defaulted but not on first declaration
                                      // still counts as user-provided
struct NonTrivial5 {
    virtual ~NonTrivial5(); // virtual destructors are not trivial
};
```

- **In C++ 11, POD is not based on aggregate any more, it divid as two different conceptions: 1) trivial class and 2) standard memory layout.**

- 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.

- std has some type trait class definition.

```
cout<< typeid( T ).name()<< "␣"
cout<<std::is_pod < T >::value<< "␣"
cout<<std::  is_trivial < T >::value<< "␣"
cout <<std::is_standard_layout < T >::value<<std::endl;
```

- 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 trivail ctor/copy... but just call malloc() or memcpy()to improve efficiency

```
template <class T> void copy(T* source, T* destination, int n, trivial_false_type)
for (; n > 0; n--,source++,destination++){
//call ctor
}
}
```

```
template <class T> void copy(T* source, T* destination, int n, trivial_true_type){
 memmove(source, destination, n); //much faster here!
}
```

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

1.1.12 Name lookup and overload

- Phases of the function call process:

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

- Name lookup include name invisibility or name hidden(introduced in the below subsection).

- 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, I can't outward search 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!**

- 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)

1.1.12.1 Name hiding

- There are three examples about immediate scope.

  1. class scope:

```
void f1(int i ){...};

class Foo{
void f1(string & str){};
```

```
void f2(void){
int i = 3;
f1(i); //compiler bark here!
}
};
```

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.

```
struct B{
  int f( int );
  int f( double );
  int g( int );
};

struct D : public B{
private:
  int g( std::string, bool );
};

D d;
int i;
d.f(i); // ok, means B::f(int)
d.g(i); // error: g takes 2 args
```

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

```
void f1(int i ){...};

namespace N{
void f1(string & str){};
void f2(void){
int i = 3;
f1(i); //compiler will bark
// you can use ::f1(i) to specify global f1
}
};
```

- 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.

```
class A{
    f(int) {}
};
class B:public class A{
    f(int)  {} // overwrite A::f(int)
};

A* pa = new B();
pa->f(3) // will call base f.
//because pa is A* even it points to B
```

- 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.

```
//example 1
int x =2;{
  int x = 3;
}

//example 2;
void foo(int);
namespace X{
    void foo();
    foo(42); // will not find `::foo`
        // because `X::foo` hides it
}
```

- 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 you local scope. Or use Base:: to specify Base scope name if you have same name in derived class.

```
// method 1;
D d;
int i;
d.f(i); // ok, means B::f(int)
d.B::g(i); // ok, asks for B::g(int)

// method 2:
struct D : public B{
using B::g;
private:
int g( std::string, bool );
};
```

1.1.12.2 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**

```
class A{
    f(int) {}
};

class B:public class A{
    using A::f;
    // add it, you can compile your c++ code now.
    f(char)  {} // overload A::f(int)
};
```

```
B b;
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.

```
class String{
bool operator==(const String& lhs, const String& rhs);
}
String str;
if(str =="hello") //will build a temp obj String("hello");

//You can define overload to avoid implicit type conversion
bool operator==(const String& lhs, const char* rhs);
```

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

```
void f();
void f(int x);
f() or f(10);

void g(int x = 0);
g() or g(10);
```

  If there is a value that you can use for a default, use default parameter.

1.1.12.3 operator overload

- Never overload &&, || and comma in C++. Just remember it!
- 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.
- 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.

```
time operator+(const time &t) const;
//member function.
// t = t1+t2

friend time operator+(int,  const time &t)
nonmember friend function
```

```
// 3+t

time operator+(int i)
member function
// t+3,
//to avoid implicit conversion from 3 to obj.
```

- 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.**

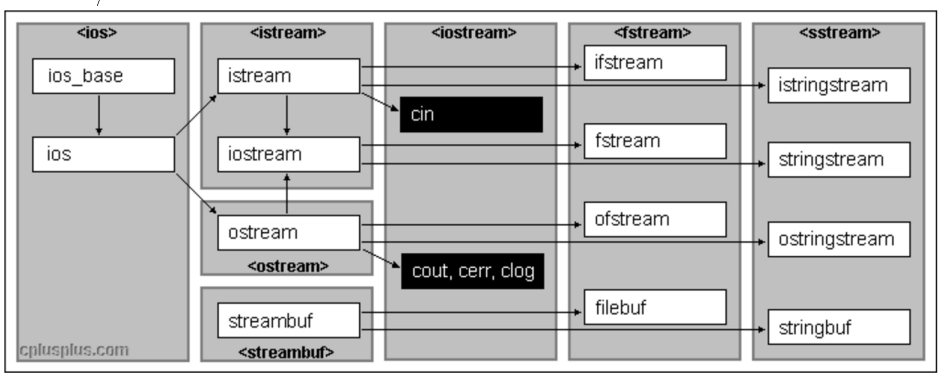
```
class A & operator=(const class A& rhs)
{
    if(this ==&rhs) return *this;

    //Do some things other
    return *this;
}
```

1.1.13 I/O

1.1.13.1 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 includes in your C++ application.
2. three classes: iostream(streambuf), fstream(filebuf), stringstream(stringbuf)
3. clog is just like cerr, but it buffer it 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 string, C++ will automatically, convert it, 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.

1.1.13.2 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	Character (including white-character)	string(line)
C	scanf("%d %f %c %s",&i, &f,&c);	int a = getchar();	fgets(stdin, char*p, n)

C++	cin»i»f»c»w;	cin.get(char & c);
		Ch = cin.get();
			cin.get(char *p, n);
			cin.getline(char *p, n);
			getline(cin, string);

- 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 scanf skip any whitespace, you can use space before %c.

```
while(true){
   scanf("%c",&c);
   //scanf(" %c",&c);
   printf("you input: %d" c);
}
// When you input a(enter)
//output will be like
you input: 97(a value)
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

```
while(true){
   cin>>c;
   cout<<"you input:"<<c<<endl;
}
// When you input "    a(enter)"
//output will be like
you input:  a
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:

```
//read a word until reach white character.
Scanf(%s,char_array) //c
Cin>>char_array or ; //c++
Cin>> str;

//line
gets(char_array)  //c
fgets(char_array, n , FILE *) //recommend to use this for safety.

cin.getline(char * ,int n) //c++ read and discard newline
cin.get(char * ,int n) //not read newline
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()

```
while(cin.get(c))
// use cin.get(char) in reading loop

cin.get() != '\n'
//use cin.get() return character to test sth.
```

```
cin.get()!= EOF
//When used in EOF, you have to use int
//because EOF may not be expressed by char tyep
```

- Three confused functions: cin.get and cin.getline are almost the same thing.

```
cin.get ( char* s, streamsize n, char delim );
cin.getline (char* s, streamsize n, char delim );
istream& getline (istream&  is, string& str);
```

1. For each line, if you don't know the max length, juse use getline(cin, string). You don't need to input any length.(you can reserve lengh 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();

1.1.13.3 custom 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.

```
char input[80];
int i = 0;
while( (ch=cin.peek()) != '.' && ch != '\n')
    cin.get(input[i++];)
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.

1.1.13.4 Input error

- **For the error of input: Just remember C language return EOF and NULL, and C++ return all false.**

	Number, non-white	character, white	string
C	scanf return EOF	getchar return EOF	fgets return NULL
c++	cin return false	cin return false	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 recoveried 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 . 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.

```
std::ifstream is;
 is.open ("test.txt");
 if ( (is.rdstate() & std::ifstream::failbit ) != 0 )
    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.

```
char ch, str[20];
cin.getline(str, 5);
cout<<"flag1:"<<cin.good()<<endl;     // check good bit
cin.clear();    //without, cin>>ch below will not work at all
cout<<"flag1:"<<cin.good()<<endl;     // check good bit again
cin>>ch;
cout<<"str:"<<str<< " ch:" <<ch<<endl;

input:
12345[Enter]
output:
flag1:0 // good()return false
flag2:1 // good()return true after clear().
str:1234 ch:5
```

1.1.13.5 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.

```
while(!cin.fail()){   // that is bad programming style
    cin>>i;  //here may make stream fail().
    .....       //then i is not valid value
}
```

- If you just want to input, You don't want to know eof or deal with failbit. You can use below:

```
while(scanf("%d",&i) != 1)
while((int c = getchar())!=EOF)
while( fgets(line, sizeof(line), fp) != NULL )
//in c language, use these to exit loop!

while(cin>>input)
while(cin.get(p, 20) )
while(getline(ifstream, string)) {
//in C++ language, use bool operator to exit loop.
  //do some useful things.  //input is right.
}
```

- If you want to know eof or deal with error. You can use below:

```
While(true)  // use break to exit loop;
{
  cin>>i // or getline(ifstream, string);
  If(cin.eof()){    //If it's EOF
     cout<<"EOF encountered"<<endl
     break;
  }
  If(cin.fail())           //Invalide input
  {
     cin.clear();     //Important. clear state and buffer
     while(cin.get()!='\n')  //get rid rest of line,
               continue ;
       cout<<"please input a number"<<endl;
       continue;
  }
  ...  //do some useful things.  //input is right.
}
```

- 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

```
cin.clear();     //Important. clear state and buffer
while(cin.get()!='\n')
      continue ;       // method 1

while(!issapce(cin.get()))
            continue;  //method 2

basic_istream& ignore(streamsize _Count = 1, \
int_type _Delim = traits_type::eof());  //method 3
cin.ignore(5, 'a');
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

- You can use istream_iterator, It can save you some trouble to judge EOF.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
class A{
private:
int x;
int y;
friend istream& operator>>(istream& in, A&);
friend ostream& operator<<(ostream& in, const A&);
};

istream& operator>>(istream& in, A& a){
in>>a.x>>a.y;
return in;
}
```

```
ostream& operator<<(ostream& out, const A& a){
out<<a.x<< " " <<a.y;
return out;
}

int main(){
vector<A> v;
copy(istream_iterator<A>(cin), istream_iterator<A>(),
                                                back_inserter(v));
copy(v.begin(), v.end(),ostream_iterator<A>(cout, "\n"));
}
```

- **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;**

1.1.13.6 Output

- For cout, It can recognize type automatically, and It is extensible, You can redefine << operator so that cout can recognize you data type.

```
class Foo{
friend ostream & operator<<(ostream& s, const Foo &r);
}

ostream & operator<<(ostream& s, const Foo &r){
s<<Foo.a<<Foo.b<<endl;
}
```

- How to print pointer address?

```
char* amount = "dozen";
cout<< amount  ; //print "dozen" string

cout<<(void*) amount;   //prints the address of pointer
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 a 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.

1.1.13.7 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.

```
std::fstream fs;
fs.open ("test.txt", ios_base::in| ios_base::binary);
if(!ifs.is_open())
      exit(1);
char c = ifs.get();
// use all previous input methods
ifs.close();
```

- Only read ifsteam; Only write ofsteam; write and read fstream.
- For ios_base::binary mode, use write() and read() funciton.
- 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.

1.1.13.8 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 bettween 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.**

```
stringstream outstr;
//change number to a text
outstr<<"salary value"<<123333.00<<endl;
String str = outstr.str() //change to string

//change text to number
Istringstream Instr(str);
While(instr>>number)
cout<<number<<endl
////////below is C language/////////
char sentence []="Yan is 41 years old";
char str [20];
int i;
sscanf (sentence,"%s %*s %d", str, &i);
//* means input will be ignored. So str = Yan, i = 41.
sprintf(.....);
```

1.1.14 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.

1.2 initialization

1.2.1 basic

- **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 name: 1)initilizer list, 2) brace(list) initilization, 3) member initilization list(mem-init)

  1. member initilization 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. std::initializer_list is a **new data type. just like std::list.**

1.2.2 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

```
int g0;  //zero initialization
int g1 = 42;    //  static initialization
extern int f();
int g2 = f();   //  dynamic initialization
```

```
int a = f(); // a exists just to call f
int x = 22;
int f() {
++x;
return 123; // unimportant arbitrary number
}
// x equals to 23, not 22
// bcause static initialization
// 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:

```
#include "x.h" // File x.cpp
X x;
//x maybe initialize before y or after y

#include "y.h" // File y.cpp
extern X x;
Y y;
Y::Y(){ //here x maybe not be constructed
x.goBowling();
}
```

- How to resolve: build a singleton class

```
#include "x.h"  // File x.cpp
X& getX(){
static X x;
return x;
//another implementation.
//static X* px = new X();
//return *px;
}

#include "y.h" // File y.cpp
Y y;
Y::Y(){
getX().goBowling();
}
```

1.2.3 Init method

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

1.2.3.1 Six init methods

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

```
T t;
new T;  //default

T t{};
T(); T{};
new T(); new T{};
: member(), member{}      // class member initializer lists (value Init)
```

```
T object(arg, ... );
T(arg1, arg2, ... );
new T(args, ...)
: member(args, ...)          // class member initializer lists
T(other)                     // function-style cast
static_cast<T>(other)        // explicit static_cast
[arg](){...}                 // lambda closure arguments captured by value
//lambda is object, so for this object, lambda_1 obj(arg), is a direct in

T object = other;            // Initialization via assignment
T array[N] = {other};        // In array-initialization, the individual
// values are copy-initialized
f(other)                     // Pass-by-value
return other;                // Return-by-value
catch (T object)             // Catch-by-value
throw object;
```

- for one word **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.**

- for list init, there are value list init, direct list init and copy list init

```
T object{};
T{};
new T{}
Class { T member{}; };
: member{}                          // Class member initializer lists

T object{arg, ...};
T {arg, ...};
new T{arg, ...}
Class { T member{arg, ...}; };      // Class member default initializer
: member{arg, ...}                  // Class member initializer lists

T object = {arg, ...};
object = {arg, ...};
Class { T member = {arg, ...}; };   // Class member default initializer
function({arg, ...});               // Initializes temporary for the functi
return {arg, ...};                  // Initializes temporary for return va
```

- Aggregate init Aggregate initialization is a special case of list initialization. It is used when initializing arrays or simple structs (all-members-public, no user-provided c'tors

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

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

```
T & ref = object ;
T & ref = { arg1, arg2, ... };
```

```
T & ref ( object ) ;
T & ref { arg1, arg2, ... } ;
```

1.2.3.2 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.

```
class exec{
public:
exec() = default;
int i;
};

class exec2
{
public:
exec2();
int i;
};
exec2::exec2() = default;

const exec e;       //erro, exec has default ctor
const exec2 e2;     //right, exec2 has user-define ctor
exec e    //default init, so e.i is random
exec e{}  //value init, but there is no user-define ctor, so e.i == 0

exec2 e  // default init, so e.i is random
exec2 e{}  //value init, but there is user-define ctor, so e.i is random
```

- There are difference when you use new with parenthesizes. Detail can be found: "Do the parentheses after the type name make a difference with new?"

```
struct A { int m; }; // POD
struct B { ~B(); int m; }; // non-POD, compiler generated default ctor
struct C { C() : m() {}; ~C(); int m; }; // non-POD, default-initialising
```

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.

1.2.3.3 copy init and direct init

- difference between copy init and 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 therotical definition. When optimization kick in, it hide the deep therotical definition. ct2 call single argument ctor directly. **copy init is difference with copy ctor.**

- An example and analysis:

```
ClassTest ct1("ab");//direct init
ClassTest ct2 = "ab";//copy init
ClassTest ct3 = ct1;//copy init
ClassTest ct4(ct1);//direct init
ClassTest ct5 = ClassTest();//copy init Class has default ctor
ClassTest ct6 = {1,2} //copy init Class has two parameter ctor.
```

1. ct2 is copy init. compile analyze the syntax according therotical 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. it think 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 therotic 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 therotic) whether it has two steps(has equal sign).

- When copy ctor is explicit

```
class A{
public:
        int i;
        A(int a = 0);
        explicit A(const A &a){} //EXPLICIT copy constructor
};

void funcX(A a) {
        //ERROR to take A by value (implicit copying)
}

A funcY(){
        A a;
        return a; //ERROR - function returning A by value (implicit copying)
}


A a1 = a;////ERROR  implicit copying of TestOverload not allowed
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 explict copy ctor, A a = 1 still work.But when the copy ctor is private, A a = 1 fail**

- When single parameter ctor is explicit

```
class A{
public:
int i;
explicit  A(int a = 0);
A(const A &a){}
};

A a = 1; // fail.
//A a = {1}  //fail
A a = A{1}  //work
A a = A(1)  //work
```

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

- Another more interesting example

```
class A{
public:
int i;
explicit A(int a = 0):i(a){cout<<"one";}
A(int a , int b ):i(a){cout<<"two";}
A(const A &T){ cout<<"copy ctor";}
//TestOverload(const TestOverload &T) = delete;
};

A too = (1,2); //fail, it (1,2) became 2, then call single ctor
A too = {1,2}; //work, call two parameter ctor
```

1. () and comma is operator, it's different with {}

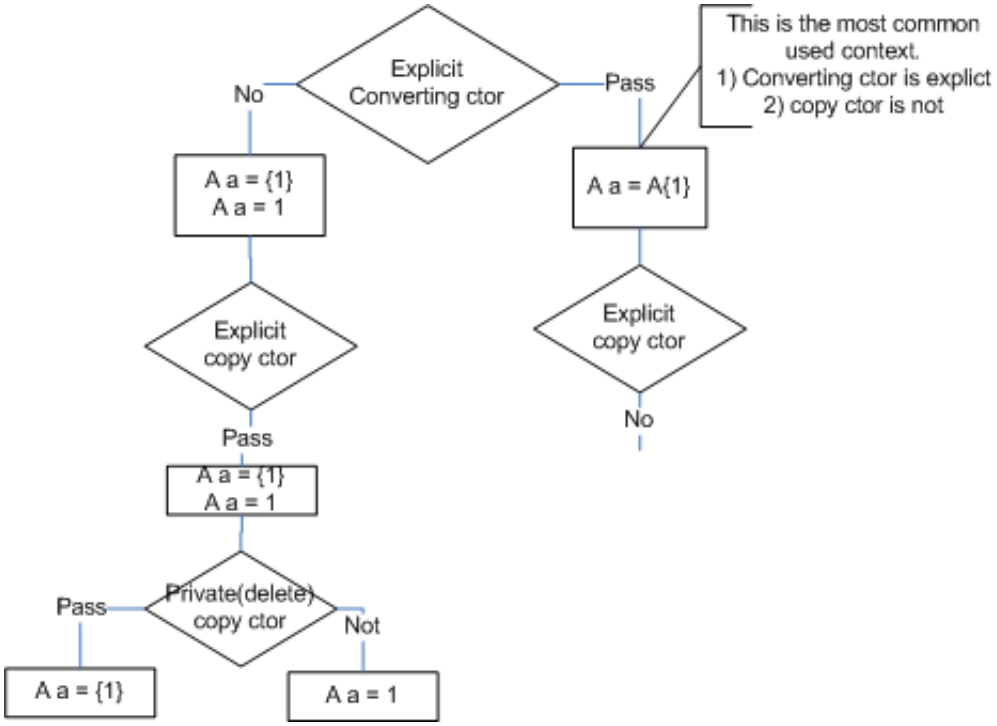
- For assignment, you can use braced init and avoid type name. But when you define explicit converting ctor, it will not work.

```
pair<int, string> p;
p = {1,"aaa"}
A a;
a = {1} //without explicit converting ctor
a = A{1} //with explicit converting ctor
```

- summary:

1. The first step is to see the right side of =, if converting ctor is expilict, 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 initlizer_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
A a = (i), A a={i}	explicit converting ctor will not work
A a = A(i), A a = A{i}	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.

```
Cents add(const Cents &c1, const Cents &c2){
        return Cents(c1.getCents() + c2.getCents());
        // return anonymous Cents value
}

cout <<add(Cents{6}, Cents{8}).getCents();
```

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

1.2.3.4 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.

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

```
int x;
int (x); //same as above
int f(int x);
int f(int (x)); //same as above
```

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

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

5. For object, thing becomes interesting.

```
int (x); // just like int x;

class A{
A(int a);
...}

A(a);  // You want to create a temporary obj
//but compile think it as A a;
int a = 10;
A(a)  //error, redefine a, because if it's decaration, it is
const A& ar = A(a); //this is OK

(A(a));
//If you really want a temporary A object,
//use another parentheses.
```

6. For function, vexing problem happen below.

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

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

```
struct A{
A (B const& b){};
void doSomething(){};
};
```

```
int main(){
A a(); // declare a function "a", return A

A a(B(x)); //declare a function "a",
//type of parameter x is B

A a(B()); //declare a function "a",
//receive function pointer, return B, void input
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.

```
list<int> data(istream_iterator<int>(dataFile),
istream_iterator<int>());
// data is function. first parameter is dataFile, type istream_iterator<in
// second parameter is function pointer.
// The First, you can add () around parameter name; the second, you can o
```

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

```
A a{B{x}};
A a{B{}};
```

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

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

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

```
class float3{
......
};

float3 x(); //x is function, not call default ctor

float3 x(float3());  // a function
float3 x(float3(i));  // still a function, float3(i) equal float3 i;
//because () can be omitted by a compiler.
//vs.
float3 y{float3{}};  // a variable
```

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

1.2.4 Brace Init

1.2.4.1 Brace syntax

- The basic syntax examples.

```
int x1 = 27;  //1) asiignment
int x2(27); //2) parentheses before C++11

int x3{ 27 }; //3) brace init, introudced in C++11
int x4={ 27 }; //same as 3, but for class type, a trivial difference. can
```

- More detail syntax analysis.

```
widget w;                    // (a)
widget w();                  // (b)
widget w{};                  // (c)

widget w(x);                 // (d)
widget w{x};                 // (e)

widget w = x;                // (f)
widget w = {x};              // (g)

auto w = x;                  // (h)
auto w = widget{x};          // (i)
```

1. (b) is function declaration. That is why we prefer use brace init, it will avoid this vex problem.
2. (d) and (e) Assuming x is not the name of a type, these are both direct initialization. ThatâĂŹs because the variable w is initialized âĂIJdirectlyâĂİ from the value of x by calling widget::widget(x). If x is also of type widget, this invokes the copy constructor. Otherwise, it invokes a ds.
3. (e) is better than (d) to avoid narrowing, for example:

```
Class A{
A(int){}
}
A a(1.2); //work
A a{1.2}; //fail
```

4. for (e), it prefer constructor that takes an initializer_list
5. For (f), when x is type of widget, just like (d), but can't work when copy ctor is explicit(introudced before)
6. 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 âĂIJthe slow way to moveâĂİ 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 âĂIJconceptuallyâĂİ a few times above. ThatâĂŹs 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 calledâĂŤthe copy constructorâĂŹs side effects may or may not happen, thatâĂŹs all.

7. For (g), just like (f)
8. (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)
9. (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 calledâĂŤsimilarly to what we saw with (f) and (g),

```
                class TestOverload{
                public:
                int i;
                TestOverload(int a = 0):i(a){}
                TestOverload(const TestOverload &T){ cout<<"copy ctor";}
                };

                TestOverload too = 2;  //test for (f) (g)
                TestOverload too = {2}   //
                1) it will work, no call copy ctor(no any output)
                2) If ctor has explicit --fail. TestOverload too{2} --succeed
                3) if copy ctor is private --fail.(strange, even I don't call
                ----------------
                ----------------TestOverload too;  //test for (h)
                ----------------auto w = too
                ----------------1) When if copy ctor has explict --fail.
                ----------------
                ----------------auto w = TestOverload{2} //test for (i)
                ----------------1) you can use auto w = TestOverload(2), it just call ctor, a
                ----------------2) {} is better, it more generic and avoid narrow auto w = Te
                ----------------3) even ctor is explict, it can work too.
                ----------------4) don't call copy ctor at all
```

```
                class A{
                public:
                A(){};
                explicit A(int k):m_a(k){};
                explicit A(const A& rhs){m_a = rhs.m_a;};
                virtual ~A(){};
                int m_a;
                };

                int main(){
                A a={110};// will compile error
                A a{110} //this will work here.
                A b = {a}; // will compile error.
                //because there are explicit before copy ctor.
                }
```

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

- 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.

1.2.4.2 Brace init advantage

- **zero init single or aggreaged varaible** 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:

```
        //C++11: default initialization using {}
        int *p{}; //initialized to nullptr
        char s[12]{}; //all 12 characters are initialized to '\0'
        char *p=new char [5]{}; // all five chars are initialized to '\0'

        int i{};                  // i becomes 0, can't use () here
        std::string s{};          // s becomes "" can't use () here
        std::vector<float> v{}; // v becomes an empty vector
        double d{};               // d becomes 0.0, can't use () here
```

- **class aggregated member**

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

- **direct init aggreaged varaible**

```
        int intArray[]= {1,2,3,4,5};
        std::vector<int> intArray1{1,2,3,4,5};
        std::map<std::string,int> myMap{{"Scott",1976}, {"Dijkstra",1972}};

        // Initialization of a const heap array
        const float* pData= new const float[3]{1.1,2.2,3.3};
```

- **init a class event without ctor**

```
        class MyClass{
        public:
        int x;
        double y;
        };

        class MyClass2{
```

```
public:
MyClass2(int fir, double sec):x{fir},y{sec} {};
private:
int x;
double y;
};

// Initializations of an arbitrary object using public attributes
MyClass myClass{2011,3.14};
MyClass myClass1= {2011,3.14};

// Initializations of an arbitrary object using the constructor
MyClass2 myClass2{2011,3.14};
MyClass2 myClass3= {2011,3.14};
```

- textbfomit type name

```
MyClass2 fun (MyClass2 m)

fun ( {2011,3.14} ){
return {2011,3.14};  //pay attention, there is ; in the end
}

fun ( MyClass2(2011,3.14) ) // fun ( (2011,3.14) ) doesn't work

MyClass2 mc2;
mc2 = {2011,3.14}  // or MyClass2(2011, 3.14)
```

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

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

// C++98
rectangle        w( origin(), extents() );       // oops, vexing parse
vector<int>      v;                              // urk, need more code
for( int i = 1; i <= 4; ++i ) v.push_back(i);    //   to initialize this

// C++11 (note: "=" is mostly optional)
rectangle        w   = { origin(), extents() };
vector<int>      v   = { 1, 2, 3, 4 };
```

- **Also work in template**And note that this isnâĂŹt just an aesthetic issue. Consider writing generic code that should be able to initialize any typeâĂę and while weâĂŹre at it, letâĂŹs gratuitously use perfect forwarding as an example:

```
template<typename T, typename ...Args>
void forwarder( Args&&... args ) {
// ...
T local = { std::forward<Args>(args)... };
// ...
}

forwarder<int>              ( 42 );                    // ok
forwarder<rectangle>        ( origin(), extents() ); // ok
forwarder<complex<double>>( 2.71828, 3.14159 );     // ok
```

```
forwarder<mystruct>      ( 1, 2 );          // ok because of {}
forwarder<int[]>         ( 1, 2, 3, 4 );    // ok because of {}
forwarder<vector<int>>   ( 1, 2, 3, 4 );    // ok because of {}
```

- A good reference article is GotW #1 Solution: Variable Initialization âĂŞ or Is It?

1.2.5 initilizaer_list

1.2.5.1 why initilizaer_list

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

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

- In the generic programming, we need to use the "uniform initilizaer" make the vt and array use the same syntatic 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

```
vector( std::initializer_list<T> init)
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 funciton.

```
void f(const initializer_list<string> &slst){
....
}

f({ "Good", "morning", "!" });
```

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

```
// A braced-init-list can be implicitly converted to a return type.
vector<int> test_function() { return {1, 2, 3}; }

// Iterate over a braced-init-list.
for (int i : {-1, -2, -3}) {}

// Call a function using a braced-init-list.
void TestFunction2(vector<int> v) {}
TestFunction2({1, 2, 3});
```

1.2.5.2 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 use braces.3) the number of parameter inside of braces is changeable. 4) the type should be the same.

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

class A{
public:
int i;
int j;
};

A a = {1, 2} //It's brace init, not initializer_list
             //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 'âĂIJAlexâĂİ, 522' in fact means 'std::pair<std::string const, int>âĂIJAlexâĂİ, 522'.

```
std::map<std::string, int> scores{
{"Alex", 522}, {"Pumu", 423}, {"Kitten", 956}
};
```

- **Am empty pair of braces can be 1)default initialization, 2) a empty std::initializaer_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:

```
        class Widget{
public:
int i;
Widget(){cout<<"default";}
Widget(initializer_list<int>){cout<<"init";}
};

Widget w1; // calls default ctor
Widget w2{}; // also calls default ctor, even  you have
             //initializer_list ctor
Widget w3(); // most vexing parse! declares a function!

Widget w4({}); // calls std::initializer_list ctor with empty list
Widget w5{{}}; // ditto
```

1.2.5.3 problem of initializer_list

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

```
class Widget{
public:
int i;
Widget(){cout<<"default";}
Widget(initializer_list<int>){cout<<"init";}
};

Widget w2{};
```

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

```
class Widget {
public:
Widget(int i, bool b); // as before
Widget(int i, double d); // as before
Widget(std::initializer_list<long double> il); // added
};

Widget w1(10, true); //  calls first ctor
Widget w1{10, true}; // std::initializer_list ctor
// (10 and true convert to long double)

Widget w2(10, 5.0); //  calls second ctor
Widget w2{10, 5.0}; // std::initializer_list ctor
// (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.**

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

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

```
std::vector<int> v1(10, 20); // use non-std::initializer_list
// ctor: create 10-element vector, all elements 20.

std::vector<int> v2{10, 20}; // use initializer_list ctor:
// 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.)

1.2.6 summary

1.2.6.1 Example demo

- Some syntactic examples:

```
class A{
        public:
        A(){};
        A(int k):m_a(k){};
        A(const A& rhs){m_a = rhs.m_a;};
        int m_a;
};

void fun(const A &a){}
int i = 3;
```

- Default ctor example

Expression	meaning
A a, A a {}	default ctor
A a()	declare function a,vexing problem
A()	A temporary A obj
B b(A())	declare function, vexing problem
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}	ctor
A (i)	**just like A i, you have define i, so compile error.**
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. **We need value for functoin parameter, not statement**

- **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.

1.2.6.2 usage direction

- **familier with Six Init methods.**

- **six methods and brace are irrelenvant. brace can be use in all six method and bring its own advantage.**

- **use textttauto var = .. style if possible, then use brace if possible, if there is initializer_list, considering use parentheses.**

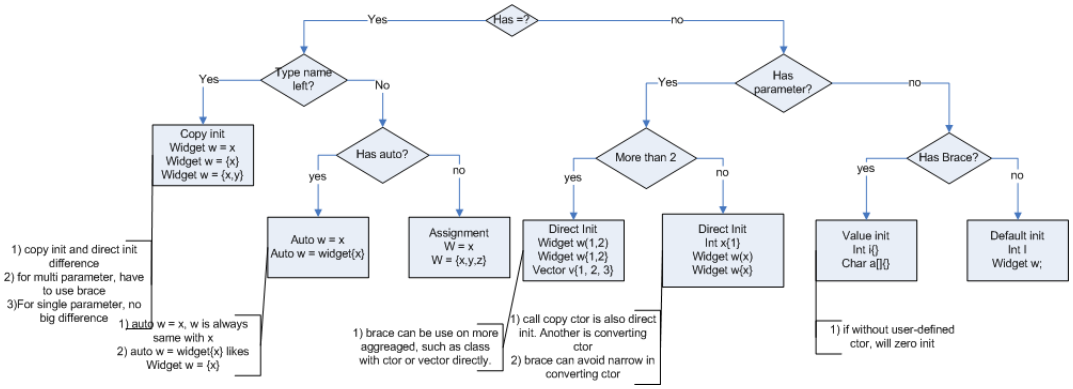
- When expr appear, prefer to use auto. expr can be 1) math express, auto a = b+c, 2) function call, auto a = v.begin() 3) new to avoid type name 4) left hand type is same with right type auto a = b (you can write Type a(b), but don't need write down the type name) . 5) lambda 6) template

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

```
//const char* s = "Hello";
auto s = "Hello";

//widget w = get_widget();
auto w = get_widget();

widget* w = new widget{};                    /* auto w = new widget{}; */
unique_ptr<widget> w                         auto w = make_unique<widget>();
= make_unique<widget>();
```

- Use braced initialization if possible, because 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.

- When use = before braced initialization, it is called copy init. Detail can be seen in the previous section.



- Pay attention two traps. 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 initilizaer_list type.)

- auto has overlap with brace init, Anytime you can use ctor directly, use ctor. just style fight, I chose ctor

```
        class A{
        A(int i, int j)
```

```
}

A a{1,2};  // I prefer to use this way.
auto a = A{1,2}
```

1.3 auto

About the auto type deduction, please refer to the next chapter Generic programming type inference section.

1.3.1 auto declaration

1.3.1.1 advantage

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

```
auto x = {5}; //x is a std::initializer_list<double>
auto p = &x  //p is int* type

auto p = new vector<pair<int, string> >;
std::vector<std::pair<int, std::string>> array;
auto it = array.begin(); // they are more useful.

double fm(double, int);
auto pf = fm // pf is double(*)(double,int)

for (const auto& element : myarray) {
//do stuff that reads from element
}
```

- 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

```
const std::map<int, Module>::iterator iter = modmap.find(123);
const auto iter = modmap.find(123);

Module& mod = vec[17];
auto& mod = vec[17];

auto p = std::make_shared<Module>();
auto p = std::shared_ptr<Module>(new Module); // not safe!
```

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

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

- three advantages:

  1. avoidance of uninitialized variables

2. verbose variable declarations

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

template<typename It>
void dwim(It b, It e) {
typename std::iterator_traits<It>::value_type currValue = *b;

void dwim(It b, It e) { // better with auto
auto currValue = *b;
```

3. avoid what I call problems related to "type shortcuts."

```
std::vector<int> v;
...
unsigned sz = v.size();
auto sz = v.size();

std::unordered_map<std::string, int> m;
for (const std::pair<std::string, int>& p : m){
... // do something with p
}

for (const auto& p : m){
... // as before
}
```

1.3.1.2 pitfall of auto

- two pitfalls 1) "invisible" proxy classes 2) init brace return init_list. An invisible proxy class example is vector<bool> [] operator.
- auto can also be used as return type. Below will produce compile error, because it has ambiguity.

```
auto f(int i){  // It will cause compile error.
if ( i < 0 )
return -1;
else
return 2.0
}
```

- return deducing

```
struct record {
std::string name;
int id;
//GUID id; //change id type, below doesn't need change
};

auto find_id(const std::vector<record> &people,
const std::string &name){...}

auto ii = find_if(people.begin(), people.end(), match_name );
if (ii != people.end()){...}
```

1. find_id return auto type. It means that it can be used return type of function.(C++14),
2. `auto ii` is another usage of auto, use auto as declaration. It's helpful to reduce complexity of complex type, such as iterator.

1.3.2 auto in function

1.3.2.1 auto in return

- auto can be used in function parameter, it should be a short hand of template function.

```
void fun1(auto i)
{
        cout<<i<<endl;
}

        fun1(23);
        fun1("abc");
```

- auto can be used in common function return.

```
auto fun1(){
        return 12;
}
```

1.3.2.2 atuo in lambda and template

-

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

```
        auto derefUPLess =                  // comparison func.
        [](const std::unique_ptr<Widget>& p1, // for Widgets
        const std::unique_ptr<Widget>& p2) // pointed to by
        { return *p1 < *p2; };                  // std::unique_ptrs

        //C++14 version
        auto derefLess = // C++14 comparison
        [](const auto& p1,  const auto& p2) // values pointed
        { return *p1 < *p2; }; // to by anything pointer-like
```

- generic lambda(just like template lambda)

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

- another example auto&& comes from generic lambda,

```
        auto timeFuncInvocation = [](auto&& func, auto&&... params) {
        // start timer;
        std::forward<decltype(func)>(func)(
        std::forward<decltype(params)>(params)...
```

```
        );
        // stop timer and record elapsed time;
        };

        struct Functor
        {
        void operator ()() const &  { std::cout << "lvalue functor\n"; }
        void operator ()() const && { std::cout << "rvalue functor\n"; }
        };


        int main()
        {

        auto perfectLambda = [](auto&& func, auto&&... params) {
        std::forward<decltype(func)>(func)(
        std::forward<decltype(params)>(params)...
        );
        };

        auto lambda = [](auto func, auto&&... params) {
        func(std::forward<decltype(params)>(params)...);
        };

        Functor fun;

        lambda(fun);
        lambda(Functor{});

        perfectLambda(fun);
        perfectLambda(Functor{});
        }
```

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

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

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

  2. C++ 14, directly use auto.

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

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

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

1.4 pointer and smart pointer

1.4.1 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.

```
void failNew(){
cerr<<"Fail now"<<endl
abort();
}
-----------------------------------------------
extern void ( *set_new_handler ( void (*)() ) ) ();
// This function declaration is very complex.
set_new_handler(failNew);
```

- A better method is to use typedef method.

```
typedef void(* FunPtr)();

FunPtr (*set_new_handler) (FunPtr);

set_new_handler(failNew);
```

- in C++ 11, When you want a function pointer, encourage you to use more lambda idiom. You also can use `auto` to store a lambda function for future use.

1.4.2 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.

```
class Person{
ToothBrush* pbrush;

buyNewBrush(string &name){
    if(pbrush != nullptr}{
                delete pbrush;
    }
    if(name == "Orlab")
          pbrush = new Brush();
    }
}
```

```
~Person(){
        if(pbrush != nullptr}{
                delete pbrush;
    }
}
//////////////////
Person Yan();
Yan.buyNewBrush("Oralb");
......Three months later.....
Yan.buyNewBrush("Philip");
}
```

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!**

- **Never say new in c++14!**

1.4.3 Smart pointer Basic knowledge

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

```
template <class T> class auto_ptr{
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr()                 {delete ptr;}
    T& operator*()      {return *ptr;}
    T* operator->()     {return ptr;}
    // ...
};

auto_ptr<int> aupr;
//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.

```
void methodA() {
    unique_ptr<int> buf(new int[256]);

    int result = fillBuf(buf))
    if(result == -1)
        return;
}
```

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!

```
auto_ptr<string> ps (new string("hello world") );
auto_ptr<string> ps1;
ps1 = ps;    // ps will be set null.
ps->size() // it will crash the application.

auto_ptr<string> parray[5]; //compiling error.
auto_ptr<string> ps = parray[2];
//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).

```
string * cp = new string("hello world");
shared_ptr<string> ps(cp);
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.

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

- Smart pointer and const:

```
shared_ptr<T> p;          ---> T * p;
const shared_ptr<T> p;    ---> T * const p;
shared_ptr<const T> p;    ---> const T * p;
const shared_ptr<const T> p; ---> const T * const p;
```

1.4.4 unique_ptr

1.4.4.1 basic

- unique_ptr basic 1-1: create unique_ptr from new. **make_unique is better than inside new, inside new is better than outside new.**

```
string * cp = new string("hello world");

unique_ptr<string> ps = cp; /$\Hilight{8}$/ NOT allow

//ok, but not good style(outside new)
unique_ptr<string> ps (cp);

//good style(inside new)
unique_ptr<string> ps (new string("hello world"));

//best style make_unique
unique_ptr<string> ps( std::make_unique<string>("hello") );
```

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

```
unique_ptr<string> ps1 (new string("hello world"));

unique_ptr<string> ps2 ( ps1 ); //compile error, not allow.
unique_ptr<string> ps2 ( move(ps1) ); //ok
//ownship transfer from ps1 to ps2, nothing delete.
```

- `unique_ptr` 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.**

```
void sink( unique_ptr<widget> arg1, unique_ptr<gadget> arg2);

//1) use move
sink( std::move(exist_uptr_wi), std::move(exist_uptr_ga))

//2) use make function to assure exception safety.
sink( make_unique<widget>(new ...),
make_unique<gadget>(new ...) );  // exception-safe
```

- unique_ptr basic 3: unique_ptr assignment

```
unique_ptr<string> ps1 (new string("ps1"));
unique_ptr<string> ps2 (new string("ps2"));

ps1= ps2; //compile error, not allow

//method1: use move
ps1 = std::move(ps2);
//pointer inside previous ps1 will be deleted.
// pointer inside ps1 point to "ps2" string now.
//pointer inside ps2 will set to null

//method2: reset
```

```
ps1.reset(cp); //ok
//pointer inside previous ps1 will be delete

string* pstr = ps1.release();
// use pstr get pointer managed by ps1.
```

- unique_ptr 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.**

```
unique_ptr<string> ps2 = uqique_ptr<string>(new string("yo") ); //OK

uqique_ptr<string> fun(){
return unique_ptr<string> temp(new string("yan");
}
pu2 = fun(); //OK
```

- unique_ptr basic 4-2: unique_ptr support source and sink idiom.

```
unique_ptr fun() //support source

fun(unique_ptr up);  //use move to support sink
fun(move(other_up));
```

- unique_ptr basic 5: Just observer, not transfer ownership , you can get pointer, or use unique_ptr reference.

```
unique_ptr<string> ps1 (new string("ps1"));

fun(string* pstr);
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.

```
unique_ptr<double []>  pda(new double[5] );
// 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).

```
class B {  // this class can't be copy
public:
   unique_ptr<int> i;
   B():i(new int(0)) { }
};
```

- 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.

```
//only c++ 14 support return type deduction
template<typename... Ts>
auto makeInvestment(Ts&&... params) {
auto delInvmt = [](Investment* pInvestment) {
makeLogEntry(pInvestment); // makedelete
delete pInvestment; // Investment
};

std::unique_ptr<Investment, decltype(delInvmt)>
pInv(nullptr, delInvmt);

if (...){
pInv.reset(new Stock(std::forward<Ts>(params)...));
}
else if (... ) {
pInv.reset(new Bond(std::forward<Ts>(params)...));
}
return pInv; // as before
}
```

- 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

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

1.4.4.2 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 unname unique_ptr, 3) use make_unique .

```
class Fruit { ... };
class Pear : Fruit { ... };
class Tomato : Fruit { ... };

std::vector<std::unique_ptr<Fruit>> m_fruits;
//method 1, bad
m_fruits.emplace_back(new Pear);

//method 2, good
m_fruits.push_back(std::unique_ptr<Fruit>(new Pear));
m_fruits.push_back(std::unique_ptr<Fruit>(new Tomato));

//method 3, best using std::make_unique:
m_fruits.push_back(std::make_unique<Pear>());
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.

```
bool compare_by_uniqptr(
                const unique_ptr<SomeLargeData>& a,
                const unique_ptr<SomeLargeData>& b) {
    return a->id < b->id;
}

sort(vec_byuniqptr.begin(), vec_byuniqptr.end(),
compare_by_uniqptr);
```

```
typedef std::unique_ptr<int> unique_t;
typedef std::vector< unique_t > vector_t;

vector_t vec2(5, unique_t(new Foo));      // Error (Copy)
vector_t vec3(vec1.begin(), vec1.end()); // Error (Copy)
std::copy(vec1.begin(), vec1.end(),
          std::back_inserter(vec2)); // Error (copy)

vector_t vec3(make_move_iterator(vec1.begin()),
                  make_move_iterator(vec1.end())); //Ok

std::sort(vec1.begin(), vec1.end());
// OK, because using Move Assignment Operator
```

1.4.5 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.**

```
string * cp = new string("hello world");

shared_ptr<string> ps = cp; // $|Hilight{8}$NOT allow

//method1: use constructor
shared_ptr<string> ps (cp); //ok, but not good style
shared_ptr<string> ps (new string("hello world")); //good style

//method2: make_shared function
unique_ptr<string> ps( std::make_shared<string>("hello world") );
```

- 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.

```
auto pw = new Widget; // pw is raw ptr

std::shared_ptr<Widget> spw1(pw, loggingDel);
// create control block for *pw
std::shared_ptr<Widget> spw2(pw, loggingDel);
// create 2nd control block for *pw!
//$|Hilight{20}$ BAD! That will cause undefine result!
```

- shared_ptr basic 2: Create shared_ptr from another shared_ptr

```
shared_ptr<string> ps1 (new string("hello world"));

shared_ptr<string> ps2 ( ps1 );
//use_count of ps1 and ps2 are all 2

shared_ptr<string> ps2 ( move(ps1) );
//the original ps1 will become null, and
//the reference count does not get modified.
//Just transfer use_count to ps2.
```

- shared_ptr basic 3: shared_ptr assignment

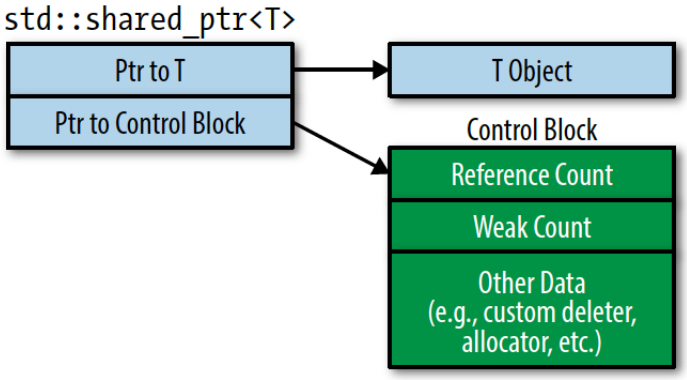
```
shared_ptr<string> ps1 (new string("ps1"));
shared_ptr<string> ps2 (new string("ps2"));

//method1: use assignment
ps1 = ps2;
// 1) previous use_count decrements 1(If equal 0, will delete)
// 2) ps1 points to current use_count
// 3) and current use_count increments 1

//method2: use move
ps1 = std::move(ps2);
// 1) previous use_count decrements 1(If equal 0, will delete)
// 2) ps1 points to current use_count
// 3) ps2 is null now.


//method3: reset
```

```
ps1.reset(cp); //ok
//pointer inside previous ps1 will decrement 1.
//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!

```
shared_ptr<string> ps1 (new string("ps1"));

fun(string* pstr);
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.

```
std::unique_ptr< Widget, decltype(loggingDel) >
                                upw(new Widget, loggingDel);
// deleter type is ptr type

std::shared_ptr<Widget> spw(new Widget, loggingDel);
 // 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.**

```
template< typename T >
struct array_deleter {
  void operator ()( T const * p){
    delete[] p;
  }
};

std::shared_ptr<int> sp( new int[10], array_deleter<int>() );
```

- usage of enable_shared_from

```
#include <iostream>
#include <boost/enable_shared_from_this.hpp>
#include <boost/shared_ptr.hpp>
class Test : public boost::enable_shared_from_this<Test>          //æŤźè£Ż1
{
public:

    ~Test() { std::cout << "Test␣Destructor." << std::endl; }
    boost::shared_ptr<Test> GetObject()
    {
        return shared_from_this();        //æŤźè£Ż2
    }
};
int main(int argc, char *argv[]){
        boost::shared_ptr<Test> p( new Test( ));
        boost::shared_ptr<Test> q = p->GetObject();

    return 0;
}
```

1.4.6 wrapping resource handler in smart pointer

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

- Example for FILE* is below:

```
unique_ptr<std::FILE, decltype(&std::fclose)>
                    fp(std::fopen("demo.txt", "r"), &std::fclose);
if(fp) // fopen could have failed; in which case fp holds a null pointer
```

- We can make FILE* example better, use unique_ptr source semantic.

```
struct FILEDeleter {
    void operator()(FILE *pFile){
        if (pFile)
            fclose(pFile);
    }
};

using FILE_unique_ptr = unique_ptr<FILE, FILEDeleter>;

FILE_unique_ptr make_fopen(const char* fname, const char* mode){
    FILE *fileHandle= nullptr;
    auto err = fopen_s(&fileHandle , fname, mode);
    if (err != 0){
        // print info, handle error if needed...
        return nullptr;
    }
    return FILE_unique_ptr(fileHandle);
}

//usage in your code!
FILE_unique_ptr pInputFilePtr = make_fopen("test.txt", "rb");
if (!pInputFilePtr)
    return false;
```

- If you want to use shared_ptr wrap FILE*, see below example.

```
using FILE_shared_ptr = std::shared_ptr<FILE>;

FILE_shared_ptr make_fopen_shared(const char* fname, const char* mode){
    FILE *fileHandle = nullptr;
    auto err = fopen_s(&fileHandle , fname, mode);
    if (err != 0){
        // handle error if needed
        return nullptr;
    }

    return FILE_shared_ptr(fileHandle , $\Hilight{15}$FILEDeleter());
    //Pay attention!, Here we use FILEDeleter(), but unique_ptr use FILEDeleter
}
```

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

```
struct HANDLEDeleter{
    void operator()(HANDLE handle) const{
        if (handle != INVALID_HANDLE_VALUE)
            CloseHandle(handle);
    }
};

using HANDLE_unique_ptr = unique_ptr<void, HANDLEDeleter>;

HANDLE_unique_ptr make_HANDLE_unique_ptr(HANDLE handle){
    if (handle == INVALID_HANDLE_VALUE || handle == nullptr){
```

```
        // handle error...
        return nullptr;
    }
return HANDLE_unique_ptr(handle);
}

auto hInputFile = make_HANDLE_unique_ptr(CreateFile(strIn, GENERIC_READ, ...));
if (!hInputFile)
    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:

```
struct SvcHandleDeleter{
    typedef SC_HANDLE pointer;
    SvcHandleDeleter() {};

    template<class Other> SvcHandleDeleter(const Other&) {};
    void operator()(pointer h) const {
        CloseServiceHandle(h);
    }
};

typedef std::unique_ptr<SC_HANDLE, SvcHandleDeleter> unique_sch;

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.

```
std::shared_ptr<SC_HANDLE> sp(new SC_HANDLE(::OpenSCManagerA(0, 0, SC_MANAGER_ALL
    [](SC_HANDLE* p){ ::CloseServiceHandle(*p); delete p; });
```

- A common example can be found here:

```
typedef struct {
    int m_int;
    double m_double;
} Foo;

Foo* createObject(int i_val, double d_val) {
    Foo* output = (Foo*)malloc(sizeof(Foo));

    output->m_int = i_val;
    output->m_double = d_val;

    puts("Foo created.");
    return output;
}
```

```
void destroy(Foo* obj) {
    free(obj);
    puts("Foo destroyed.");
}

std::shared_ptr<Foo> foo(createObject(32, 3.14), destroy);
```

```
struct FooDeleter {
    void operator()(Foo* p) const {
        destroy(p);
    }
};
using FooWrapper = std::unique_ptr<Foo, FooDeleter>;
FooWrapper foo(createObject(32, 3.14));
```

- Use shared_ptr to wrap a handle, A good introduction is "Making a HANDLE RAII-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 RAII 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.

```
void setLog(const Foo & foo, const Bar & bar) {
//File file("/path/to/file", File::append);  1) //RAII auto obj
//File* file = new File("/path/to/file", File::append); //2) raw new
   shared_ptr<File> file =
               new File("/path/to/file", File::append); //3) best

   foo.setLogFile(file);
   bar.setLogFile(file);
}
```

- 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. You can use FILE* if you don't mind using C language.

- 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.

```
shared_ptr<fstream> fp{new fstream(name,mode)};
if(!*fp)
  throw No_file{};
ClassA a(fp);
ClassB b(fp);
```

- 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.Is there a reason the same idea was not used for std::unique_ptr?

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.

- by now, shared_ptr and std::functions use type erase technology.

1.4.7 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_ptrs 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.

```
auto spw =
std::make_shared<Widget>();
// the pointed-to Widget's ref count (RC) is 1.

std::weak_ptr<Widget> wpw(spw);
// wpw points to same Widget as spw. RC remains 1
...
spw = nullptr;
// RC goes to 0, and the Widget is destroyed.
// wpw now dangles
if (wpw.expired())
// if wpw doesn't point to an objectâĂę
```

```
std::shared_ptr<Widget> spw1 = wpw.lock();
// if wpw's expired, spw1 is null
auto spw2 = wpw.lock();
// same as above, but uses auto

std::shared_ptr<Widget> spw3(wpw);
// 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.

```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
static std::unordered_map<WidgetID,
std::weak_ptr<const Widget>> cache;
auto objPtr = cache[id].lock();
// objPtr is std::shared_ptr to cached object (or null
// if object's not in cache)
if (!objPtr) { // if not in cache,
objPtr = loadWidget(id); // load it
cache[id] = objPtr; // cache it
}
return objPtr;
}
```

- 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.**

1.4.8 smart pointer and polymorphism

- 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 tempoary base shared_ptr.

```
void doSomething(const std::shared_ptr<Base>& ptr) {
// you must use const
    std::cout<<ptr.use_count()<<std::endl; //cout 2
}

int main() {
    std::shared_ptr<Derived1> pd1 = std::make_shared<Derived1>();
    doSomething(pd1);
        doSomething(static_pointer_cast<Base>(pd1));
}
```

- Most of time, use smart pointer reference just for observe. at this time, you can use raw poiner 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.

```
void doSomething(const std::unique_ptr<Base> ptr) {
    ptr->run();
}

int main() {
    std::unique_ptr<Derived1> pd1 = std::make_unique<Derived1>();
    doSomething(std::move(pd1));
}
```

- For const reference

```
void f(const unique_ptr<Base>  &base)
unique_ptr<Derived> derived = unique_ptr<Derived>(new Derived);
f(derived); //this fail;

f(std::move(derived)); //method 1 work

void f(std::unique_ptr<Derived> const &); //method 2

std::unique_ptr<base> derived = std::make_unique<Derived>(); //method 3
```

- 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 agaion.

```
template<typename TO, typename FROM>
unique_ptr<TO> static_unique_pointer_cast (unique_ptr<FROM>&& old){
    return unique_ptr<TO>{static_cast<TO*>(old.release())};
    //conversion: unique_ptr<FROM>->FROM*->TO*->unique_ptr<TO>
}

unique_ptr<Base> foo = fooFactory();
unique_ptr<Derived> foo2 = static_unique_pointer_cast<Derived>(std::move(foo));
```

1.4.9 make function

- 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::uniqu_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.
  2. The second reason to prefer make functions has to do with exception safety.

```
auto upw1(std::make_unique<Widget>()); // with make func
std::unique_ptr<Widget> upw2(new Widget); // without make func

auto spw1(std::make_shared<Widget>()); // with make func
std::shared_ptr<Widget> spw2(new Widget); // without make func
```

  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.
  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.

```
std::shared_ptr<Widget> spw(new Widget, cusDel);
processWidget(spw, computePriority()); // correct, but not
// optimal; see below

processWidget(std::move(spw), // both efficient and
computePriority()); // exception safe
```

1.4.10 smart pointer Scenario

- Smart pointer general guides:

1. **There are three kinds of place 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.

```
ObserveFun(Foo* p);
ObserveFun(smart_pointer.get() );
ObserveFun(unique_ptr<Foo> &p);
//Not use very often, can be used as fun_obj
// in a container of unique_ptr.

UniqueFun(unique_ptr<Foo> p);
UniqueFun(make_unique_ptr<Foo>(new Foo() )); //get ownership
UniqueFun(move(other_unique_ptr) )  //transfer ownership

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 or 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.

- 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**

- 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

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

- **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.

```
fun(){
Foo fo();
// it make obj directly,
//when you don't need dynamic.
```

```
if(input == "Foo")
        uniqu_ptr<Foo> up(new Foo() );
//When you need new, use uniqu_ptr
}
```

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

  1. As a parameter, pass it to a function. if you don't want to create dynamicly 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, **DonâĂŹt use a const unique_ptr& as a parameter;**
  5. **If you want to transfer ownership to callee from caller, use uniqu_ptr, and use move.** Passing unique_ptr by value means âĂIJsink.âĂİ
  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 youâĂŹre not sure whether or not youâĂŹll 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.

```
Foo *fo = new Foo();  //bad smell here.
fun(Foo * p);
delete fo;

fun(Foo &p); //use reference to improve efficiency

uniqu_ptr<Foo> up(new Foo() );
fun(uniqu_ptr<Foo>& up); //use reference here
 //to avoid copy, uniqu_ptr can't copy

fun(uniqu_ptr<Foo> down);  //prototype
fun(std::move(up) );

std::unique_ptr<std::string> unique = std::make_unique<std::string>("test");
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)

```
Foo* fun(){    //old c style.
........
return new Foo();
}

unique_ptr<Foo> fun(){  // this is better.
........
return unique_ptr<Foo>(new Foo()) ;
}
```

4. Below use shared_ptr, because Server is public used, and no single owner.

```
shared_ptr<Server> buildNewServer(){  // this is better.
return shared_ptr<Server>(new Server()) ;
}

shared_ptr<Server> serverForClass1 = buildNewServer();
shared_ptr<Server> serverForClass2  = serverForClass1;
```

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

Everything You ever wanted to know about move semantics

1.5 reference and rvalue reference

1.5.1 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 initiliaze it. Imaging reference is a kind of top level const pointer.

```
int x = 12;
int* const p = &x; // you have to init it.
p  = &y ; //you can't point to different object.
```

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

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

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

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

3. If you have 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.

```
int *p;
int **p;
int *& rp = p
int &* pr //not illegal in c++
int && rv //rvalue reference.
```

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

```
const int i = 12;
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.

```
//method 1
const int &fun(const int& i){
  return i;
}//Why we need this?
//Just a syntactic demo, no any semantic meaning

//method 2
int &fun(const int& i){
  return const_cast<int&>(i);
}
```

- 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:

```
//case 1:
Foo f(){
return obj;
}
const Foo & rf = f();

//case 2:
class Foo(int i); // ctor
f(const Foo & crf); //function declaration
//here, must const & in f argument.
//if you skip const, compiler will report error:
f(1); //1 -->"temp obj build by ctor" -->crf
```

- `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:

```
class Foo{
Foo(const Foo & foo);
}
```

```
Foo f3 = f1+f2;
// because a temp Foo is produce first from f1+f2
// without const in copy ctor, compiler will bark
```

1.5.2 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?

1.5.2.1 definition of xvalue

- How do rvalues in C++ stored in memory? It's depends! The main reasoning behind this is as always 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.

- 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 and **rvalue reference type**. Each expression belongs to exactly one of the three primary value categories: lvalue, prvalue, and xvalue;

- 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, for example).

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

- 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)

```
int a = 7;  // a is lvalue, It has Identity
int && r1 = 13,  //r1, r2 and r3 are lvalue
int && r2= x+y; //x+y is prvalue
int && r3 = sqrt(2.0);  //sqrt(2.0) is prvalue

cont int & lv = x+y; //lv is lvalue
int & r = x+y; //error,
// lvalue reference can not bind with an rvalue.
```

- 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):

```
std::string s1 = "Test";
//  std::string&& r1 = s1;
// error: rvalue ref can't bind to lvalue

const std::string& r2 = s1 + s1;
// okay: lvalue reference to const extends lifetime
//  r2 += "Test";
// error: can't modify through reference to const

std::string&& r3 = s1 + s1;
// okay: rvalue reference extends lifetime
r3 += "Test";
// okay: can modify through reference to non-const
```

- 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 it xvalue reference.**

- To know this conception can help you to understand decltype. Detail can be seen in the last chapter.

- Summary table:

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

1.5.2.2 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**.

```
string& lvalue_fun();
lvalue_fun(); // after you call this fun, lvalue still exist.
lvalue_fun() = "hello" // you can modify because it persist
&lvalue_fun(); // you can get address because it has identity.
string a = lvalue_fun //call copy ctor, can't move

string pvalue_fun();
pvalue_fun(); // after call this fun, value disappear.
pvalue_fun() = "hello" //NOT modify because it doesn't persist
&pvalue_fun(); // NOT get address because it no identity.
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.**

- 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.

```
struct A {
int m;
};

A&& operator+(A, A); // a+a is xvalue
A&& f();  //f() is xvalue
f().m // f().m is also xvluae
A a;
A&& ar = static_cast<A&&>(a);
//static_cast<A&&>(a) is xvalue, but ar is lvalue
```

1.5.2.3 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**;

```
void foo(int&& t) {
// t is initialized with an rvalue expression
// but is actually an lvalue expression itself
}

std::string a;   //a, b, c are all lvalue.
```

```
std::string& b;
std::string&& c;
```

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

```
X&& goo();
X x = goo();
// calls X(X&& rhs) because the thing on
// the right hand side has no name

std::move(x)  // get rid of x name,
//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.

```
SomeType &&Func() { ... }

SomeType &&val = Func();
SomeType otherVal{val}; // Do you really want to move
....
cout<<val; //what happen if you have forget you have move.
//so here, we think val is lvalue. because it's a named rvalue reference.
```

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

```
Derived(Derived&& rhs)
: Base(rhs) // wrong: rhs is an lvalue
{
// Derived-specific stuff
}

Derived(Derived&& rhs)
: Base(std::move(rhs)) // good, calls Base(Base&& rhs)
{
// Derived-specific stuff
}
```

1.5.2.4 Why need xvalue

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

```
template<class T>
void swap(T& a, T& b) {
T tmp(std::move(a));
a = std::move(b);
b = std::move(tmp);
}
```

- 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 introduced a new value type–xvalue.**

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

- In one word:
  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.

1.5.3 rvalue reference and move scenario

- rvalue reference can only bind to rvalue, and rvalue is composed of prvalue+ xvalue. A plain reference can only bind to lvalue. Const lvalue reference can bind to both lvalue and rvalue, but you can't change it.

- because const reference can bind both lvalue and rvalue, so we can't distinguish when to move, when to copy, that is why we need 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 allow a function to branch at compile time (via overload resolution) on the condition "Am I being called on an lvalue or an rvalue?" It is true that you can overload any function in this manner, as shown above. **But in the overwhelming majority of cases, this kind of overload should occur only for copy constructors and assignment operators, for the purpose of achieving move semantics.**

```
void foo(X& x); // lvalue reference overload
void foo(X&& x); // rvalue reference overload
```

```
X x;
X foobar(); //return a X obj

foo(x); // argument is lvalue: calls foo(X&)
foo(std::move(x) // argument is xvalue: calls foo(x&&)
foo(foobar()); // argument is rvalue: calls foo(X&&)
```

- 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.

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

- You need to know, previous codes and explanations just give the basic idea of rvalue reference. Just like function pointer, we never use function pointer in simple way, instead, we use it as a function argument to implement call back. **For rvalue reference, We just use it as move semantic.** I will explain it in the next section.
- given a type T, you can have lvalues of type T as well as rvalues of typeT. It's especially important to remember this when dealing with a parameter of rvalue reference type, because the parameter itself is an lvalue.
- how funciton overload kick in?

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

- forward reference can be type dedcuted both lvalue and rvalue. T will be different depend on the value categorye
-

1.5.4 move copy and assignment

- **copy ctor will copy a obj inside of function, So all the knowledge discussed in previous section can be used to help you understand copy ctor. A better method is overload function to deal with lvalue and rvalue separately.**
- Basic copy ctor syntax explanation:

```
class obj = 2 //same as class obj(2)
                    // call single argument ctor

obj =2          //call assignment operator
```

```
class obj1 = obj2 // call  copy ctor
                          // not call assignment operator

obj1 = obj2          //call assignment operator

class obj1 = obj2+obj3    //call move ctor
//if you dont' have move ctor, It will call copy ctor

obj1 = obj2+obj3 //call move assignment operator  if you define.
```

- **Typically, if your class allocate a lot of allocated resource (new, or manually manage some resources). You should implement two copy ctor** 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 destory 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

```
Foo::Foo(const Foo & foo){
  while(ptr++;)
    ptr[i] = foo.ptr[i]  //expensive copy
}

Foo::Foo(Foo && rhs){ //no const here
   ptr = other.ptr;  //efficient move(steal)
  rhs.ptr = nullptr;
}

Foo& Foo::operator=(Foo&& rhs){
   delete[] ptr;
   ptr = other.ptr;  //efficient move(steal)
   rhs.ptr = nullptr;
   return *this;
}
```

- 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.

```
Foo Foo::operator+( const Foo & f) const {
    Foo temp = Foo(n+f.n);
    //copy happen here.
    return temp;
}

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. (becasue 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).

- 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.

```
Foo choices[10];
Foo best;
best = choices[3];
//I don't want to keep choices after I pick up what I want
//here, It will call normal move assignment,
//because choices[3] is not rvalue.

best = std::move(choices[3]);
//It will call move assignment
```

1.5.5 universal reference parameter

1.5.5.1 basic

- **Universal reference has been renamed as forward reference. It's more descriptive name, It tell you universal reference should always been used with forward.**

- 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.

```
auto&& var2 = var1; // universal reference

template<typename T>
void f(T&& param); // universal  reference

template<class T, class Allocator = allocator<T>>
class vector {
template <class... Args>
void emplace_back(Args&&... args);  //args is universal reference
};
```

```
//--------------below are not universal reference-----
template<typename T>
void f(std::vector<T>&& param); // rvalue reference
// form is quite constrained. It must be precisely "T&&".

template<typename T>
void f(const T&& param); // with const

template<class T, class Allocator = allocator<T>> // from C++
class vector { // Standards
public:
void push_back(T&& x); //no type deduction
...
};
```

- 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:

```
A& & becomes A&
A& && becomes A&
A&& & becomes A&
A&& && becomes A&&
```

- There is a special template argument deduction rule for function templates that take an argument by rvalue reference to a template argument:

```
template<typename T>
void foo(T&&);
```

  Here, the following apply:

  1. When foo is called on an lvalue of type A, then T resolves to A& and hence, by the reference collapsing rules above, the argument type effectively becomes A&.
  2. When foo is called on an rvalue of type A, then T resolves to A, and hence the argument type becomes A&&.

- 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.

```
auto&& var = some_expression_that_may_be_rvalue_or_lvalue;
// var was initialized with either an lvalue or rvalue, but var itself
// is an lvalue because named rvalues are lvalues
use_it_elsewhere($\Hilight{9}$std::forward<decltype(var)>(var));
```

- A good example of " some_expression_that_may_be_rvalue_or_lvalue;" under this way, only auto&& can deal with two different condition.

```
std::vector<int> global_vec{1, 2, 3, 4};

template <typename T>
T get_vector(){
  return global_vec;
}

template <typename T>
void foo(){
  auto&& vec = get_vector<T>();
  auto i = std::begin(vec);
  (*i)++;
  std::cout << vec[0] << std::endl;
}

foo<std::vector<int>>();
std::cout << global_vec[0] << std::endl;
foo<std::vector<int>&>();
std::cout << global_vec[0] << std::endl;
```

```
if (std::is_lvalue_reference<decltype(var)>::value) {
  // var was initialised with an lvalue expression
} else if (std::is_rvalue_reference<decltype(var)>::value) {
  // var was initialised with an rvalue expression
}
```

1.5.5.2 advantage

- Below, there are some reasons that I prefer to use universal reference:

  1. It can provide you unify interface, with variadic template parameter, it support variadic number. That is make_unique and make_shared and emplace-kind funciton possible.
  2. 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.
  3. For overload method, more source code to write and maintain (two functions instead of a single template).
  4. 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.

```
class Widget {
public:
 template<typename T>
 void setName(T&& newName) // newName is universal reference
 { name = std::forward<T>(newName); }

 ...
 string name;
};
```

5. When you use forward, you can use emplace more effectively.
6. 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

```
template<class T, class... Args> // from C++11
shared_ptr<T> make_shared(Args&&... args); // Standard

template<class T, class... Args> // from C++14
unique_ptr<T> make_unique(Args&&... args); // Standard
```

1.5.5.3 disadvantage

- 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). At the same time, 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).

```
template<typename T> // reference
fun(T&& value){
vector<Foo> vect;
vect.push_back(std::forward<T>(value)); // use move here to implement
}
```

- 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"

1.5.5.4 When to use

- 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.

  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 contaner, only has POD type value, move is just like copy. You don't need to use universal references.** Most of time, in your universial function, you have a container(string is container too), because move container is much cheaper than copy it.
  4. **For universial referene, you must forward it to ctor, container, object or function. and these four things will take different action toward lvalue and rvlaue.**
  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.

```
auto&& vec = some_expression_that_may_be_rvalue_or_lvalue;
auto i = std::begin(vec);
(*i)++;

auto       => will copy the vector, but we wanted a reference
auto&      => will only bind to modifiable lvalues
const auto& => will bind to anything but make it const, giving us const_iter
const auto&& => will bind only to rvalues
$\Hilight{19}$//with const, It's not univerisal reference any more.
```

1.5.5.5 example

- There are three function emplace, make_shared and make_unique. They use:

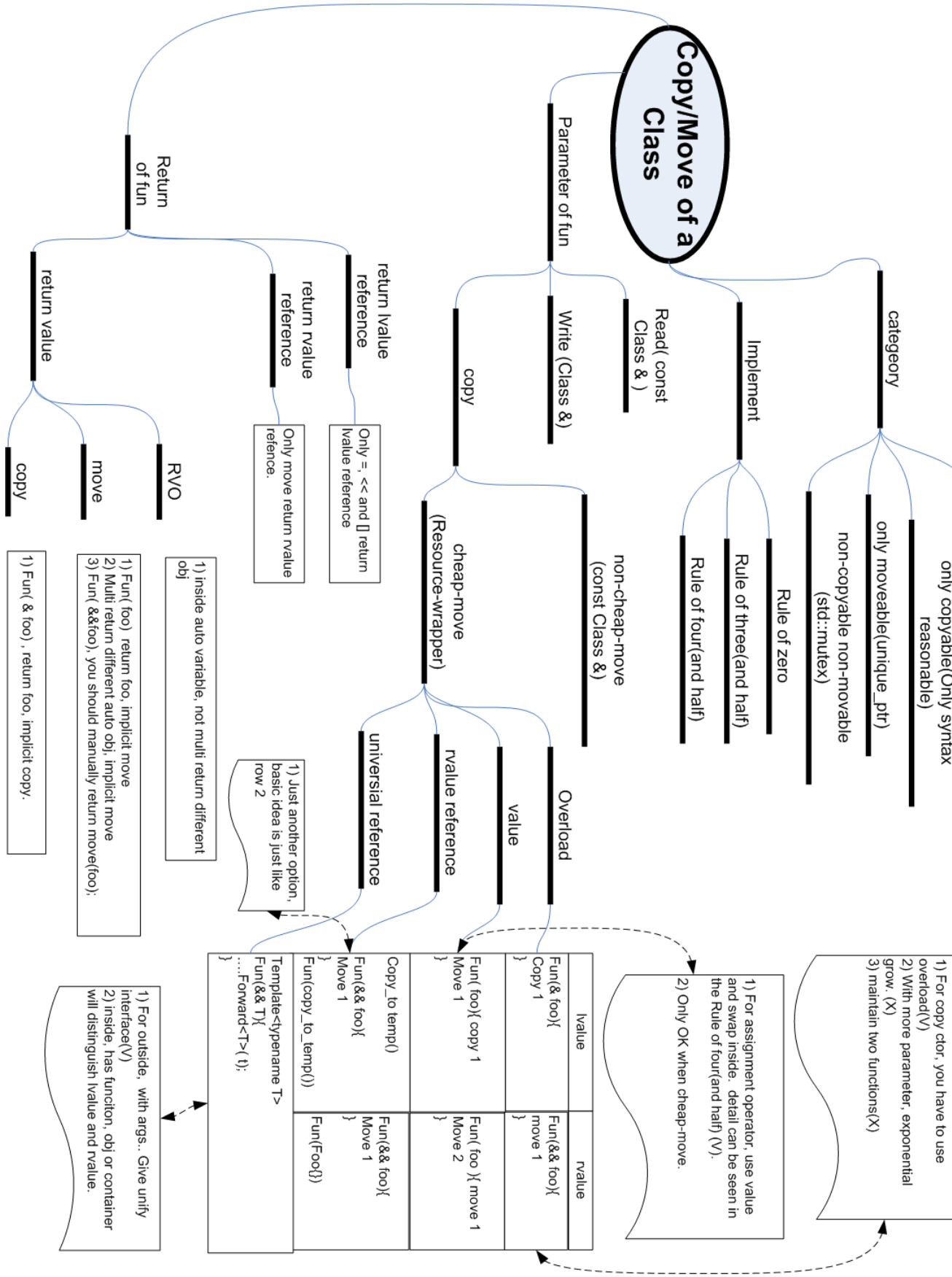
  1. **variadic template**, because it need to receive **any number and any type** parameter.
  2. **universal reference**, because it's template, and I also want to keep rvalue semantic to improve effiencie.

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.

1.5.6 function interface design

1.5.6.1 parameter design-basic

- Main content in this section illustrated below:



- The move ctor and move assignment operator give us an idea how to improve the function interface design. By now, you imagine you have a function, then you have three operations inside: **1) read 2) copy 3) write.**

- For read, "const type&" is enough, For write, "type&" is good, because I don't need to write sth into prvalue. Based on previous explanation, below code will not compile.

```
writeFun(Foo &);

writeFun(foo1+foo2) // will not compile, foo1+foo2 is rvalue.
```

- Another tips is "DonâĂŹt copy your function arguments. Instead, pass them by value and let the compiler do the copying." You can google "Want speed pass by value" One place 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:

```
T& T::operator=(T const& x) // x is a reference to the source
{
    T tmp(x);           // copy construction of tmp does the hard work
    swap(*this, tmp);   // trade our resources for tmp's
    return *this;       // our (old) resources get destroyed with tmp
}

// A better one is below
T& operator=(T x)     // x is a copy of the source; hard work already done
{
    swap(*this, x);   // trade our resources for x's
    return *this;     // our (old) resources get destroyed with x
}
```

- If a function is value return, rvalue reference parameter, you can use move in the return .

```
Matrix // by-value return , by rvalue parameter
operator+(Matrix&& lhs, const Matrix& rhs){
 lhs += rhs;
  return std::move(lhs); // move lhs into return value
  return lhs // will copy lhs into return value
}
```

- the same idea apply to universial reference. At this time, you have to use forward

```
template<typename T>
Fraction // by-value return
reduceAndCopy(T&& frac) // universal reference param
{
 frac.reduce();
 return std::forward<T>(frac); // move rvalue into return
} /
```

- For other condition, don't use move at all. 1) for local varaible 2) for function paramter 3) multi return points, compiler will use move implicitly.

```
Widget makeWidget(){
 Widget w;

 return std::move(w); //Bad, no RVO here
 return w; //Good.
} //
```

```
Widget makeWidget( Widget w;){

 return std::move(w); //Bad, rudundant,
 return w; //Good, compiler will use move here implicitly.
} //
```

- For copy, thing become interesting. Because for rvalue, we can move directly. Our goal is that we can use move for rvalue, so there are three options:

  1. overload copyFun to support const Foo& and Foo &&.
  2. Use pass-value fun(Foo) or Use pass-rvalue-reference fun(Foo&&)
  3. 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&&> copyFun(T &&)	copy inside	move inside

- Use Overload function to deal with rvalue only. So usually function to accept rvalue reference doesn't exist by itself, it just stay with another version to accept lvalue, fun(const Foo& foo);

```
//rvalue reference
fun(Foo&& foo);
fun(foo1+foo2) // will compile, It means that inside fun
// you will steal resource.
fun(f_return_foo()); // call move ctor.
```

- Next, No overload, just pass value to deal with lvalue and rvalue at the same time. this idea can be seen in the "more effective C++ item ?"

```
struct S{
    void initByVal(SomeType param); // can now steal from param
};

SomeType t;
s.initByVal(t); // deal with lvalue
s.initByVal(std::move(t)); // move data to param, then steal inside
```

```
fun(Foo value){
vector<Foo> vect;
vect.push_back(std::move(value)); //use move to implement
}
```

- Last, No overload, 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";

```
struct S{
    void initByRef(SomeType&& param);
};

SomeType t;
s.initByRef(std::move(t)); // this works, and even better (less moves)
s.initByRef(SomeType()); // this also works
```

- But for lvalue, you need a help function here.

```
template<typename T>
T copy_to_temp(const T& t) { return t; }

SomeType t; // I'll need t later, so cannot std::move(t)
s.initByRef(copy_to_temp(t)); // but can copy

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>)

```
template<typename T>
T move_to_temp(T& t) { return std::move(t); }

void foo(unique_ptr<SomeType>&& param);

unique_ptr<SomeType> p;
foo(move_to_temp(p)); // note: this does move, p will become empty
foo(std::move(p)); //here, p will not become empty.
```

- More explanation.

```
template<typename T>
T move_to_temp1(T& t) { return std::move(t); }
//here move ctor is called when return to a value


T&& move_to_temp2(T& t) { return std::move(t);}
//here no move ctor is called, just reference value assignment.

void foo(unique_ptr<SomeType>&& param);

unique_ptr<SomeType> p;
foo(move_to_temp1(p));  //compile ok, return value is prvalue
foo(move_to_temp2(p));  //compile ok, return value is xvalue
```

1.5.6.2 RVO and copy elision

- RVO is a kind of copy elision.
- RVO is only effective for 1)local variable. 2)the type of the local object is the same as that returned by the function.

```
Matrix operator+(Matrix&& lhs, const Matrix& rhs){
...
return lhs; //here, no RVO.
return move(lhs); //A better solution.
}
```

- When you function return a value.

  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.

- Foo f1 = fun(); When function finish, ctor is called first time, from auto->temp. Then, ctor is called second time, temp->f1. With RVO, It can be build on f1 directly, no ctor is called at all.

```
// operator+ and fun has same idea, both return obj value.
Foo operator+(const Foo& other){
  Foo temp;
  ......
  return temp;
}

Foo fun(){
  Foo temp();
  return temp;
}

//experiment 1: g++ -fno-elide-constructor, turn off RVO
fun(); // call ctor once;
Foo foo1 = fun(); //call ctor twice
Foo& foo2 = fun(); // compile error
const Foo& foo3 = fun(); //call ctor once

//experiment 2: define move ctor
fun(); // call move ctor once;
Foo foo1 = fun(); //call move ctor twice
const Foo& foo3 = fun()
//call move ctor once auto->temp

//experiment 3:
Foo&& foo4 = fun() //OK,  foo4's recourse can be stolen.

//experiment 4: g++ turn on RVO
fun(); // call nothing;
Foo foo1 = fun(); //call nothing
```

- In theory, with RVO, you don't need write move ctor any more. but there are still some usage, such as:

  1. For multi return, no RVO, such as fun() below.

```
Foo fun(){
    if(condition)
      return Foo(1);
    else
      return Foo(2);
```

  2. swap fun

```
swap( Foo & f1  , Foo& f2){
    temp = move(f1);
    f1 = move(f2);
    f2 = move(temp)
```

  3. vector container include rvalue version and value version.

```
vector.push_back(Foo(1)); //better

Foo f1 = Foo(1);
vector.push_back(f1);
//Don't need f1, you can use
vector.push_back(std::move(f1));
```

1.5.6.3 parameter design-value

- usually, 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.

```
std::vector<std::string>
sorted(std::vector<std::string> names){
    std::sort(names);
    return names;
}

// names is an lvalue; a copy is required so we don't modify names
std::vector<std::string> sorted_names1 = sorted( names );

// get_names() is an rvalue expression; we can omit the copy!
std::vector<std::string> sorted_names2 = sorted( get_names() );
```

```
std::vector<std::string>
sorted2(std::vector<std::string> const& names) {
    std::vector<std::string> r(names);    // and explicitly copied
    std::sort(r);
    return r;
}
```

- A basic explanation can be found here:

	lvalue	rvalue

Foo Fun(){
Foo foo;
Return foo;
}
	Foo foo = Fun();
	//One ctor, with RVO	

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.

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(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

Foo Fun(Foo&& foo){
Return std::move(foo);
Return foo;
}
	XXX	Foo foo = Fun(Foo{});
		// with std::move, one move
		// without move, one copy

Left value input	ctor	Copy	move
Ref parameter	1	1	0
Value parameter	1	1	1

Right value input	ctor	Copy	move
Ref parameter	1	1	0
Value parameter	1	0	1

- basic analysis:

  1. For reference, When return, you can have RVO, no copy or move at all. It's good and green color.
  2. For value, When return, you have to have one move, It's bad and color is red.
  3. For lvalue, one more move, but for rvalue, although one more move, but I don't need a copy (Copy-elision at parameter).It's good and color is blue.

- summary:

  1. For c++03, we don't 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.

- Two good articles about this topic are:"Want Speed? Pass by Value." and "WANT SPEED? DONâĂŹT (ALWAYS) PASS BY VALUE."

1.5.6.4 return design-return value

- **Three basic knowledges about function return:**

1. It's very important understand there are two phrases when you return from function. The first step is from inside fun fauto to ouside of function ftemp(unname tempory), then fauto disappear. Then in second step, **Move** from ftemp to flast. because ftemp is rvalue.

```
Foo fun(){
 return fauto;
}

Foo flast = ($\Hilight{9}$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

```
Foo fun(){
 return fauto;
}

const Foo& flast = ($\Hilight{9}$ftemp created here)fun();
Foo&& flast = fun();
```

- For local variable or value parameter, don't use move. compiler will use RVO or default think return value is rvalue. Return local value can use RVO and move, so It has already had high efficiency.

```
Foo fun(){
Foo foo;
return foo;
$\Hilight{12}$//Don't use move here.
}

Foo fun(Foo foo){
return foo;
//Don't use move here.
}
```

1.5.6.5 return design-return plain reference

- Now talk about return reference: Never return local variable, it will cause dangling problem, So return plain reference only heppen 1) You input a reference

first, such as overload «. 2) member function return some member data, such as copy ctor and overload [] inside a class.

- 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

```
string& a = FindAddr( emps, "John Doe" );
emps.clear(); // This statement will invalid a.
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.

```
const string&
FindAddr( /* pass emps and name by reference */ ){
for( /* ... */ ){
if( i->name == name ){
   return i->addr;
}
}
static const string empty;
return empty;
}
```

- Why don't we usualy return plain reference?

  1. You want to return a reference to avoid copy of auto obj inside of function, but in the end you will get a dangling reference. It's very a dangerous action. **Don't return reference just for efficient return, It's unnecessary and dangerous**.
  2. You don't need return rvalue reference&& either. return value ftemp outside of fun is always rvalue. ftemp->flast is move. So you don't need to declare Foo&& fun
  3. 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?

- 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.

```
operator =
operator []
operator << and >>
```

- **Don't return reference or pointer to private member variables through you public member functions. It will break encapsulation.**

1.5.6.6 return design-return rvalue reference

- 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.

```
A&& rrfun1(A&& arg){
return std::move(arg);
//You have to use move here,
//because arg is lvalue.
}

A&& rrfun2(A& arg){
return std::move(arg);
}

   A a;
   A b= rrfun1(std::move(a) );
   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**.

```
A rrfun(A& arg){
return std::move(arg);
}

 A a;
 A&& b = rrfun(a);
 //after this a is invalid, and b refer a ftemp
```

- return rvalue reference 3: 1)ftemp type is A 2)below code will call move ctor once(arg move to b) 2)turn off RVO, then arg move ftemp, then ftemp move to b.3)move ctor once, arg move to ftemp. b prolong ftemp life.(Why do you want to do that?)

```
A rrfun(A& arg){
return std::move(arg);
}

 A a;
 A b = rrfun(a); // move ctor once
 A b = rrfun(a); //move ctor twice with -fno-elide-constructors
 A&& b = rrfun(a); // move ctor once.
```

- return rvalue reference 4: below code will call move ctor once from ftemp to b.

```
A&& rrfun(A& arg){
return std::move(arg);
}

 A a;
 A b = rrfun(a);
```

- return rvalue reference 5: 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.

```
A&& rrfun(A& arg){
return std::move(arg);
}

 A a;
 A&& b = rrfun(a);
```

- return rvalue reference 6: below code will call copy ctor once. **name rvalue reference is lvalue, you should use std::move to force move it**

```
A rrfun(A& arg){
return move(arg); //call move ctor
return (arg); // call copy ctor
}

A a;
 A b = rrfun(a);
```

- return rvalue reference 7: Why we use move in below example? lhs inside is lvalue, not auto variable, so we need to copy it to outside ftemp. But you can see lhs type is rvalue refence, so you can use std::move to force move it.

```
Matrix operator+(Matrix&& lhs, const Matrix& rhs) {
    lhs += rhs;
    return std::move(lhs); // move lhs into outside function.
}
```

- return rvalue reference 8: All the previous example, rrfun argument should be A&& arg, I just focus on return part, so I simplify the argument to a plain reference. But you should know, std::move(plain reference) is not good design. You only can do it if you STRONG sure that you don't want use obj after you std::move(obj).

- You can understan all these examples by three function knowledge int he begining. And all these exmaples are just have academic meaning, no any practial sense at all. Why?

- **rref is ref first, so if we have reason don't return ref from function, all these reason is also valid for rref.**

- If a function return rref, 1)it must receive rref first, 2) inside function, he modify rref, 3) return this rref to another rref. These requirment is so werid. Why not modify the first rref directly. Just like we seldome receive a plain ref and return plain ref again. Because all your modification has affect input plain ref. if return plain ref is unnessary, return another rref is dangerous, because it will cause dangling problem, because rref refer a temporary varaible, it can disappaer or move at any time.

- Why return value is better than return rvalue reference? 1) more clear semantic 2) when use with auto, it's support RVO.

```
DataType data() && { return std::move(values); } // why DataType?
auto values = makeWidget().data();
//with ROV, just move once.

DataType && data() && { return std::move(values); }
auto&& values = makeWidget().data();
//values will be dangling
```

- Another implicit dangling case.

```
int&& fun(int&& a){
    return std::move(a);
}
int main(){
   int&& a = 1+2;
   int&& b = fun(move(a));
   int&& c = fun(1+2);
    cout<<b<<" "<<c<<endl;
    printf("Hello World");

    return 0;
}
```

- **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 movable lvalue, move; For universal refence, forward.**

```
Matrix operator+(Matrix& lhs, const Matrix& rhs) {
    return lhs
}

Matrix operator+(Matrix&& lhs, const Matrix& rhs) {
    return move(lhs)
}

template<type T>
T operator+(T&& lhs) {
    return forward<T>(lhs)
}
```

- By now, I only know std::move return rvalue reference. Function return rref is rare. ab is neither a local automatic nor a temporary rvalue. Now, the ref-qualifier && says that the second function is invoked on rvalue temporaries, making the following move, instead of copy

```
struct Beta {
  Beta_ab ab;
  Beta_ab const& getAB() const& { return ab; }
  Beta_ab && getAB() && { return move(ab); }
   $/Hilight{30}$// return && is not Good interface design.
```

```
    Beta_ab ab = Beta().getAB();
    // It will call move version.
    //Beta_ab && ab = Beta().getAB(); ab is dangling rref

  2) Beta_ab getAB() && { return move(ab); }
  $/Hilight{30}$//Good interface design.
};
```

- 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) What are copy elision and return value optimization?

  4) One variable init form to rule them all, via mandatory elision.

  5) Episode Eleven: To Kill a Move Constructor

1.6 OOP

1.6.1 Design

1.6.1.1 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

```
template<class T, class Allocator =
                          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.

```
template< class T >
struct is_integral{
    static const bool value
     /* = true if T is integral, false otherwise */;
    typedef std::integral_constant<bool, value> type;
};

template <class T>
T f(T i){
    static_assert(std::is_integral<T>::value, "Integer required.");
    return i;
}

int main() {
    std::cout << f(123) << '\n'; //output 123
}
```

1.6.1.2 relationships between classes

- OOP has four relationships:

  1. Composition: Ownership, same life time, not change in the middle. Car and engine
  2. Aggregation: Ownership, maybe same life time, may change in the middle, People and toothbrush(same life time, changeable) Teacher and student(not same life time, changeable), Container and pointer(same life time, changeable).
  3. Associate: No Ownership, Includes as a member, different life time, person and father(Not Nullity), person and wife(Nullity)
  4. Dependency: No Ownership, Not Includes as a member, person and friend.
     (a) Dependency definition: If class X's member function argument is class Y, X is dependency of Y.
     (b) 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.

- OOP example 1: If it's a mother-son(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.

```
class Car{
Engine eng; // member obj, not use pointer here.
Car(int carArg, int engArg): eng(engArg){}
}
```

- OOP example 2: If it's an aggregation relationship, If has same life time, use uniqu_ptr. If you want to change, use uniqu_ptr reset function or move semantic from another uniqu_ptr. TootuBrush example, use unique_ptr.

```
class Person{
unique_ptr<Brush> unpbrush;

buyNewBrush(string &name){
    unpbrush.reset(new Brush());
// you can't use unpbrush= new Brush()
// unpbrush assignment only support( unique_ptr<T> &&);
// use reset, it will make original deleted automatically
}

// don't need destructor any more.
}
```

- 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 uniqu_ptr at all.**

- OOP example 3-1: What's different with raw pointer, weak_ptr or shared_ptr?

```
class Man{
Woman*  wife; // can be set to nullptr.
//maybe change or live longer than you.

weak_ptr<>

Woman &mother; //must give a mother in initial list
}
```

- OOP example 4: if it's Dependency, such as friend relationship, Most of time, we just use pointer or reference as function parameter.

```
class Man{
lendMoney(Friend* mike);
}
//just use it in function. not a member of class.
```

- OOP example 5: Suppose Man and Computer is has-a relationship 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.

```
class Worker{
Computer m_desktop;
Worker (Computer u): m_desktop(u){}
}

Computer commonComputer;
Worker Yan(commonComputer);
Worker Gang(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.

```
class Unit{
shared_ptr<Bus> shr_p_bus;
}
```

1.6.1.3 Has-a relationship

- Interestingly, 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:

```
class B                      { /*...*/ };
class D_priv : private   B { /*...*/ };
class D_prot : protected B { /*...*/ };
class D_publ : public    B { /*...*/ };
class UserClass              { B b; /*...*/ };
```

  1. None of the derived classes can access anything that is private in B.
  2. In D_priv, the public and protected parts of B are private.
  3. 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.
  4. 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).
  5. 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:

```
class D_prot : protected B {
public:
  using B::f;  // Note: Not using B::f(int,float)
};
```

- 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:

```
class Student{ // 1) use composition.
int getNameLen();
private:
string m_name;
vector<double> score;
};
int Student::getNameLen(){
  return m_name.length();
}
```

```
class Student: private string, private std::vector<double>{
int getNameLen();
// We don't need private memeber data any more
}

int Student::getNameLen(){
  return string::length();
  //use class name and scope-resolution operator
}
const string& Student::getName(){
  return (const string&) *this;
}
```

- **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).

```
class Brake {
public: virtual void stopCar() = 0;
};

class BrakeWithABS : public Brake {
public: void stopCar() { ... }
};
```

```
class Car {
  Brake* _brake;
public:
  Car(Brake* brake) : _brake(brake) { brake->stopCar(); }
};
```

2. Version 2a: Template

```
template<class Brake>
class Car {
  Brake brake;
public:
  Car(){ brake.stopCar(); }
};
```

3. Version 2b: Template and private inheritance

```
template<class Brake>
class Car : private Brake {
  using Brake::stopCar;
public:
  Car(){ Brake::stopCar(); }
};
```

- 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.

1.6.1.4 Is-a relationship

- 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.**

- About Inheritance, There are three principles.

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

- 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 refenence 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.

```
class Car{
.....................
Power* pow
};

Car::start(){
............
pow->ignite();
..........
}

class Power{
  virtual ignite();
};
class GasPower : public Power
class ElePower : public Power

Car car(gas);
car.start();
```

- 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.

- Consider making virtual functions nonpublic, and public function nonvirtual. This is similar with Template Method design pattern. C++ coding standards Item 39

- A common mistake, is inherit 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 class myString : public string.

- 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.

```
myFind(const string& str){
//you custimized behavior.
}
// use myFind(str) and myStr.find()
//I think only difference is syntax.
```

- Even you have specific state, you also can use composition to avoid inheritance.

```
class MyString{
size_t length(){return str.length}
//just need to write forward function here;

myFind(){
  //You customized behavior.
}

std::string str;
}
```

- **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 squre 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. e.g. A Bird may need only the fly behavior of an Airplane. 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.

```
class Shape{
virtual draw() = 0 // you have to re
}

class Circle: public Shape{
}
```

```
class Flyable{
fly(){}
};

class bird{
Flyable* fa
fly(){fa->fly()}
}
```

- No source code, just header and lib, you also can use inheritance in C++ language.
- Pure virtual class can't be instance, it just a abstract interface, it's agreement.
- Both reference and pointer support polymorphism.
- 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)

```
class shape{
virtual draw() = 0 // you have to re
virtual error(); // there is default implement.
                 // but you may change it.
int objectID(); // you don't need to rewrite it.
}
```

- Sometimes, syntax is right, but it doesn't mean it is logical or design right. For example, You can define a virtual function in base class, but if it's not pure virtual, A new derived class will inheritance default implementation. In this way, a better method is make it become pure virtual. It will oblige all the derive class give it's own implementation. It can be thought as strategy pattern, it looks like template method, but it's not exactly the same. If two derived class share the same implementation, you can upgrade the implementation to base class, and make a protect member function, it make derive class can access it in it's own implementation base class pure virtual function. A practical example can be seen in effective C++ item 36 Airplane example.

1.6.1.5 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:

```
class Widget {
public:
void doWork() &; // this version of doWork applies
// only when *this is an lvalue
void doWork() &&; // this version of doWork applies
}; // only when *this is an rvalue

Widget makeWidget(); // factory function (returns rvalue)
Widget w; // normal object (an lvalue)
```

```
w.doWork();
// calls Widget::doWork for lvalues (i.e., Widget::doWork &)
makeWidget().doWork();
// 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.

```
class Base {
public:
virtual void mf1() const;
virtual void mf2(int x);
virtual void mf3() &;
void mf4() const;
};
class Derived: public Base {
public:
virtual void mf1();
virtual void mf2(unsigned int x);
virtual void mf3() &&;
void mf4() const;
};
```

- override specifier should be used in derived class member funcition used to check if they are match with member function in base class.

```
struct A{
    virtual void foo();
    void bar();
};

struct B : A{
    void foo() override; // OK: B::foo overrides A::foo
    void bar() override; // Error: A::bar is not virtual
}
```

- Use final in Base class, to stop sub class override.

```
class Base{
virtual void method1() final;
}
```

1.6.1.6 MI or bridge

- Multiple Inheritance will cause one grandson has two copy of grandfathers. A solution is use virtual key word: 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.

```
Father 1: virtual public grandfather
Father2: virtual public grandfather
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:

	Grow Gracefully?	Low Code Bulk?	Fine Grained Control?	Static Detect Bad Combos?	Polymorphic on Both Sides?	Share Common Code?
Bridge	☺	☺ (N+M chunks)	–	–	–	☺
Nested generalization	–	– (N×M chunks)	☺	☺	–	–
Multiple inheritance	–	– (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" doesnâĂŹt 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.

1.6.2 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 it's 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.

```
//below is string.h file
#pragma once
namespace Yan{
class String{
public:
String();  //default constructor
String(const char *a ); // specify constructor

String (const String &);  //copy ctor
String (String && other); //move copy ctor

String& operator=(const String &); //assignment
String& operator=(String&& other); //move assignment
String& operator=(const char*a); // option.

~String();  //usually, you should have these Seven member
              //functions if you use new inside your class.

friend ostream& operator<<(ostream & os, const String & st);
friend istream& operator>>(istream & is, String &st);

private:
const static int NUM= 1000; // const used inside of this class.
char* m_str;

};
ostream& operator<<(ostream & os, const String & st);
istream& operator>>(istream & is, String &st);
}
```

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. 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.

```
//user.cpp
String s;
s = "aaa"  //two actions
String s("aaa");   //one actions
String s{"aaa"};  // new feature in c++11
String s={"aaa"}; // same as previous one

String str; char temp[40];
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.

```
Class Matrix{
//100 member function.
} ; //bad design.

//Good design with small class with
nonmember function.
namespace MAT{
Class Matrix{
//core data and member function
}

Cal1(Matrix &);
Cal2(Matrix m1, Matrix m2)
......
}
```

- 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++ item31to 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 nonmember
  4. can be implement using the class public interface alone, use nonmember.

1.6.3 Special member functions

1.6.3.1 Basic

- When you write a empty class, compiler will produce at least six member functions. New compiler will produce move ctor and move assignment too.

```
class Empty{};

Empty();
Empty(const Empty& rhs);
Empty& oprator=(const empty & rhs);
Empty* operator&(){return this};
const Empty* operator&() const ;
~Empty();
```

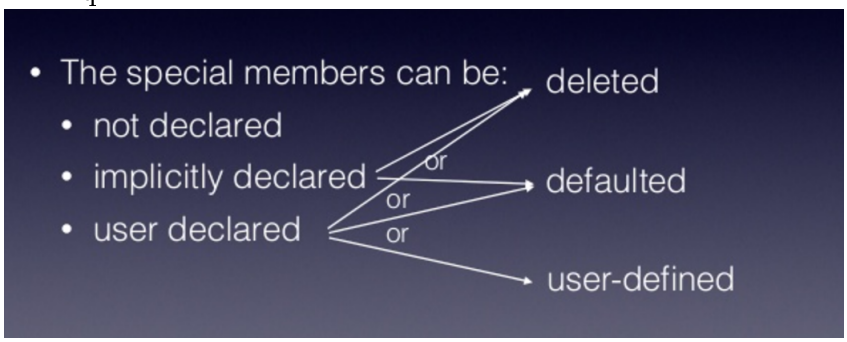
- Why you need to pay attention to these special member functions? Given half a chance, the compiler will write them for you. Another reason is that C++ by default treats classes as value-like types, but not all types are value-like. Know when to write and disable them make you get correct code.

- A good reference is "Everything You ever wanted to know about move semantics" in slideshare.net.

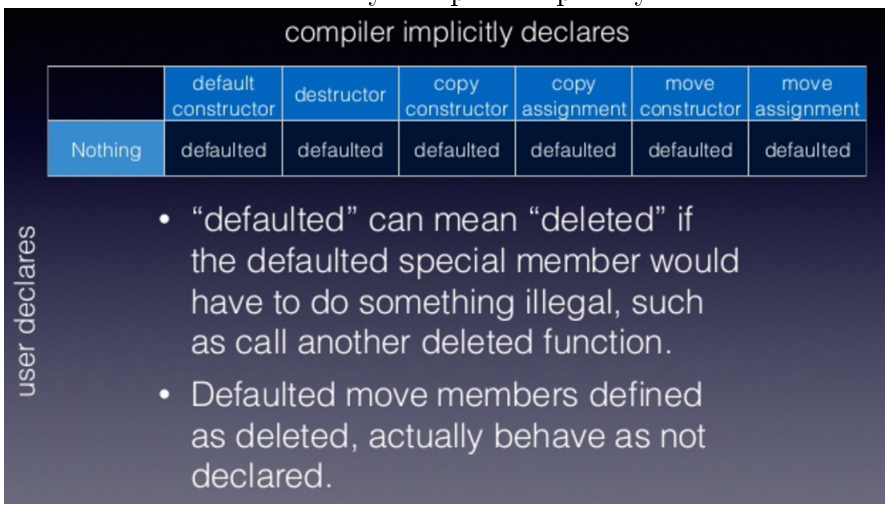
- For these special member functions, main operations can be:

  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?



- If you just define a class without any special member function, all six member function will be declared by compiler implicitly.



- Default can mean "deleted"

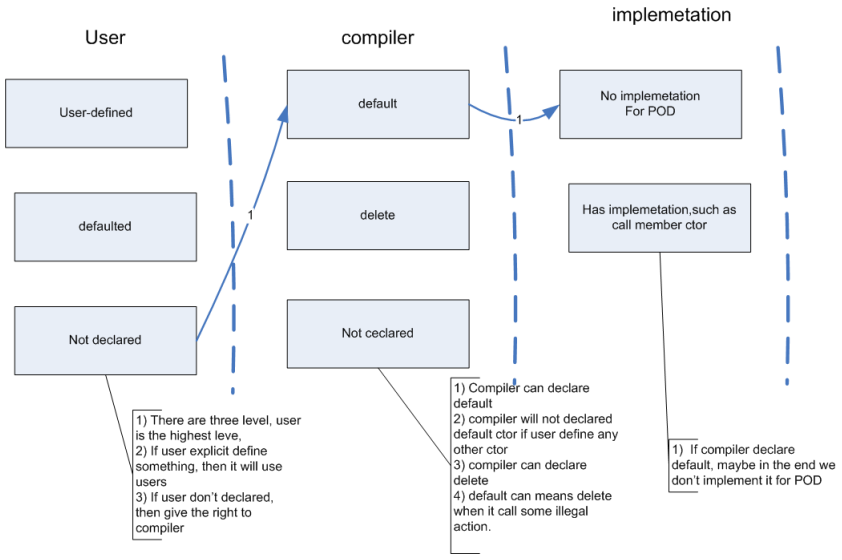
```
class A{
A() = delete;
};


class B{

A a;
};

int main()
{
B b;

}
```

```
main.cpp:16:7: note: âĂŸB::B()âĂŹ is implicitly deleted because the default defini
```

- 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:

```
struct noncopyable {
  noncopyable() {};
private:
  noncopyable(const noncopyable&);
  noncopyable& operator=(const noncopyable&);
};
```

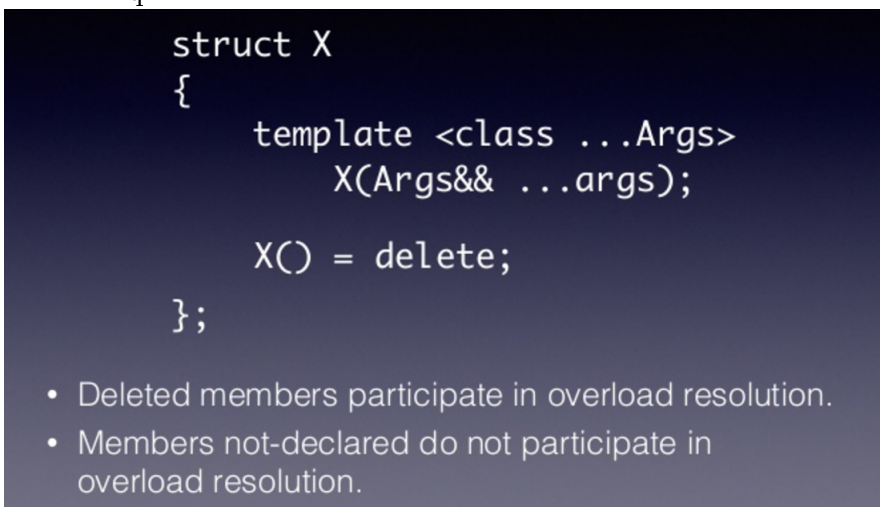
1. The copy constructor has to be declared privately to hide it, but because it's declared at all, automatic generation of the default constructor is prevented. You have to explicitly define the default constructor if you want one, even if it does nothing.
2. Even if the explicitly-defined default constructor 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:

```
struct noncopyable {
  noncopyable() =default;
  noncopyable(const noncopyable&) =delete;
  noncopyable& operator=(const noncopyable&) =delete;
};
```

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"



1.6.3.2 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;

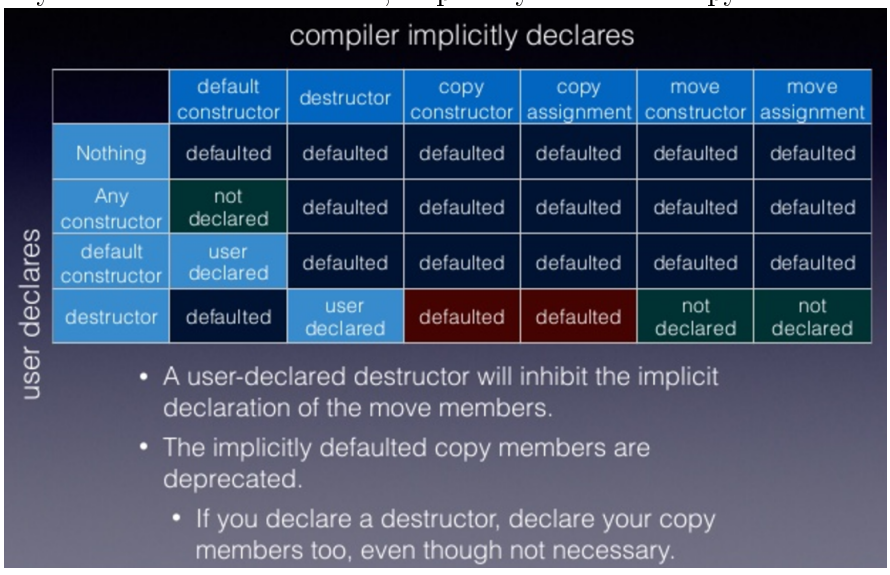
- The justification is that declaring a copy operation (conâĂŘstruction or assignment) indicates that the normal approach to copying an object(memberwise copy) isn't appropriate for the class, and compilers figure that if memberwise copy isn't appropriate for the copy operations, memberwise move probably isn't appropriate for the move operations.

- The same idea as the previous item, declaring a move operation (construction or assignment) in a class causes compilers to disable the copy operations.

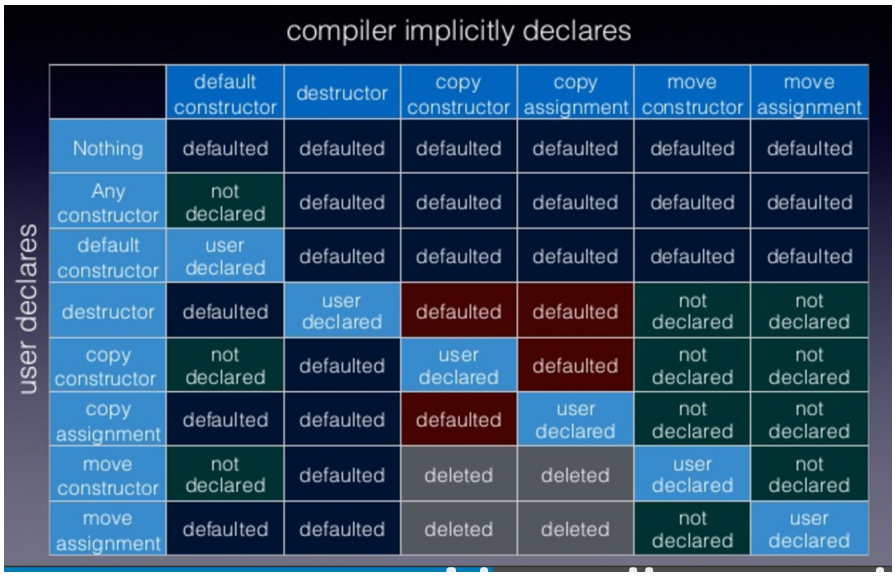
- The two copy operations are independent: declaring one doesn't prevent compilers from generating the other. The two move operations are not independent. If you declare either, that prevents compilers from generating the other.

- 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.



- A summary can be seen blow



1.6.3.3 initializer list

- **Always use initializer list instead of assignment inside ctor.**
- When to use member initalize list, you can see in this link:
  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

```
class A {
    int i;
public:
    A(int );
};

// Class B is derived from A
class B: A {
public:
    B(int );
};

B::B(int x):A(x) { //Initializer list must be used
    cout << "B's Constructor called";
}
```

  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/`

```
//method 1:
class(string &a, string &b): m_a(a),m_b(b){}
// just call string copy ctor,
//so you don't need string default ctor

//method 2:
class(string &a, string &b){    // two action.
m_a = a;  //call default constructor to build m_a
m_b = b; // then call assignment operator.
}
```

- 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**.

```
class Student{
string m_email;  //m_email will be init first, ingore order
string m_first_name;  // in the ctor initialization list.
Student(first_name) :m_first_name(first_name),
                m_email(m_first_anme+"@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.

```
template<class T>
class Array : private ArrayBase, public Container

typedef Array AIType;
public:
Array( size_t startingSize = 10 )
: Container( startingSize ),
ArrayBase( Container::GetType() ),
```

1.6.3.4 operator

- **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.

```
Vector2D& Vector2D::operator+=(const Vector2D& right)
{
    this->x += right.x;
    this->y += right.y;
    return *this;
}
```

```
Foo operator+(const Foo& lhs, const Foo& rhs)
{
  Foo result = lhs;
  result += rhs;
  return result;
}
```

- Use swap implement operator =. It's called "copy-and-swap idiom?" That is a sub conception under another important conception of "Rule of four(and half)"
- When you want to implement operator =, first implement friend function swap, then use swap to implement operator =.

```
class dumb_array{
..
private:
    std::size_t mSize;
    int* mArray;
};

friend void swap(dumb_array& first, dumb_array& second) // nothrow
{
    // enable ADL (not necessary in our case, but good practice)
   using std::swap;

   // by swapping the members of two objects,
   // the two objects are effectively swapped
   swap(first.mSize, second.mSize);
   swap(first.mArray, second.mArray);
}
```

- With above swap friend function. canonical, easy-to-write, always-correct, strong-guarantee, copy-and-swap assignment operator is often seen written this way

```
dumb_array& operator=(dumb_array other) {
    swap(*this, other);
    return *this;
}
```

1.6.3.5 Basic routines

- **Big five rule:** If you need customized copy ctor, assignment, and destructor, Then you also need move ctor and move assignment. Because it include memory

allocation inside your class, you need you own copy ctor to perform deep copy and use move ctor to "move resource".

- **Normally, ctor , destructor and assignment should be public. In inheritance context, all the base class desctructor should be virtual.**
- Avoid calling virtual functions in ctor and dtor. "C++ Coding Standards" item 49.
- What does a typical user defined move constructor do?

```
class x : public Base{
Member m_;
X(X&& x): Base(std::move(x)), m_(std::move(x.m_)){
x.set_to_resourceless_state();
}
}
```

- What does a defaulted move assignment do?

```
class x : public Base{
Member m_;
X& operator=(X&& x) {
Base::operator=(static_cast<Base&&>(x));
m_ = static_cast<Member&&>(x.m_);
return *this;
}
}
```

- 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:

```
class Widget {
Widget(Widget&& rhs)
: s(std::move(rhs.s))
{ ++moveCtorCalls; }
private:
static std::size_t moveCtorCalls;
std::string s;
};
```

- **If you define a specify 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.

```
class obj; //error
class* obj = new class(); //error
class arra[10] //error
template<class T>
class Array{
T t;
};

Array<class> a; //error
```

- If no special demand, you can declare you own default ctor and use system implicit generated one. you can use keyword default

```
class Empty{
 Empty() = default;
 // you don't need to give implementation of default ctor.
 Empty(int i) ;
}
```

- Make Constructors Protected to prohibit direct Instantiation. Make constructors Private to prohibit Derivation.

- Use default arguments to reduce the number of ctor.

```
class Brush{
Brush();
Brush(Color c);
Brush(Texture t);

Brush(Color c= Black, Texture t = Solid); // it will be better.
}
```

- 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.**

```
class Worker{
char* SSN;
Worker(const char*);
Worker(){SSN=nullptr); //bad smell.
}
```

- 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 resourced aquired by the class during its leftime Example 1: The class has a handle to a file, this needs to be closed when the object destructs; the destructor is the perfect location. Exempel 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 functon by value.
3. When a function returns an object by value.
4. When the compiler generates a temporary object.

```
1) Person r(p);    or  Person p = q;        // copy constructor
p = q;  // but not called in assignment; p has existed before

2) A value parameter is initialized from its argument.
fun(Person r)     fun(p);
// copy constructor, produce a new object r.
//just like Person r = p;

3) An object is returned by a function.
Person fun();
Person r = fun();
//here copy ctor is called twice.
// use -fno-elide-constructors in g++ produce two copy ctor
// or It will use Return value optimization.

4) class(string &a, string &b): m_a(a),m_b(b){}
//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.**

```
class & class::operator=(class &a){
  if(this == &a)
    return *this;  //avoid assignment
    ............   // assignment operation here.
    return *this ; // return *this reference.
}
```

- 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**

1.6.4 special member functions in inheritance

subsubsectionctor

- **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.

```
class base{
public:
  base();  //default ctor
  base(int b); //specify ctor
private:
  int b;
};

class subclass: public base{
subclass();  //default ctor
subclass(int s, int b); // specify ctor1
subclass(int s);      //specify ctor2
int s;
};

subclass::subclass(int s, int b): base(b){
  m_s = s;
}
////////////////////////////////////////
subclass sc1(2,3);
//call specify ctor1, then
// explicit call base specify ctor

subclass sc2(2);  // implicit call base default ctor
subclass sc3;   // implicit call base default ctor
// if base has not default ctor,
//sc2 and sc3 will produce error.
}
```

- 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.

```
//method 2:
DeriveClass::DeriveClass(int base_a, b){
  //it will call base_class default constructor.
   // in this case, base_a is not assigned at all
   Derive_b = b
}
//////////////////////
//method 3:
DeriveClass::DeriveClass(int base_a, b) {
base_a = a     //It can be thought as a bad design.
//You can't  access private base member data.
//base_a need to be public member data,
Derive_b = b
}
```

- Idea behind rules: Do best to make sure a obj can be built.

1.6.4.1 destructor

- For destructor:

  1. 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.
  2. If you define a subclass destructor, It will call base destructor automaticlly, you don't need to call it explicitly.
  3. 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.

```
template <class Arg, class Result>
struct unary_function{
  typedef Arg    argument_type;
  typedef Result result_type;
};

Illegal code that you can assume will never exist.
void f( std::unary_function* f ){
  delete f; // error, illegal
}
```

- 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-unwindling, when a obj is destructed, 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.

1.6.4.2 copy ctor

- copy ctor(assignment ctor) 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.

```
class Base{};
class Derived1 : Base{};
class Derived2 : Base{};
```

```
Derived1 d1;
Derived1 d2;
////////////////////
Base b = d1; //1)

Fun(Base b);
Fun(d1);  //2)

Base& Bref = d1;
Fun(Bref) //3)

Base* bp = new B(Bref); //4)

Base* bp1 = new Derived1();  //5) sibling slicing
base* bp2 = new Derived2();
*bp1 = *bp2;  // bp1 just copy Base part in Derived2.
//so bp1 now is MIXTURE of d1 and d2.
```

- Think a problem as below.

```
class Base{};
class Derived1 : Base{};
class Derived2 : Base{};
Derived d1;
Derived d2;

Base* Copy(Base& Bref){
//How to avoid slicing and make deep copy.
}

Base& Bref = d1
Base* d1p = Copy(Bref)

Base& Bref = d2
Base* d2p = Copy(Bref)
```

- Continue- Think this problem: error method

```
Base* Copy(Base& Bref){
Base* p = new Base(Bref)
//Slicing happen. bad
}
```

- Continue- Think this problem: TypeID method.

```
Base* Copy(Base& Bref){
//Use tyid and dynamic_cast
//involve a lot of if and switch about type.
//anytime if you use dynamic_cast and if,
//you can think about virtual function
}
```

- 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.

```
class Base{
  virutal Base* Clone() = 0;
};

class Derived1 : Base{
    virutal Derived1* Clone(){return new Derived1(*this);}
}

Base* Copy(Base& Bref){
Base* p = Bref.Clone();
}
```

- 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.

- copy ctor and copy initilization. Detail can be found "Is there a difference in C++ between copy initialization and direct initialization?"

```
class A{
explicit A(const &A);
}
A a1;
A a2(a1) //OK
A a3 = a1 //doesn't work
```

- 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-init

https://stackoverflow.com/questions/1051379/is-there-a-difference-between-copy-in
```

- Basic idea of all kinds of initialization.What's the difference?

```
widget w{x};  1)
widget w = x; 2)
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 âĂIJthe slow way to moveâĂİ 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));

- 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

```
B(A const&)
operator B(A&);
```

```
#include <iostream>
struct B;
struct A {
operator B();
};

struct B {
B() { }
B(A const&) { std::cout << "<direct> "; }
};

A::operator B() { std::cout << "<copy> "; return B(); }

int main() {
A a;
B b1(a);  // 1)
B b2 = a; // 2)
}
// output: <direct> <copy>
```

- For the number 2, This is why "S1 s11 = s; // ill-formed" fail in the below code snippet.

```
struct Intermediate {};

struct S
{
operator Intermediate() { return {}; }
operator int() { return 10; }
};

struct S1
{
S1(Intermediate) {}
};

S s;
Intermediate im1 = s; // OK
Intermediate im2 = {s}; // ill-formed
S1 s11 = s; // ill-formed
S1 s12 = {s}; // OK
```

```
// note: but brace initialization can use operator of conversion to int
int i1 = s; // OK
int i2 = {s}; // OK
```

- For the number 3, This is called âĂIJcopy list initialization.âĂİ It means the same as widget wx; except that explicit constructors cannot be used. ItâĂŹs guaranteed that only a single constructor is called. So number 3 is different with number 2

-

- Assignment operator and copy ctor in inheritance:

  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 syntaxt. For assignment operator, use two different methods depends on if base class declare its own assignment operator(). Source code is below:

```
DerivedClass::DerivedClass(const DerivedClass &dc): \
BaseClass(dc){...}  //init list syntax here.

DerivedClass & DerivedClass::operator=(const DerivedClass &dc){
BaseClass::operator=(dc);
// base class declare explicitly operator

( (BaseClass&) *this ) = dc
//base class no explicitly operator
// change *this to BaseClass reference,
// if you change to BaseClass, It will call copy ctor.
}
```

1.6.5 inheriantce

- OO just kick in when you use reference and pointer

```
        class A{
                public:
                virtual void fun(){ cout<<"A";}
        };
```

```
class B: public A{
        public:
        void fun(){cout<<"B";}
};

int main()
{
        B b;
        A a = b;
        a.fun(); //output A

        A& ra = b;
        ra.fun(); //output B
        A* pa = &b;
        pa->fun(); //output B
        (*pa).fun(); //output B

        A* pa1 = new B;
        (*pa1).fun(); //output B
}
```

1.6.6 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 objectâĂŹs 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.

```
//C version,
File* fp = fopen("/path/to/file");
// throw exception here, then resource leaking
fclose(fp);

//Java version
try {
    File file = new File("/path/to/file");
    // throw exception here, go to finally.
} finally {
    file.close();
}

//c++ version
fun{
fstream if("path/to/file")
```

```
if.getline
// you don't need to if.close().
}

//c++ smart version.
std::unique_ptr<FILE,           // <-- the wrapped raw pointer type: FILE*
                int(*)(FILE*)> // <-- the custom deleter type: fclose() prototype
myFile( fopen("myfile", "rb"), // <-- resource (FILE*) is returned by fopen()
        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 go out 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.

```
Int *p = new int;
string a   //a is ok, a will be destructed properly
             //due to the C++ unwind stack.
throw exception.
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.

```
unique_ptr<int> aupr (new int(100);
string a   //a is ok, a will be destructed properly
             //due to the C++ unwind stack.
throw exception.
// 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.

```
class module {
public:
    explicit module(std::wstring const& name)
    : handle { ::LoadLibrary(name.c_str()) } {}

    ~module{
        ::FreeLibrary();
    }
private:
    HMODULE handle;
};
```

- 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.

     ```
     class RAII {
     private:
         string m_str;
         vector<int> vc;
     };
     ```

  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.

     ```
     class RAII {
     private:
         string* m_str;
         vector<int*> vc;
     };
     ```

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

     ```
     class RAII {
     private:
         unique_ptr<string>  m_str;
         vector<unique_ptr<int> > vc;
     };
     ```

     ```
     class module {
     public:
         explicit module(std::wstring const& name)
         : handle { ::LoadLibrary(name.c_str()) } {}
     private:
     using module_handle = std::unique_ptr<void, decltype(&::FreeLibrary)>;
        module_handle handle;
     };
     ```

- Another question is ownership of resource:

1. For auto member resource: **1)Same life duration(RAII), 2)exclusive ownship 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 ownship to a single object, and copyable and efficient move**

```
Class RawPointer{
...........
RawPointer(const RawPointer& rhs){
pRes = new Resource( *(rhs.pRes));
}

RawPointer(RawPointer&& rhs){
pPes = rhs.pRes;
rhs.pRes = nullptr;
}
private:
Resource* pRes;
}
```

3. For uniqu_ptr; **1) Same life(RAII) 2)exclusive ownship 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 youself, 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 ownship, 3) copyable and moveable**. When you move a shared_ptr, origin one is set to nullptr and ref count doesn't increas. You still follow "rule of zero".

- **Conclusion, If you consider RAII and ownship 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 follow "Rule of Five" and support copyable and movable.
2. For special demand, for example Car class, people can **custom** its engine, and buy **two** at the same time. In this context, you 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"

```
class Car{
//follow "Rule of Five"
Engine *pEn;
~Car(){delete pEn} // assure RAII
}
```

3. For special context, It doesn't support object copy: for example 1) Person class in semantic; 2) other perform consideration, Class Bigint [30000];; 3) Other implementation constrain, such as iostream class. Under such context, you can use uniqu_ptr to manage the resource and implement uncopyable.

```
class Person{
uniqu_ptr<Resource> pRes;
}
```

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.

```
class Student{
shared_ptr<SchoolBus> pBus
}
```

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

1.7 Generic programming

1.7.1 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 is: Use typename, not class. At the same time. simple names, such as T.

```
template<typename T>
swap(T& a, T&b){
.........
}
```

1.7.1.1 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

```
template <typename T1,  typename T2>
class Pair{ }
Pair<double, int> pair1;

template <typename T1,  typename T2=int>
class Pair{ }
Pair<double> pair2;
```

- you can use Non-Type Argument in a template. But it will cause code bloat problem.

```
template <typename T, int n>
class ArrayTP{
T ar[n];
......
}

template <typename T, T n>
class Foo{
}
```

- type parameter can be another template calss. It's different with "template template parameter" which is introduced below.

```
template<typename T> // int T is here
class A

template<typename T> //A<int> is T here.
class B

B< A<int> > obj;
//Two T has no any relationship.
//you can give them any better description name
//just like vector< vector<int> >. That is a good example.
```

- Another usage 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.

```
template <typename T, class Cont>
class Stack {
public:
~Stack();
void push( const T & );
//...
private:
Cont s_;
};

Stack<int, List<int> > aStack1; // OK
Stack<double, List<int> > aStack2; // legal, not OK
Stack<std::string, Deque<char *> > aStack3; // error!
```

- We use template template parameter.

```
template <typename T, template <typename ELEM,
                                typename = std::allocator<ELEM>>
                                 class Cont = std::deque>
class Stack {
//...
};
//...
Stack<int> aStack1; // use default: Cont is Deque
Stack<std::string,std::list> aStack2; // Cont is List
```

1. Why you need std::allocator here? because std::deque is template calss with two template parameter.
2. you can pass type, non-type, or another template as template parameter.

1.7.1.2 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.

```
template<typename T, int n>
class ArrayTP...

ArrayTP<int, 100> stuff //implicit instantiation
// you have to declare obj stuff.

template ArrayTP<string, 100>;
//generate a specialized class even you don't declare an obj
```

2. explicit specification is define a different template impllementation for certain type. For example, For bool type, we can use optimize array storage method

```
template<typename T, int n>
class ArrayTP{...common implementation}

template<>
ArrayTP<bool, 100>{...specific implementation}
//There is empty<> after tempalte
//You should give you won implemenation.
```

3. implicit, explicit instantiation and explicit specification are all **specializations**. Because produce a real function definition that uses specific types, Not a 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.

```
template<> class Pair<int,int>{...}; //complete specialization
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 specializaiton, you have to redefine you own function body

```
Template<typename T>
void sortedArrary (T) {...};

template void sortedArray<Person>(Person) // instantiation
template<> void sortedArray<Person>(Person){ // specialization.
.... //give you own definition of fun body.
};
//Pay attention there is empty <> after template in specialization.
```

7. For function template, there is no partial specialization, So explicit specialization is complate 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. One problem with that is that 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.

- That is very important for you to understand three examples in the article "Dependent name lookup for C++ templates".

  1. Because there is template specialization, so derived template class will not look for name in base template class.

```
template <typename T> struct Base {
void f() {
std::cerr << "Base<T>::f\n";
}
};

template <typename T> struct Derived : Base<T> {
void g() {
std::cerr << "Derived<T>::g\n␣␣";
f(); //Will fail here
// this->f() or Base<T>::f() will resolve this problem.
}
};
```

  2. when a dependent name is encountered, it is assumed to be some sort of identifier (such as a function or variable name).When it is type name or template name, you need to explict specify it.

```
template <typename T> struct Base {
        typedef int MyType;
};

template <typename T> struct Derived : Base<T> {
        void g() {
                // A. error: âĂŸMyTypeâĂŹ was not declared in this sc
                // MyType k = 2;

                // B. error: need âĂŸtypenameâĂŹ before âĂŸBase<T>::M
                // âĂŸBase<T>âĂŹ is a dependent scope
                // Base<T>::MyType k = 2;

                // C. works!
                typename Base<T>::MyType k = 2;

                std::cerr << "Derived<T>::g␣—>␣" << k << "\n";
        }
}
```

  3. When compiler see T::foo_method, **he will not just specify T as Foo.** Think that your are compiler, you just delete the whole struct Foo, then try to understand it.

```
struct Foo {
template<typename U>
static void foo_method(){
}
};

template<typename T> void func(T* p) {
// A. error: expected primary-expression before âĂŸ>âĂŹ token
```

```
        // T::foo_method<T>();

        // B. works!
        T::template foo_method<T>();
        }
```

- C++ Coding standards 65 states customization of point. In order to understand it. You need to understand two some basic conception: "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.

1.7.1.3 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:

  1. A template has multiple types and only some of them need to be specialized. The result is a template parameterized on the remaining types.
  2. 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.
  3. 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 spcialization of member.

```
template <class T>
class Storage{
T m_value;
public:
Storage(T value){
m_value = value;
}

// Partial-specialization of Storage class for pointers
template <class T>
class Storage<T*>
{
T* m_value;
public:
Storage(T* value)
{
        m_value = new T(*value);   //To make deep copy
}
// Full specialization of constructor for type char*
```

```
template <>
Storage<char*>::Storage(char* value)
{
// Figure out how long the string in value is
int length = 0;
while (value[length] != '\0')
++length;
}
```

- 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 <> 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.

```
template <typename T1, typename T2>
class Pair{}  //general one

tempalte <typename T1>
class Pair<T1, T1>  // partial specializations

tempalte <typename T1>
class Pair<T1, int > // partial specializations
//-------------------------------------------

tempalte <typename T>
class Foo //general one

tempalte <typename T*>
class Foo //partial specializations.

tempalte <typename T>
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.

```
template <typename T>
class A{
public:
    void onlyForInts(T t){
        static_assert(std::is_same<T, int>::value, "Works only with ints!");
    }

protected:
    std::vector<T> myVector;
```

```
};

int main(){
    A<int> i;
    i.onlyForInts(1); // works !

    A<float> f;
    //f.onlyForInts(3.14f); // does not compile !
}
```

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. 1)You can complete(full) specialization of template class. 2)You also can complete(full) specialization of template function. 3)complete(full) specialization of member in template class. 4) Or partial template class.
2. Pay attention the specialization of declaration. it should be after the basic template.

1.7.1.4 template and friend

- you can have 1) friend class, 2)friend function or 3)friend template. at the same time, they can be bound or non-bound. If it's bound, you can find hoster 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.

```
template<typename T>
class HasFriend{
        friend void counts();
}
```

- Bound-template friend function. You need to define explicit specialization for the friends you plan to use. **Here, reports is not template function.**

```
        template<typename T>
        class HasFriend{
        friend void reports(HasFriend<T> &);
        }

        void reports(HasFriend<int> &hf){
        cout <<hf.item<<endl;
        }
        void reports(HasFriend<short> &hf){..}
        void reports(HasFriend<char> &hf){..}
        //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) template <typename T> void report(T &);

2)template<typename TT>
class HasFriend{
friend void reports<>(HasFriend<TT> &);
//it has <> after reports function name.
}

3)
template<typename T>
void reports(T &hf){
cout <<hf.item<<endl;
}
```

- non-bound friend template.

```
template<typename T>
class ManyFriend{
template<typename C, typename D>
friend void show(C& , D&);
};

template <typename C, typename D> void show2(c& c, D& d){
cout<<c.item<<d.item<<endl;
}

ManyFriend<int> mi;
ManyFriend<double>md;
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.

1.7.2 Type Inference

1.7.2.1 template type deduction

```
template<typename T>
void f(ParamType param);

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.

```
template<typename T>
void f(T& param); // param is a reference

int x = 27; // x is an int
const int cx = x; // cx is a const int
const int& rx = x; // rx is a reference to x as a const int

f(x); // T is int, param's type is int&
f(cx); // T is const int, param's type is const int&
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.

```
template<typename T>
void f(T&& param); // param is now a universal reference
int x = 27; // as before
const int cx = x; // as before
const int& rx = x; // as before

f(x); // x is lvalue, so T is int&, param's type is also int&
f(cx); //cx is lvalue, so T  and param's type are const int&
f(rx); //rx is lvalue, so T  and param's type are const int&
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 that, too. If it's volatile, also ignore that.

```
template<typename T>
void f(T param); // param is now passed by value

int x = 27; // as before
const int cx = x; // as before
const int& rx = x; // as before
f(x); // T's and param's types are both int
f(cx); // T's and param's types are again both int
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 fererences-to-const or pointers-to-const, we don't skip const.

```
template<typename T>
void f(T param); // param is now passed by value
```

```
const int* const p1 = &x;
f(p1)  //param is const int*, top const has been skipped.

const int*& rp = p1;
f(p1)  //param is const int*, top const is default for reference.
```

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 lvaue, T is lvalue reference. (keep const)
   (c) reference is ignored
   (d) for value-type ParamType, const and volatile are ignored.

1.7.2.2 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 use assignment. Here just think the auto is T in the template deduction.** The basic rules can be referred in type deduction in generic programming

```
auto x = 27; // (x is neither ptr nor reference)
const auto& rx = x; //  (rx is a non-universal ref.)
auto && ax = 27 // ()ax is forwarding ref)
```

- **When combine auto and brace init, it means std::initializer_list<T>**

```
auto x = { 11, 23, 9 }; // x's type is  std::initializer_list<int>

template<typename T> // template with parameter
void f(T x);
f({ 11, 23, 9 }); // error! can't deduce type for T

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.

```
auto createInitList(){
return { 1, 2, 3 }; // error: can't deduce type
} // for { 1, 2, 3 }

std::vector<int> v;
auto resetV =
[&v](const auto& newValue) { v = newValue; }; // C++14

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.

```
auto z1 {99}; // z1 is an initializer_list<int>
auto z2 = 99; // z2 is an int
```

```
auto d = {1.23};
// d is a std::initializer_list<double>
auto d = double{1.23};
auto d = 1.23
// Good -- d is a double, not a std::initializer_list.
```

- When auto use in forward reference. auto&& is a kind of forwarding reference.

```
auto&& uref1 = x; // x is int and lvalue,
// so uref1's type is int&
auto&& uref2 = cx; // cx is const int and lvalue,
// so uref2's type is const int&
auto&& uref3 = 27; // 27 is int and rvalue,
// so uref3's type is int&&
```

- 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.

```
std::vector<int> global_vec{1, 2, 3, 4};

template <typename T>
T get_vector(){
return global_vec;
}

template <typename T>
void foo(){
auto&& vec = get_vector<T>();
//only auto&& work here.
//auto, auto& , const auto&, const auto&& all failed.
auto i = std::begin(vec);
(*i)++;
std::cout << vec[0] << std::endl;
}

foo<std::vector<int>>();
std::cout << global_vec[0] << std::endl;
foo<std::vector<int>&>();
std::cout << global_vec[0] << std::endl;
```

1.7.2.3 decltype deduction

- decltype has two different deduct rules when facing different kinds of expression.

  1. Case1: 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,

```
vector<int> v; // decltype(v) is vector<int>

struct S {
int m_x;
};

int x;
const int cx = 42;
const int& crx = x;
const S* p = new S();

decltype(x) a;  // a is int
auto a = x;  // a is int

decltype(cx) b; // b is const int
auto b = cx;  //auto ignore const, b is int

decltype(crx) c;  // c is const int&.
auto c = crx; //ignore reference and const, c is int

// S::m_x is declared as int: m_x_type is int
// Note that p->m_x cannot be assigned to. It is effectively
// constant because p is a pointer to const. But decltype goes
// by the declared type, which is int.
decltype(p->m_x) d; // d is int
auto d = p->m_x; //auto ignore const, d is int
```

2. Case 2: 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.

```
struct S {
int m_x;
};

int x;
const int cx = 42;
const int& crx = x;
const S* p = new S();
//-----------------------------------------
// (x) has type int, and decltype adds references to lvalues.
// Therefore, x_with_parens_type is int&.

typedef decltype((x)) x_with_parens_type;

// 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&.
typedef decltype((cx)) cx_with_parens_type;

// The type of (crx) is const int&1, 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&.
typedef decltype((crx)) crx_with_parens_type;
```

```
// 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&.
typedef decltype((p->m_x)) m_x_with_parens_type;
```

3. for some complex expression examples

```
const S foo();
const int& foobar();
std::vector<int> vect = {42, 43};

// 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.
typedef decltype(foo()) foo_type;

// The type of foobar() is const int&1, 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&.
typedef decltype(foobar()) foobar_type;

// std::vector<int>'s operator[] is declared to have return
// type int&. Therefore, the type of the expression vect[0]
// is int&1. 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&.
decltype(vect[0]) first_element = vect[0];

// The type of the expresson is double, and the expression
// is an lvalue. Therefore, a reference is added, and
// cond_type is double&.
typedef decltype(d1 < d2 ? d1 : d2) cond_type;

// The type of the expresson 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.
typedef decltype(x < d2 ? x : d2) cond_type_mixed;
```

- **decltype can be used in template function return value.**

```
template<typename Container, typename Index> // works
auto authAndAccess(Container& c, Index i) -> decltype(c[i]){
// ->decltype(c.begin())
authenticateUser();
return c[i];
}

template<typename T, typename U>
auto eff(T t U u) -> decltype(T*U){
....
}
```

```
template<typename Container, typename Index> // C++14; works,
decltype(auto) authAndAccess(Container& c, Index i) {
authenticateUser();
return c[i];
}
//Pay attention to decltype(auto)
```

- make template function support universal reference.

```
template<typename Container, typename Index> // final c++14
decltype(auto) authAndAccess(Container&& c, Index i) {
authenticateUser();
return std::forward<Container>(c)[i];
}

template<typename Container, typename Index> // final c++11
auto authAndAccess(Container&& c, Index i)
-> decltype(std::forward<Container>(c)[i]){
authenticateUser();
return std::forward<Container>(c)[i];
}
```

- 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.

```
Widget w;
const Widget& cw = w;

auto myWidget1 = cw; // auto type deduction:
// myWidget1's type is Widget

decltype(auto) myWidget2 = cw;
// 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:

```
std::vector<int> vect;
assert(vect.empty());
typedef decltype(vect[0]) integer; //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:

```
template<typename R>
class SomeFunctor {
public:
typedef R result_type;
```

```
result_type operator()() {
return R();
}
SomeFunctor(){}
};

SomeFunctor<int> func;
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.

```
        class A {
        std::vector<std::pair<int, std::string>> array;
        decltype(array.begin()) iter;
        // you can't use auto here, because auto need init expression
        // You don't need init expression here.
        };
```

- 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:

```
template <typename C>
//Here no any variable in below expression.
decltype(std::declval<const C>().begin())
foo(const C& c){
return one iterator of c
}
```

- 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.

```
struct Default { int foo() const { return 1; } };

struct NonDefault{
NonDefault(const NonDefault&) { }
int foo() const { return 1; }
};

decltype(Default().foo()) n1 = 1;    // type of n1 is int
//decltype(NonDefault().foo()) n2 = n1; // error: no default ctor
decltype(std::declval<NonDefault>().foo()) n2 = n1;
//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.

```
struct foo {
double operator()(char, int);
};
//same
std::result_of<foo(char, int)>::type d1 = 10.0;
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"

1.7.2.4 check type

- **How to determine programmatically if an expression is rvalue or lvalue**

```
template <typename T>
constexpr bool is_lvalue(T&&) {
return std::is_lvalue_reference<T>{};
}
```

- There are three ways to view deducted type

  1. use IDE
  2. use Compiler

     ```
     template<typename T> // declaration only for TD;
     class TD; // TD == "Type Displayer"

     TD<decltype(x)> xType; // elicit errors containing
     TD<decltype(y)> yType; // x's and y's types
     ```

  3. use typeid and std::type_info::name

     ```
     std::cout << typeid(x).name() << '\n'; // display types for
     ```

1.7.2.5 summary

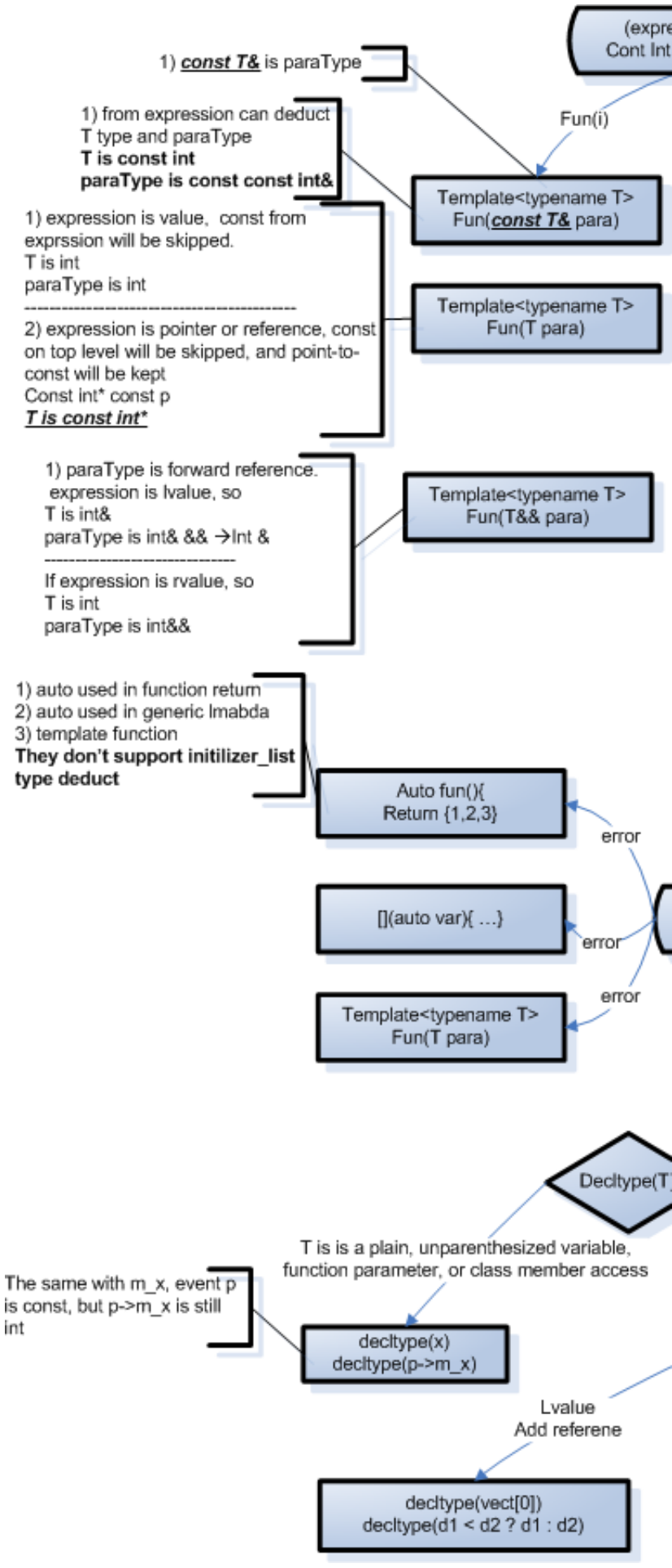
- auto, template T and decltype are three kinds of type deduction context. auto and template T are almost same, beside when we meet initilizaer_list. but decltype are quite different with the other two.

- When you use decltype(auto), auto tell compiler to automaticly type deduction, but please use decltype rules. They are most used when we deal with someexpression 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 topmost 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

- decltype(auto) is an alternate way of setting the type of a newly declared variable from its initializing expression. It uses the type deduction rules of decltype instead of those of auto.



- Basic idea can be illustrated below:

1.7.3 type traits and policy

1.7.3.1 implementation

- **The most common way of implementation is template class specification.**

  1. most common usage of trait is member constants is_integer<T>::value
  2. 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

```
template <typename T>
struct is_swapable {
static const bool value = false;
};

template <>
struct is_swapable<short> {
static const bool value = true;
};

template <>
struct is_swapable<int> {
static const bool value = true;
};

assert( is_swapable<T>::value && "Cannot swap this type" );
```

  3. A good article is "A simple introduction to type traits"

- Another good way is to build new trait based on existing trait.

```
template <typename T>
struct is_swapable {
static const bool value = std::is_integral< T >::value && sizeof( T ) >= 2
};
```

- Another type trait implementation method, use sizeof and static_cast, these two can be done at compile time. That is to test if a class derived from another class.

```
template<typename D, typename B>
class IsDerivedFromHelper
{
class No { };
class Yes { No no[3]; };

static Yes Test( B* );
static No Test( ... );
public:
enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };

};
```

```
template <class C, class P>
bool IsDerivedFrom() {
return IsDerivedFromHelper<C, P>::Is;
}
```

- 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. use test function to initialized constexpr **value**
  4. There are two examples, one is judge if T has member function. Another it so test if T is function.

```
template <class T> struct IsFunction
{

//This is pointer to array of function.
// int (*p)[1]  pointer to int [1]
// int *p[1], that is array of pointer to int.
//Pay attention to these difference.
template <typename C>
static constexpr bool test(U (*)[1] ){
return true;
}

template <typename C>
static constexpr bool test(...){
return false;
}

// int is used to give the precedence!
//use 0 as pretend pointer.
static constexpr bool value = test<T>(0);
};
```

```
template <class T> struct hasSerialize
{
// We test if the type has serialize using decltype and declval.
template <typename C> static constexpr decltype(std::declval<C>().serializ
{
// We can return values, thanks to constexpr instead of playing with size
return true;
}

template <typename C> static constexpr bool test(...)
{
return false;
}

// int is used to give the precedence!
```

```
static constexpr bool value = test<T>(int());
};
```

- last kind of implementation is use category. An example can be found in iterator.

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};

template < ... > // template params elided
class deque {
class iterator {
public:
typedef random_access_iterator_tag iterator_category;
...
};

template<typename IterT>
struct iterator_traits {
typedef typename IterT::iterator_category iterator_category;
...
};

template<typename T> // partial template specialization
struct iterator_traits<T*> // for built-in pointer types
{
typedef random_access_iterator_tag iterator_category;
...
};
```

- How to deal with member funciton pointer? It may look a bit confusing, because if the type of &A::fun was spelled out explicitly, it would have to be written int (A::*)() const &. However, in the template's case, the type int () const & is "hidden" behind the name U, so the pointer to member is then just U A::*. It's similar to how type names can be used to simplify the syntax of normal function pointers:

```
int foo(char, double) { return 42; }

using Fun = int (char, double);
Fun *p = &foo;
// instead of:
int (*q)(char, double) = &foo;
```

```
struct A {
int fun() const&;
};

template<class T, class U>
struct PM_traits<U T::*> {
using member_type = U;
};
```

```
int main() {
using T = PM_traits<decltype(&A::fun)>::member_type; // T is int() const&
}
```

- Based on previous knowledge, we can implement

```
template< class T >
struct is_member_function_pointer_helper : std::false_type {};

template< class T, class U>
struct is_member_function_pointer_helper<T U::*> : std::is_function<T> {}

template< class T >
struct is_member_function_pointer : is_member_function_pointer_helper< std

class A {
public:
void member() { }
};

int main()
{
// fails at compile time if A::member is a data member and not a function
static_assert(std::is_member_function_pointer<decltype(&A::member)>::value
"A::member is not a member function.");
}
```

- summary:

  1. template specification: is_integer
  2. use basic trait: is_swap
  3. use template paramter deduciton : std::is_function and is_member_function, is_array, is_class(T::*)
  4. define trait in existing class : iterator_trait.
  5. overload template function, deduct return or function parameter based on template parameter T's expression: has_serialize, isIsFunction

1.7.3.2 usage

- This is where type traits come in handy. 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.
- Now question is how can you know if a type T has a member function? 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"

1.7.4 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.

```
swap(i1, i2);
swap(f1, f2);
// don't need to specify int, compiler
//can extract type information from i1

// compiler will produce two swap functions bodies.
```

- Why we need a overload template function, Because not all type support the same operator. For example, in your template function, you use = operator, but when you use array for this template function type, compiler will report error.

```
template <typename T>
void swap(T a[], T b[], int n)

template <typename T>
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.

- "Why Not Specialize Function Templates?" You can see it in the ref.

- "Why Argument Dependent Lookup doesn't work with function template dynamic_pointer_cast". See it in the ref

- "C++ template function taking template class as parameter" See in ref.

1.7.4.1 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 math, 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.s

- 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.

```
int i = 2;

f(const int& j); //#1
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 in side this specialized argument template.

```
template<typename T>
f(T t);  //#1

template< >f<int*>(int* t) #2

template<typename T>
f(T* t); //#3

template<> f<int> f( int* t) //#4

int * p;
f(p) // #4>#3>#2>#1
```

- For above example, If you omit type inside <> after function name f, It will depends on location.

```
template<typename T>
f(T t);  //#1

template< >f<>(int* t) //#2 is specialization of #1
////////////////////////////////////////
template<typename T>
f(T* t); //#3
```

```
template< >f<>(int* t) //#4 put here is specialization of #3


int *p;
f( p ); //If you put specialization at #2, it will call #3, That is wrong.
//If you put specialization at #4, it will call #4, that is right.
```

- You can tell compiler that you prefer template function over overload one

```
template<typename T>
void f(T t);  //#1

void f(int t) //#2

f<>(2) //tell compiler use #1, not #2.
```

1.7.4.2 template function specification

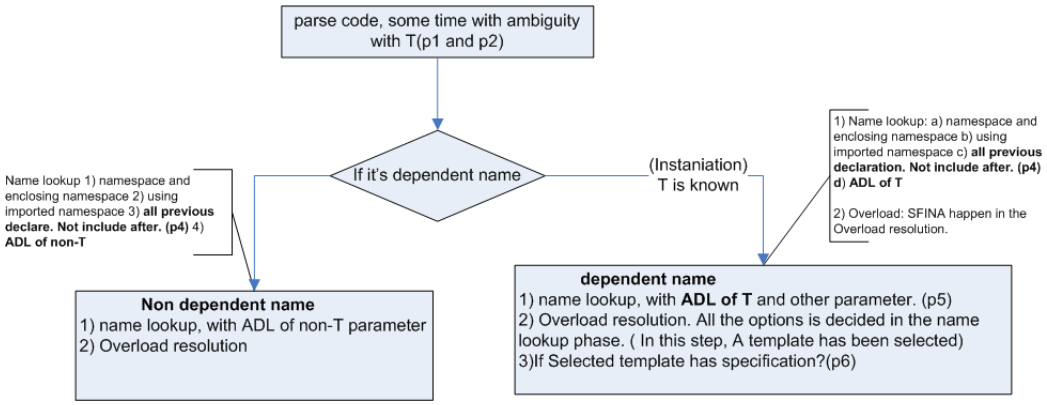
- Explicit Specialization: Because for char *, you can't use > , but use strcmp. So you need to build explicit Specialization Version. The prototype and definition of an explicit spcialization should be preceded by template<> and should mention the specialized type by name.

```
Template<typename T>
      void sortedArrary(T) {...};

template<> void sortedArray<const char *>(){...}
```

- Some explicit specification syntax.

```
template<typename T>
void foo(T param);

//syntax 1: not a specialization, it is an overload
void foo(int param);

//syntax 2: ill-formed, not recommend
void foo<int>(int param);

//syntax 3: normal explicit specialization, specify T directly
template <> void foo<int>(int param);

//syntax4: same as above, but use template argument deduction
template <> void foo(int param);

//syntax5: same as above, but use template argument deduction
template void foo(int param);
```

- 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!




```
template <typename T> void func(T param) {} // definition

template void func<int>(int param); // explicit instantiation.

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.**

- template function only support full specification.

1.7.4.3 summary

- First name look up, then overload, last instanitiation.

- 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.**

- Summary

- p1 and p2,

```
template <typename T> struct Base {
    typedef int MyType;
};

template <typename T> struct Derived : Base<T> {
    void g() {
        // A. error: âĂŸMyTypeâĂŹ was not declared in this scope
        // MyType k = 2;

        // B. error: need âĂŸtypenameâĂŹ before âĂŸBase<T>::MyTypeâĂŹ because
        // âĂŸBase<T>âĂŹ is a dependent scope
        // Base<T>::MyType k = 2;
```

```
        // C. works!
        typename Base<T>::MyType k = 2;

        std::cerr << "Derived<T>::g ---> " << k << "\n";
    }
};
```

```
struct Foo {
    template<typename U>
    static void foo_method()
    {
    }
};

template<typename T> void func(T* p) {
    // A. error: expected primary-expression before âĂŸ>âĂŹ token
    // T::foo_method<T>();

    // B. works!
    T::template foo_method<T>();
}
```

- p3

```
template <typename T> struct Base {
    void f() {
        std::cerr << "Base<T>::f\n";
    }
};

template <typename T> struct Derived : Base<T> {
    void g() {
        std::cerr << "Derived<T>::g\n  ";
        f(); //error happen here.
        this->f() will resolve problem.
    }
};
```

- 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**

```
namespace framework  // library 1{
  template <typename T>
  void f(T) { puts("master"); }

  template <typename T>
  void process(T v) { f(v); }
}

namespace boost      // library 2{
  template <typename T>
  struct optional {};
```

```
}

namespace framework  {
  template <typename T>
  void f(boost::optional<T>) { puts("optional<T>"); }

  inline
  void f(boost::optional<bool>) { puts("optional<bool>"); }
}

int main()            // our program logic{
  int i = 0;
  boost::optional<int>  oi;
  boost::optional<bool> ob;

  framework::process(i); //output three master
  framework::process(oi);
  framework::process(ob);
}
```

- p5, In instaniation, with help of ADL, boost namespace is searched and two other functions are added into overload options.

```
namespace framework  // library 1
{
  template <typename T>
  void f(T) { puts("master"); }

  template <typename T>
  void process(T v) { f(v); }
}

namespace boost      // library 2
{
  template <typename T>
  struct optional {};
}

namespace boost      // some glue between 1 and 2
{
  template <typename T>
  void f(optional<T>) { puts("optional<T>"); }

  inline
  void f(optional<bool>) { puts("optional<bool>"); }
}

int main()            // our program logic
{
  int i = 0;
  boost::optional<int>  oi;
  boost::optional<bool> ob;

  framework::process(i);
  framework::process(oi);
  framework::process(ob);
```

```
}
```

- p6 template specification happen in the end and after overload.

```
namespace framework  // library 1
{
  template <typename T>
  void f(T) { puts("master"); }

  template <typename T>
  void process(T v) { f(v); }
}

namespace boost      // library 2
{
  template <typename T>
  struct optional {};
}

namespace framework  // some glue between 1 and 2
{
  template <>
  void f<boost::optional<bool>>(boost::optional<bool>)
  { puts("optional<bool>"); }
}

int main()           // our program logic
{
  int i = 0;
  boost::optional<bool> ob;

  framework::process(i);
  framework::process(ob);
}
```

- Another interesting example

```
template<typename T>
void f(T i){
        cout<<"template "<<i<<endl;
}

//template<> void f<short>(short s);

void g(void){
        int s = 5; //output tempalte
        short s = 5; //uncomment above specification, it will call specification
                     //comment above specification decalaration., it will report er
                     //"Specification after initialzation.
        f(s);
}

template<> void f<short>(short s){
        cout<<"specification" <<s <<endl;
}
```

```
void f(short s){
        cout<<"short " <<s<<endl;
}

int main(){
        g();
        return 0;
}
```

1.7.5 template and inheritance

- Non-Template class Derived from Template Base class specification

```
        class u8toU16 : public std::codecvt<wchar_t, char, std::mbstate_t>{
        };
```

- Template class Deriving from the non-template class

```
        class Base

        template<typename T>
        class Derived: public Base
```

-

- template class can be used as base class and used as a component class. So we can inehrite from a template class

  1. A basic example

```
        template< typename Type>
        class SpecialStack: public Stack<Type>{
        Array<Type> array;
        }
```

  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.

```
        template<typename T>
        class codecvt_base{
        public:
        enum result {ok, partial, error, noconv};
        };
```

  3. You also can extending the derived class

```
        template< typename T>
        class Base

        template<typename T, typename U>
        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 bloating.

```
template<typename T> // size-independent base class for
class SquareMatrixBase { // square matrices
protected:
...
void invert(std::size_t matrixSize); // invert matrix of the given si
...
};

template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
private:
using SquareMatrixBase<T>::invert; // make base class version of inve
// visible in this class; see Items 33
// and 43
public:
...
void invert() { invert(n); } // make inline call to base class
}; // version of invert
```

5. Specialising the base class

```
template< typename T>
class Base

template<typename T>
class Derived: public Base<int>
```

- Parameterised inheritance

1. Basic example

```
template<typename T>
class Derived: public T
```

2. **Two famous idioms are Mixin and Policy, can be found below sections. In this chapter, we only focuse syntaxtic 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.

```
template <class T>
struct OpNewCreator{
static T* Create(){
return new T;
}
};
```

```
template <class T>
struct MallocCreator{
static T* Create(){
void* buf = std::malloc(sizeof(T));
if (!buf) return 0;
return new(buf) T;
}
};

// Library code
template <class CreationPolicy>
class WidgetManager : public CreationPolicy
{
...
};

// Application code
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.**

1.7.6 template common usage and idiom

- A common used idiom in std
  1. concept
  2. traits
  3. tag dispatching
  4. adaptors
  5. type generators
  6. object generators
  7. policy classes

1.7.6.1 member function templates

-

- Use member function templates to accept âĂIJall compatible types.âĂİ detail can be found in effective c++ item 45

```
template<typename T>
class SmartPtr {
public:
template<typename U> // member template
SmartPtr(const SmartPtr<U>& other); // for a âĂİgeneralized
... // copy constructorâĂİ
};
```

- 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.

```
template<typename T>
class Rational {
public:
...
  friend const Rational operator*(const Rational& lhs, const Rational& rhs){
  return Rational(lhs.numerator() * rhs.numerator(),
  lhs.denominator() * rhs.denominator());
  }
};
```

- 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 âĂIJnormalâĂİ C++ program, because the traits approach is TMP. **Remember, traits enable compile-time if...else computations on types by overload**

1.7.6.2 policy

-

- All above technologies are part of generic programming. A good and more example about generic programming examples can be found in "Modern C++ desing" a book. The first chapter introduce policy pattern. The second chapter introduce a lot of ways to manage Type information at compile time.

- The Policy class can be thought a optional way to used as bridge pattern and multi-inheritance. You can move this part to OOP chapter if you have time later.

```
        template <typename OutputPolicy, typename LanguagePolicy>
        class HelloWorld : private OutputPolicy, private LanguagePolicy{
        using OutputPolicy::print;
        using LanguagePolicy::message;

        public:
        // Behaviour method
        void run() const{
        // Two policy methods
        print(message());
        }
        };

        class OutputPolicyWriteToCout{
        protected:
        template<typename MessageType>
        void print(MessageType const &message) const{
```

```
        std::cout << message << std::endl;
        }
        };

        class LanguagePolicyEnglish{
        protected:
        std::string message() const{
        return "Hello, World!";
        }
        };

        class LanguagePolicyGerman{
        protected:
        std::string message() const{
        return "Hallo Welt!";
        }
        };

        int main(){
        /* Example 1 */
        typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyEnglish> HelloW

        HelloWorldEnglish hello_world;
        hello_world.run(); // prints "Hello, World!"

        /* Example 2 * Does the same, but uses another language policy */
        typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyGerman> HelloWor

        HelloWorldGerman hello_world2;
        hello_world2.run(); // prints "Hallo Welt!"
        }
```

item Templates Parameterised by a Derived Class (Static polymorphism)

```
        template <class T>
        struct Base{
        void interface()    {
        // ...
        static_cast<T*>(this)->implementation();
        // ...
        }

        static void static_func()    {
        // ...
        T::static_sub_func();
        // ...
        }
        };

        struct Derived : Base<Derived>{
        void implementation();
        static void static_sub_func();
        };
```

- A easy way of "call back" is function pointer, but it can't be used inline. so There is efficiency problem.

- Based on function pointer, you can use inheritance, It's runtime polymorphism, and has some runtime overhead.
- 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.

```
// char_traits::eq
#include <iostream>    // std::cout
#include <string>      // std::basic_string, std::char_traits
#include <cctype>      // std::tolower
#include <cstddef>     // std::size_t

// traits with case-insensitive eq:
struct custom_traits: std::char_traits<char> {
static bool eq (char c, char d) {
return std::tolower(c)==std::tolower(d);
}

// some (non-conforming) implementations of basic_string::find call this
static const char* find (const char* s, std::size_t n, char c)
{ while( n-- && (!eq(*s,c)) ) ++s; return s; }
};

int main ()
{
std::basic_string<char,custom_traits> str ("Test");
std::cout << "T found at position " << str.find('t') << '\n';
return 0;
}
```

- 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.

1.7.6.3 tag dispatch

- A famous usage of tag dispatch is iterator

1. Pay attention to iterator_traits. Why do we use it here? It's also a template. With specialization, it also support common pointer type.
2. Client side may or may not be template. For advance, it should be template, because it need template parameter Inputiterator.
3. 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.

```
namespace std {
struct input_iterator_tag { };
struct bidirectional_iterator_tag { };
struct random_access_iterator_tag { };

namespace detail {
template <class InputIterator, class Distance>
```

```
void advance_dispatch(InputIterator& i, Distance n, input_iterator_tag) {
while (n--) ++i;
}

template <class BidirectionalIterator, class Distance>
void advance_dispatch(BidirectionalIterator& i, Distance n,
bidirectional_iterator_tag) {
if (n >= 0)
while (n--) ++i;
else
while (n++) --i;
}

template <class RandomAccessIterator, class Distance>
void advance_dispatch(RandomAccessIterator& i, Distance n,
random_access_iterator_tag) {
i += n;
}
}

template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n) {
typename iterator_traits<InputIterator>::iterator_category category;
detail::advance_dispatch(i, n, category);
}
}
```

- The second implementation of tag dispatch is use integral_constant

  1. **std::true_type use inegral_constant(type generator)to define a new type. is_arithmetic derived form true_type or false_type**

```
std::integral_constant<bool, true>

template< class T >
struct is_arithmetic : std::integral_constant<bool,
std::is_integral<T>::value ||
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 inheriated from ture_type, Then use overload resolving.

```
template <typename T>
int foo_impl(T value, std::true_type) {
// Implementation for arithmetic values
}

template <typename T>
double foo_impl(T value, std::false_type) {
// Implementation for non-arithmetic values
}

template <typename T>
```

```
auto foo(T value) {
// Calls the correct implementation function, which 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
return foo_impl(value, std::is_arithmetic<T>{});
}
```

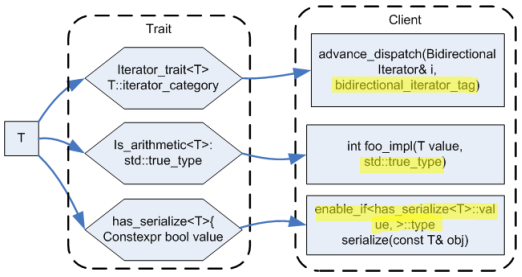
- The third method: build your own trait, it define a value
  1. build you own trait has_serialize, it includes a bool memeber ::value
  2. You can use decltype, declval and constexpr to build this trait.
  3. From this trait, it return value, not a type. So you need use enable_if template in the client code. change the value to a type, then use the type to overload resolving.
  4. For false, enable_if is empty define.

```
template <bool, typename T = void>
struct enable_if
{};

template <typename T>
struct enable_if<true, T> {
typedef T type;
};
```

```
template <class T> struct has_serialize
{
// We test if the type has serialize using decltype and declval.
template <typename C>
static constexpr decltype(std::declval<C>().serialize(), bool())
test(int /* unused */)
{
// We can return values, thanks to constexpr instead of playing with sizeo
return true;
}
template <typename C>
static constexpr bool test(...)
{
return false;
}
// int is used to give the precedence!
static constexpr bool value = test<T>(int());
};


//client side
template <class T>
enable_if<has_serialize<T>::value, std::string>::type
serialize(const T& obj)
{
return obj.serialize();
}
// Contra-SFINAE to avoid ambiguity
template <class T>
```

```
        enable_if<!has_serialize<T>::value, std::string>::type // Watch out for th
        serialize(const T& obj)
        {
        return to_string(obj);
        }
```

- summary:



1.7.6.4 mixin

- method 1, use Mixin

1. Mixin refers to the following idiom - a template class that is parameterised on its base class
2. 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;
3. if you have a MyTask value, you can build a helper function to build Mixin class

```
template<typename Task>
LogTask<Task> getLogTask(Task const& task){
        //Call the below explicit ctor in LogTask class
        return LogTask<Task>(task);
}

getLogTask(task).Execute //with log funcitons
```

4. 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"

```
class MyTask{
        public:

        void Execute(){
        std::cout << "...This is where the task is executed..." << std::endl;
        }
};

template< class T >
class LoggingTask : public T{
        public:

        explicit LoggingTask(T const& t) : T(t) {}
```

```
        void Execute(){
        std::cout << "LOG: The task is starting -" <<std::endl;
        T::Execute();
        std::cout << "LOG: The task has completed -" <<  std::endl;
        }
};

LoggingTask<  MyTask  > t;
t.Execute();
```

- method 2, use CRTP

  1. Use base class extend and reuse function(Execute_imp)
  2. Expose Execute interface to client by base class
  3. Client can directly use MyTask.

```
template< class Task >
class LoggingTask {
        public:
        void Execute(){
                std::cout << "LOG: The task is starting -" <<std::endl;
                static_cast<Task const&>(*this).Execute_imp();
                std::cout << "LOG: The task has completed -" <<  std::endl;
        }
};

class MyTask:public LoggingTask<MyTask> {
        public:
        void Execute_imp(){
        std::cout << "...This is where the task is executed..." << std::endl;
        }
};

MyTask t;
t.Execute();
```

- method 3, template Method

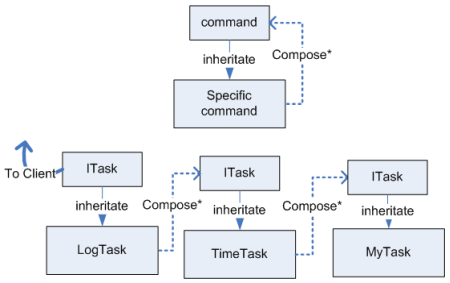
  1. No template, just use run time polymorphism
  2. Base class define a basic flow of action, some part of action is override in the Child class, That is Template Method pattern in design pattern.

```
class BaseLoggingTask
{
public:
virtual void Execute()
{
std::cout << "LOG: The task is starting -" << std::endl;
OnExecute();
std::cout << "LOG: The task has completed -" <<std::endl;
}
```

```
virtual void OnExecute() = 0;
};

// Concrete Task implementation that reuses the logging code of the BaseLoggingTa
class MyTask : public BaseLoggingTask
{
public:
virtual void OnExecute()
{
std::cout << "...This is where the task is executed..." << std::endl;
}

};
```

- 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. Base picture.




```
class LoggingTask : public ITask
{
ITask* task_;
public:
LoggingTask( ITask* task ) : task_( task )
{
}

~LoggingTask()
{
delete task_;
}

virtual void Execute()
{
        std::cout << "LOG: The task is starting - " << GetName().c_str() << std::
        if( task_ ) task_->Execute();
        std::cout << "LOG: The task has completed - " << GetName().c_str() << std
}
};



class MyTask : public ITask
{
```

```
public:
virtual void Execute()
{
std::cout << "...This is where the task is executed..." << std::endl;
}

virtual std::string GetName()
{
return "My task name";
}
};

ITask* t = new LoggingTask(
new TimingTask(
new MyTask() ) );
t->Execute();
```

- method 5, mixin many mixin

  1.

```
template< class T >
class LoggingTask : public T
{
public:
void Execute()
{
std::cout << "LOG: The task is starting - " << GetName().c_str() << std::endl;
T::Execute();
std::cout << "LOG: The task has completed - " << GetName().c_str() << std::endl;
}
};

template< class T >
class TimingTask : public T
{
Timer timer_;
public:
void Execute()
{
timer_.Reset();
T::Execute();
double t = timer_.GetElapsedTimeSecs();
std::cout << "Task Duration: " << t << " seconds" << std::endl;
}
};

class MyTask
{
public:
void Execute()
{
std::cout << "...This is where the task is executed..." << std::endl;
}
};
```

```
edef LoggingTask<
TimingTask<
MyTask > > Task;
Task t4;
t4.Execute();
```

- method 6, policy base

  1.

```
class coutLog{
        void beginLog(){
        cout<<"std::cout<<"LOG: The task is starting - "<<endl
        }

        void endLog(){
        std::cout<<"LOG: The task has completed - "
        }
}


template<typename LogPolicy, typename TimePolicy>
class Task: private LogPolicy, TimePolicy{

void Execute()
{
        beginLog();
        beginTime();
        std::cout<<"...This is where the task is executed..."<<std::endl;
        endTime();
        endLog();

}
```

- summary:(insert a picture)

  1. Policy use can be used when you log and time funciton change a lot
  2. Mixin and command can be compose the multi-actions
  3. template is more effeicent than inheritance.
  4. Command pattern can also be thought as composite pattern.
  5. Mixin will not change current class, but CRTP change the current class interface.

1.7.7 type erasure and concept

1.7.7.1 function

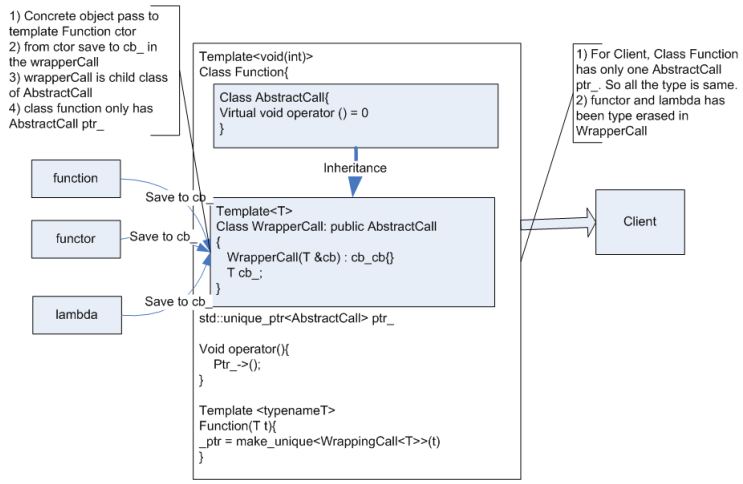
- function template support copyable and callable

```
        #include <cmath>
        #include <functional>
        #include <iostream>
```

```
#include <map>

double add(double a, double b){
        return a + b;
}

struct Sub{
        double operator()(double a, double b){
                return a - b;
        }
};

double multThree(double a, double b, double c){
        return a * b * c;
}

int main(){

        using namespace std::placeholders;

        std::cout << std::endl;

        std::map<const char , std::function<double(double, double)>> dispT
// (1)                                                                    |
                {'+', add },                                            |
// (2)                                                                    |
                {'-', Sub() },                                          |
// (3)                                                                    |
                {'*', std::bind(multThree, 1, _1, _2) },                |
// (4)                                                                    |
                {'/',[](double a, double b){ return a / b; }}};
// (5)                                                                    |

        std::cout << "3.5 + 4.5 = " << dispTable['+'](3.5, 4.5) << std::en
        std::cout << "3.5 - 4.5 = " << dispTable['-'](3.5, 4.5) << std::en
        std::cout << "3.5 * 4.5 = " << dispTable['*'](3.5, 4.5) << std::en
        std::cout << "3.5 / 4.5 = " << dispTable['/'](3.5, 4.5) << std::en

        std::cout << std::endl;

}
```

In this way you can write a type-erased wrapper for any Whatever whatever âĂŤ callbacks, counters, output streams, input generators.

- A good article is "What is Type Erasure" A detail implementation can be found in `http://blog.bitfoc.us/p/525`



- A picture

```
struct Plus1 {
int call(int x) const { return x+1; }
};

struct AbstractCallback {
virtual int call(int) const = 0;
virtual ~AbstractCallback() = default;
};

template<class T>
struct WrappingCallback : AbstractCallback {
T cb_;
explicit WrappingCallback(T &&cb) : cb_(std::move(cb)) {}
int call(int x) const override { return cb_(x); }
};

struct Callback {
std::unique_ptr<AbstractCallback> ptr_;

template<class T>
Callback(T t) {
ptr_ = std::make_unique<WrappingCallback<T>>(std::move(t));
}
int operator()(int x) const {
return ptr_->call(x);
}
};

int run_twice(const Callback& callback) {
return callback(1) + callback(1);
}

int main() {
int y = run_twice([](int x) { return x+1; });
assert(y == 4);
}
```

1.7.7.2 type erasure

- Baisc type erasure tech.

- When you implement type erasure (in C++ or even in Go), it always starts with making a list of the things you want to be able to do with your type-erased object âĂŤ call it, destroy it, copy it, and so on. Don Norman, in the book The Design of Everyday Things (1988), calls this a list of affordances. 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.
    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.
    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.
    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.

1.7.7.3 enable_if

- Note SFINAE at work here. 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.

```
template <typename T>
void do_stuff(typename enable_if<std::is_integral<T>::value, T>::type &t) {
  // an implementation for integral types (int, char, unsigned, etc.)
}

template <typename T>
```

```
void do_stuff(typename enable_if<std::is_class<T>::value, T>::type &t) {
  // an implementation for class types
}
```

- This technology has been used widely inside of vector.

```
template <typename T>
class vector {
    vector(size_type n, const T val);

    template <class InputIterator>
    vector(InputIterator first, InputIterator last);
    ...
}

template <class _InputIterator>
vector(_InputIterator __first,
       typename enable_if<__is_input_iterator<_InputIterator>::value &&
                          !__is_forward_iterator<_InputIterator>::value &&
                          ... more conditions ...
                          _InputIterator>::type __last);
```

- A good article is "SFINAE and enable_if"
- Another example about how to use type information is to use Int2Type. example is in the Modern C++ Design C, second chapter. You can see another example about Type2Type. The same function can be finished by enable_if.

```
DoSome(T*, Int2Type<true>){
         ...
         }

DoSome(T*, Int2Type<false>){
         }

         Do(T* pObj){
                          DoSome(pObj, Int2Type<IsPolymorphic>());
                          }
```

- **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.**

use tag

```
template <typename T>
auto get_value(T t, std::true_type) {
    return *t;
}

template <typename T>
auto get_value(T t, std::false_type) {
    return t;
}
```

```
template <typename T>
auto get_value(T t) {
    return get_value(t, std::is_pointer<T>{});
}
```

use SFINAE

```
template <typename T, std::enable_if_t<std::is_pointer_v<T>, int> = 0>
auto get_value(T t) {
    return *t;
}

template <typename T, typename std::enable_if_t<!std::is_pointer_v<T>, int> = 0>
auto get_value(T t) {
    return t;
}
```

use constexpr if:

```
template <typename T>
auto get_value(T t)
{
    if constexpr (std::is_pointer_v<T>) {
        return *t;
    } else {
        return t;
    }
}
```

1.7.8 Template example

1.7.8.1 A basic example

- .h file. They can be compiled wit -std=c++11

```
template<class X, class Y>
class my2D{
        X x;   Y y;
public:
        my2D(X x, Y y){
                this->x = x;    this->y = y;
        }
        X getX();  Y getY();
};

//create a new template class my2D_child which inherits from
//this template with new member function length(),
template<class X, class Y>
class my2D_child :my2D<X,Y>
{
        public:
        my2D_child(X x, Y y) : my2D<X,Y>::my2D(x,y){}
    auto length()->decltype(my2D<X,Y>::x+my2D<X,Y>::y);
};

//write a template specilization of my2D_child for string
template<>
```

```
class my2D_child<std::string, std::string> :
                        my2D<std::string, std::string>{
public:
        my2D_child(std::string x, std::string y) :
                my2D<std::string,std::string>::my2D(x,y){}
    void length(){
                std::cout<< getX()<<getY() <<std::endl;}
};
```

- .cpp file.

```
//file my2D.cpp
#include "my2D.h"
#include <iostream>
using namespace std;

//impletment the function getX in file my2D.cpp
template int my2D<int,int>::getX();
template float my2D<float,float>::getX();
template<class X, class Y>
X my2D<X,Y>::getX(){
        return x;
};

//implement this length function of my2D_child to return x*x+y*y
template int my2D_child<int, int>::length();

template<class X, class Y>
//C++14 has new syntax auto(decltype).
auto my2D_child<X,Y>::length()->decltype(my2D<X,Y>::x+
                                my2D<X,Y>::y){
        return my2D<X,Y>::getX()*my2D<X,Y>::getX()
                        +my2D<X,Y>::getY()*my2D<X,Y>::getY();
};

int main(){
//to use this my2D template for cases (X=int(3), Y=int(4))
//                              , (X=float(5.5), Y=float(6.6),
my2D<int, int> my_int_2d(3,4);
my2D<float, float> my_float_2d(5.5, 6.6);

cout<<my_int_2d.getX()<<endl;

my2D_child<int, int> my_int_child_2d(3,4);
cout<<my_int_child_2d.length()<<endl;

//print an output of my2D_child's length for
//(X="I need ",Y="Time to finish")
my2D_child<string, string> ms_2d_child("I need", "time");
ms_2d_child.length();
return 0;
}
```

1.8 Exception and error

1.8.1 End application

- You can call abort(), exit(), quick_exit and _exit to end you problem anytime. 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)

```
void myProgramIsTerminating1(void){
  cout<<"exit function 1"<<endl;
}

int main(int argc, char**argv){
  atexit (myProgramIsTerminating1);
  //abort();
//if you uncomment it, myProgramIsTerm will not be called.
  return 0;
}
```

- 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.

```
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.

```
int main() {
    try {
        // your stuff
    }
    catch( ... ) { // catch all exceptions.
        return EXIT_FAILURE;
    }
}
```

- according to previous main, if you want to end applicaiton in the other fun, you need to throw a exception, then leave it un-handle or rethrow it until it reach main, in this way, stack unwinding will make sure all the destrcutor will be called. **Don't use exit, it's C-style function and will not perform any stack unwinding**

```
try{
  fun(){
    throw end_exception();
 }
}
catch(end_exception& ex){
//do something here
throw;
}
// or just skip the whole catch, end_exception will reach
//main function all clean all the local object,
//when it return from main,
// it will do the same work as exit();
```

1.8.2 Bug and assert

1.8.2.1 Use assert to find bugs early

- **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 nunormal" will cause error in other place, then It's a little difficult to trace back source.

```
fun(char* p){
    assert(p!=nullptr);
    ..... // a lot of codes here
    strcpy(p) //error happen here,
    // but You don't know the source is the beginning of fun.

    fun1(p); It will cause error in other place.
}
```

- 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.

```
#include <cassert>
assert(I<5 && "I is more than 5");
//string literal is always true;
//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 you 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 a 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.

```
FILE *f = fopen("hr.dat".....);
if(f==nullptr){  // It's an exception, so don't use assert here.
   return -1;
   throw runtime_error();
}

assert(age>=0); //Here use assert.
fprintf(f, %d, age);
```

1.8.2.2 Trace

- you can implement trace, such as it in the MFC

```
#if defined NDEBUG
    #define TRACE( format, ... )
#else
    #define TRACE( format, ... )    printf( "%s::%s(%d)"
            format, __FILE__, __FUNCTION__,  __LINE__, __VA_ARGS__ )
#endif
```

1.8.3 Handling exceptions

1.8.3.1 errno in C

- There are three common error methods:

```
//1) global error code, C language method
errno() and strerror()
//1) status in object , C++ method
cin.fail()

//2) return value by parameter or return value
//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."

```
FILE * pFile = fopen ("unexist.ent","rb");
  if (pFile==NULL)
    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),

```
#include <cerrno>
errno = 0
// set it zero before call any math library function.
double not_a_number = std::log(-1.0);
  if (errno == EDOM) {
        std::cout << "log(-1) failed: " <<
                     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

```
#include <math.h>        /* math_errhandling */
#include <errno.h>       /* errno, EDOM */
#include <fenv.h>
/* feclearexcept, fetestexcept, FE_ALL_EXCEPT, FE_INVALID */

#pragma STDC FENV_ACCESS on
  errno = 0;
  if (math_errhandling & MATH_ERREXCEPT)
          feclearexcept(FE_ALL_EXCEPT);

  sqrt (-1);
  if (math_errhandling & MATH_ERRNO) {
    if (errno==EDOM) printf("errno set to EDOM\n");
  }

  if (math_errhandling  &MATH_ERREXCEPT) {
    if (fetestexcept(FE_INVALID)) printf("FE_INVALID raised\n");
  }
```

1.8.3.2 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, 1) it make progamme 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.

```
Class my_ex :public std::exception
{
   const char* what()
   {return "my_ex reason is here"}
}
```

- 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.

```
catch(my_base_ex &me){
......
throw;
//not use throw me,
//because maybe me is child class of my_base_ex;
}
```

- 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 unhandle exception will can terminate in the end.

1.8.4 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 appliation can be continue after you reslove 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 responds.** logical error should use assert or unit tesst 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!**

```
fun(int* pi, int j){
  assert(j< 10);
  if (pi == nullptr)
     //don't throw it, It's logical error.
     //you need to check caller of fun
     //why pass a nullptr into fun

  if(file.open())
    throw runtime_error();
     // file maybe deleted by other. It's not your logic error
     //but runtime error.
}
```

- 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

```
atomic_fun(){
try{
  //complex operation.
  .......
  }
 catch(...){
    cout<<"I can't do it"<<endl;
    throw;
 }
```

- 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.

1.9 STL

- A Good reference book is "effective STL".
- 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_differene, 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.

- STL is a good example of generic programming, there is specific book about topic of generic programming, if you have time in the future, you can look at it.

- 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_ea

```
struct Sum{
    Sum(): sum{0} { }
    void operator()(int n) { sum += n; }
    int sum;
};
//////////////////////
 std::vector<int> nums{3, 4, 2, 8, 15, 267};

 std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });
 // g++ -std=c++11 compile lambda syntax.

 Sum s = std::for_each(nums.begin(), nums.end(), Sum());
 // It return a Sum struct.
 cout<<s.sum<<endl;
```

1.9.1 Container

1.9.1.1 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.

```
int j = 10;
vector<int> vc;
vc.push_back(j); // not j, but copy of j to vc

int i = vc.pop_back(); //
i = -99 // when you modify i, not effect on vc
```

- Even copy in, copy out. vector is better than array in C language, see demo code below:

```
Foo farray[50]; //default ctor has been called 50 times.
Foo *parray = new Foo[50]; Same, ctor will be called 50 times.

vector<Foo> fvc; // no default ctor has been called.
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

```
getMutexFor(v);
// do something on v.
releaseMutexFor(v);
```

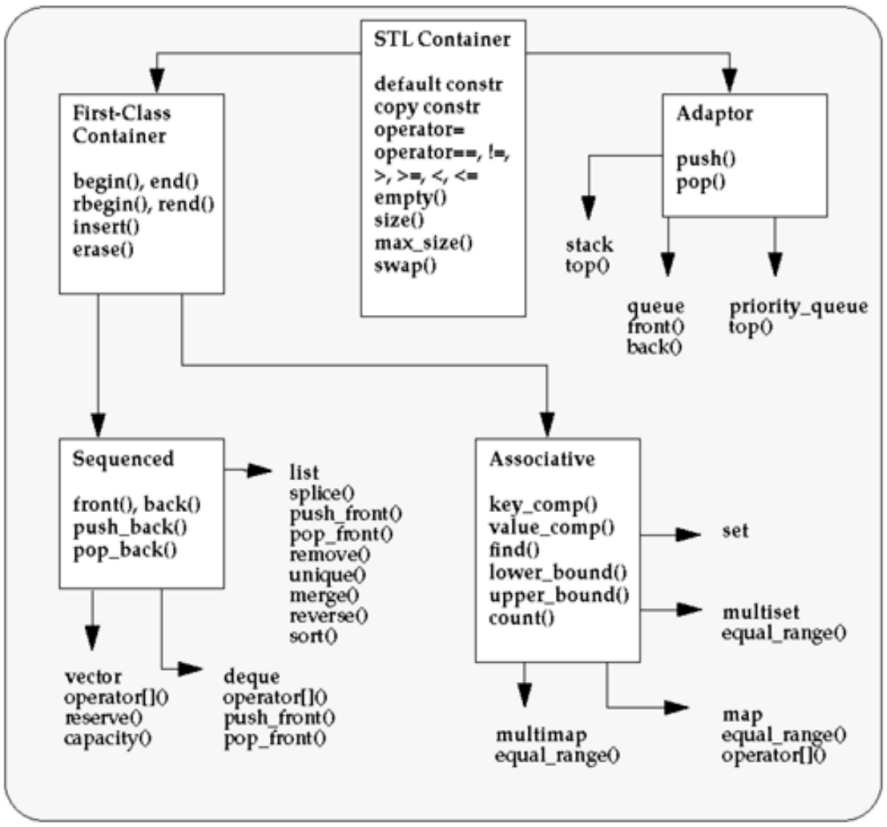
- There are three kinds of for-loop container syntax;

```
for(auto it = con.begin(), itend != con.end()
          it !=itend ; ++it){
   foo(*it);
}

for_each(con.begin(), con.end(), [](Element& e){
   foo(e);
});

for(auto & e : con){
   foo(e);
}
```

1.9.1.2 Basic classifications

- 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 function.) By now deque support (4+6+4+3 = 17 functons)**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 not 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.

1.9.1.3 contiguous or node

- Another classification of containter : 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.)

```
vecArr.insert ( it + 2, 1 , 200 );

// Reinitialize the invalidated iterator to the begining.
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

```
        auto it = std::find(vecArr.begin(), vecArr.end(), 5);
    if(it != vecArr.end())
        vecArr.erase(it);
        it = vecArr.erase(it);

    // Now iterator 'it' is invalidated because it
    //  still points to old location, which has been deleted.
for(; it != vecArr.end(); it++)//Unpredicted Behavior
 std::cout<<(*it)<<"  "; //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

1.9.1.4 Search in Container

- Distinguish among count, find, binary_search, lower_bound, upper_bound, and equal_range.

What You Want to Know	Algorithm to Use		Member Function to Use	
	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 | 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 >=1, (find one or more match, can replace find() or count() function.) .

```
 std::sort (v.begin(), v.end());
 // 10 10 10 20 20 20 30 30
auto p =std::equal_range (v.begin(), v.end(), 20);
for ( auto i = p.first; i != p.second; ++i )
        std::cout << i->name << ' ';

 //or code below:
 if(distance(p.first , p.second)  >=1){
      cout<<"found match"<<endl;
 }
```

1.9.1.5 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.**

```
v1.clear();
copy(v2.begin(), v2.end(), back_inserter(v1) );

//prefer to use below three range memeber function
v1.insert(v1.end(), v2.begin(),v2.end() );  //better than copy
v1.assign(v2.begin(), v2.end() )
// if want to replace all value in v1.
vector<int> v1( v2.begin(), v2.end() );
// build from sractch.
v1.erase( v1.begin(), v1.begin()+5);
//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.

```
//don't use front_inserter() here,
//because vector don't support push_front()
```

```
copy(v2.begin() , v2.end(),
          inserter(v, v.begin() ) );

// a better method is here.
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:

```
container::container(inputIterator1 , inputIterator2);
container::insert(insertPosition , inputIt1 , inputIt2);
container::erase(inputIterator1 , inputIterator2);
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.

1.9.1.6 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 it's own remove or remove_if.
3. For associative container, use its erase member function.

```
vect.erase(remove(vect.begin(), vect.end(),1963), vect.end());
list.remove(1963);
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 sen in effective STL item 9;

```
bool badValue(int x);
vc.erase(remove(vc.begin(), vc.end(), badValue),
                                                vc.end());

lsit.remove_if(badValue);
for(auto i = map.begin();i!=map.end(); /* no ++i here*/) {
        if(badVaue(*i)) map.erase(i++);
        // or i = map.erase(i);
        else  ++i;
}
```

- To do something inside the loop (in addition to erasing objects); You can't use remove, just write a loop,

```
bool badValue(int x);
for(vect<int>::iterator i = vect.begin();i!=vect.end(); ){
        if(badVaue(*i)) {
                i = vect.erase(i);
                ...do something else
        }
        else  ++i;
}
```

1.9.1.7 value_type in container

- value_type in container.

```
vector<uint> vecs;
cout << sizeof(vecs.value_type)  //error usage
cout<<sizeof(vector<uint>::value_type);
//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:

```
template<typename TMap>
void insert_default_pair(TMap& map)
{
    map.emplace(typename TMap::key_type(),
                        typename TMap::mapped_type());
```

```
}
share
```

- 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.

```
template <typename T>
class TSContainer {
private:
        T container;
public:
        void push(typename T::value_type& item){
                container.push_back(item);
        }
```

- summary:

  1. complex type
  2. change a lot, in above example
  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)it will save a lot of typing.
 typedef vector< pair<int, string> > ComVec;
ComVec::value_type aaa;

//2)int to float, but below code don't need change at all.
typedef vector< pair<float, string> > ComVec;

//3) inside of funciton.
typedef std::vector< std::pair<int,std::string> > Record_t;
// typedef is your good friend, reuse it below
Record_t k1;

int find_it(std::string value, Record_t const& stuff){
  auto fit = std::find_if(stuff.begin(), stuff.end(),
             [value](Record_t::value_type const& vt) -> bool
                  { return vt.second == value; });
```

1.9.1.8 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.

```
std::vector<int> myvector;
for (int i=0; i<100; i++) myvector.push_back(i);

std::cout << (int) myvector.size() << '\n'; // 100
std::cout  << (int) myvector.capacity() << '\n';  //128
std::cout << (int) myvector.max_size() << '\n';  //1073741823
```

- Four confused conceptions:

```
capacity() //how many CAN hold
size()   //how many are in NOW
resize(n) // forces the container to change to n,
//if n <size(),  object in the end will be lost
// if n>size(),  default ctor of element will be called.
// after resize(n), size will return n.
// but resize don't change capacity

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)**

```
vector<class> v1;
v1.reserve(1000); v1.size(); // only 5

vector<class>(v1).swap(v1);
//1) vector<class>(v1) copy ctor create a temp vector
//2) temp just copy real object, so it's capacity (maybe 8)is small.
//3) temp.swap(v1) then temp has 1000 space,  v1 capacity is 8 now
//4) in the end of statment, temp destory.
string(s).swap(s) // the same idea behind.
```

- Worthwhile note about vector: It will avoid allocate new space many times.

```
std::vector<int> v
for(){  ...
 v.clear
}  //good smell.

for(){vector<int> v  ...} //bad smell
```

- resize and reserve difference.

```
vector<Foo> vcFoo;
vcFoo.reserve(10);
vcFoo.resize(10); // will call Foo ctor 10 times
vcFoo[2] = foo; // will call Foo ctor and assignment 1 times
// if no reserve or resize. vcFoo[2] = foo will fail.
```

1.9.1.9 Usages Tips

- **When you use STL Container, you should realize that typedef are your best friends.**

```
typedef vector<Foo, SpecialAllocator<Foo> > FooContainer;
typedef FooContainer::iterator FooIt;

FooContainer fc1; // make you programming more clearly,
FooContainer fc2;  // you should use typedef more
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

  1. you can't change key in map because it's const default.

```
map.begin()->fist = 10 //compile error
```

  2. For set or map, you can modify non-key part.

```
iterator i = set.find(employee);
if( i != set.end())
   i->setTitle("manager")  // it's also ok or
   const_cast<Employee &> (*i).setTitle("manager")
   // you must change it to a reference, then you can modify it.
   // if you just use  const_cast<Employee >
   // this is not right. it will create a temporary obj
   //and then modify a temporary obj.
```

  3. If you want to modify the key part in set or map.

```
// 1) find the one
iterator i = se.find(employee);
//2) create temp one.
if(i != se.end())
Employee e(*i);
```

```
//3) modify
e.setKey("new key")
//4) delete the old one and keep ;
 se.erase(i++);
 //5) insert new one
 se.insert(i,e);
```

- Item19 effective STL, Understand the difference between equality and equivalence in associative Containers.

  1. 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.
  2. 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:
  3. It will cause container.find and generic algorithm find has different result

```
set<string, case_insensitive_compare> ss;
//ss has "AA";
ss.find("aa"); //return true;
find(ss.begin(),ss.end(),"aa") //return false
//find algorithm use operator == .
```

  4. It will lead to item 21 in effective STL. Always have comparison functions return false for equal values. (strict weak ordering)
  5. It will lead to item 20 in effective STL. For associative container of pointers, You need to specify comaprsion 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)

```
struct stringLess: binary_fucntion<const string* ,
                          const string * , bool>{
   bool operator()(const string* s1 ,
                             const string * s2){
        return *s1<*s2;
}
}
 set<*string, stringLess> ss;
```

1.9.1.10 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:

```
string(const char* s);
 string(const char*, size_type n);
 string(const string& str, size_type pos, size_type n = npos)
 ......
```

- 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:

```
Old_c(const char* p); string str;
Old_c(str.c_str());  // legacy C function read a string

// legacy C write
size_t Old_c(char *pArray, size_t arraySize);

vector<char> vc(maxNumChars);
// 1) create a vector whose size is maxNumChars
size_t charsWritten = Old_c(&vc[0], vc.size());
// 2) have fillString write into vc
string s(vc.begin(), vc.begin()+charsWritten);
// 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()	
max_size() resize()	maximum number of characters 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()	Searches through a string for the first character that is not any element of a specified string.
find_first_of()	
find_last_not_of()	
find_last_of()	.

- find function in the string return position. Vector doesn't have find member function, you can use find function in algorithm category.

```
size_t found;
found=str.find("haystack");
// or found=str.find('.');
if (found!=std::string::npos)
    std::cout << "'haystack' also found at: " << found << '\n';

vector<int> myints = { 10, 20, 30, 40 };
auto  it = std::find (myints.begin(), myints.end(), 30);
if(it!=myints.end())
   cout<< "find 30"<<endl;
```

begin(),end()	Returns an iterator addressing the first element in the string.
rbegin(), rend()	

clear()	Erases all elements of a string.
insert(), erase()	Inserts an element or a number of elements or a range of elements into the string at a specified position.
append(), push_back()	Adds characters to the end of a string.
assign()	Assigns new character values to the contents of a string.
replace()	

at()	a reference to the element at a specified location
c_str(), data()	
compare()	
copy()	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()	Returns a copy of the allocator object used to construct the string.
swap()	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

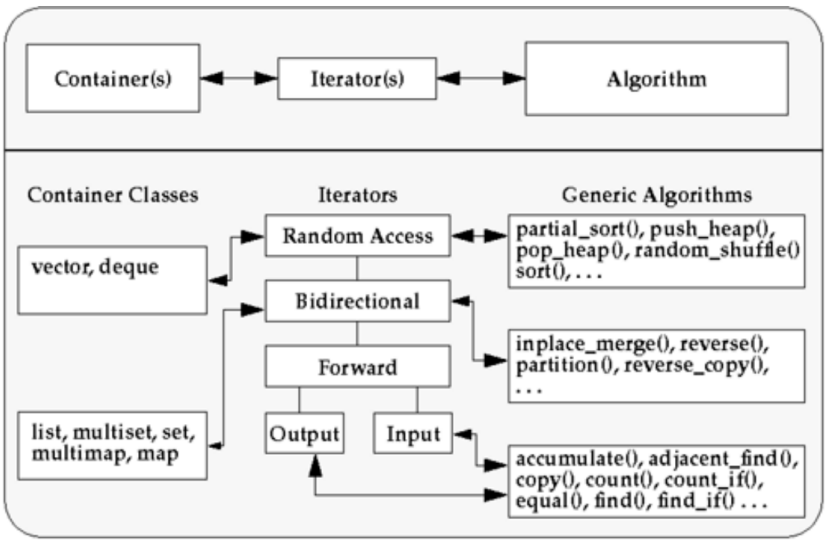
```
std::string str ("Please, replace the vowels with asterisks.");
  std::size_t found = str.find_first_of("aeiou");
  while (found!=std::string::npos)
  {
    str[found]='*';
    found=str.find_first_of("aeiou",found+1);
  }
```

1.9.2 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.

<table>
<tr><th colspan="4">category</th><th>properties</th><th>valid expressions</th></tr>
<tr><td colspan="4">all categories</td><td>copy-constructible, copy-assignable and destructible</td><td>X b(a);
b = a;</td></tr>
<tr><td colspan="4">all categories</td><td>Can be incremented</td><td>++a
a++</td></tr>
<tr><td rowspan="8">Random Access</td><td rowspan="5">Bidirectional</td><td rowspan="4">Forward</td><td>Input</td><td>Supports equality/inequality comparisons</td><td>a == b
a != b</td></tr>
<tr><td></td><td>Can be dereferenced as an rvalue</td><td>*a
a->m</td></tr>
<tr><td>Output</td><td>Can be dereferenced as an lvalue
(only for mutable iterator types)</td><td>*a = t
*a++ = t</td></tr>
<tr><td rowspan="2"></td><td>default-constructible</td><td>X a;
X()</td></tr>
<tr><td>Multi-pass: neither dereferencing nor incrementing affects dereferenceability</td><td>{ b=a; *a++;
*b; }</td></tr>
<tr><td colspan="2"></td><td>Can be decremented</td><td>--a
a--
*a--</td></tr>
<tr><td colspan="3" rowspan="4"></td><td>Supports arithmetic operators + and –</td><td>a + n
n + a
a - n
a - b</td></tr>
<tr><td>Supports inequality comparisons (<, >, <= and >=) between iterators</td><td>a < b
a > b
a <= b
a >= b</td></tr>
<tr><td>Supports compound assignment operations += and –=</td><td>a += n
a -= n</td></tr>
<tr><td>Supports offset dereference operator ([])</td><td>a[n]</td></tr>
</table>

- Basic container iterator is below:



- In each container, iterator of container is implemented by itself. There is no iterator class inheritance hierarchical structure. I means that in vector, there is vectIter: Iterator (vectIter inheriate 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:

```
template<type T>
vector{
....
typedef T* iterator;
};

vector<int>::iterator ip; //use it outside.
// ++ and -- is done by pointer automatically.
```

and All the member function of vector know the iterator very well, inside, it will use T* directly,

```
push_back(T x){
*end++ = x;
}  // you can see there is no iterator
   // 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.**

```
template <class InputIterator, class OutputIterator>
  OutputIterator copy (InputIterator first, InputIterator last,
            OutputIterator result);
```

- Common four iterator errors.

1. Valid values: Is the iterator dereferenceable? For example, writing "*e.end()" is always a programming error.
2. Valid lifetimes: Is the iterator still valid when it's being used? Or has it been invalidated by some operation since we obtained it?
3. 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.)

```
int main(){
vector<Date> e;
copy( istream_iterator<Date>( cin ), istream_iterator<Date>(), back_inserter( e )
vector<Date>::iterator first = find( e.begin(), e.end(), "01/01/95" );
vector<Date>::iterator last = find( e.begin(), e.end(), "12/31/95" );
*last = "12/30/95";
copy( first, last, ostream_iterator<Date>( cout, "\n" ) );
e.insert( --e.end(), TodaysDate() );
copy( first, last, ostream_iterator<Date>( cout, "\n" ) );
}
```

- You can inheriate you 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().

```
std::list<int>::iterator first = mylist.begin();
std::list<int>::iterator last = mylist.end();

std::advance(first, 3) //
std::cout << std::distance(first,last)
```

- istream_iterator and istreambuf_iterator topic:

```
std::ostream_iterator<int> out_it (std::cout,", ");
std::copy ( myvector.begin(), myvector.end(), out_it );

ifstream inputFile("aa.txt");
string fileData((istreambuf_iterator<char>(inputFile)) ,
 istreambuf_iterator<char>());
// It will just read all the character.(including white character)
// I don't need format data,

ifstream inputFile("aa.dat");
list<int> data((istreambuf_iterator<char>(inputFile)) ,
 istreambuf_iterator<char>());
// It wil  read format data, and use white space as delimiter.
```

1.9.2.1 Insert iterator

- Three common iterator generator: inserter(), back_inserter(), front_inserter(); They will produces three insert_iterator: insert_iterator, back_insert_iterator and 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.

```
list<State> res2( src.size() );
copy( src.begin(), src.end(),
    res2.begin() ); // often works...
```

- Another way is to use back_insert_iterator.

```
std::vector<int> foo;
for (int i=1; i<=5; i++)
  { foo.push_back(i); bar.push_back(i*10); }
  std::copy (bar.begin(),bar.end(),back_inserter(foo));
  //foo support push_back

  std::deque<int> foo;
  // deque or list
  for (int i=1; i<=5; i++)
  { foo.push_back(i); bar.push_back(i*10); }
  std::copy (bar.begin(),bar.end(),front_inserter(foo));
  //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 âĂŤ *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.

1.9.2.2 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:

```
std::find(foo.begin(), foo.end(), L'a');
                    // Returns an iterator pointing
                       // to the first a character.
std::find(foo.rbegin(), foo.rend(), L'a').base()-1;
                       //Returns an iterator
                        // pointing to the last A.
std::find(foo.end(), foo.begin(), L'a');
                     //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.

```
reverse_iterator ri =
    find(foo.rbegin(), foo.rend(), L'a');
foo.insert(ri.base()); // you can use ri.base() directly.
                                  // when you want to insert.
foo.erase((++ri).base()); //you can't use ri.base() directly.
                                    //when you erase what you want.
      ri
1 2 3 4 5
        i = ri.base()
```

1.9.3 Algorithms

1.9.3.1 Basic

- Effective STL item 43, **Prefer algorithm calls to hand-written loops**.

  1. Handing writing is prone to bug

     ```
     for( size_t i = 0; i < max; ++i )
     d.insert( current++, data[i] + 41 ); // do you see the bug?
     ```

  2. You can fix it bu with careful design

     ```
     for( size_t i = 0; i < max; ++i ) {
     current = d.insert( current, data[i] + 41 ); // be careful to
     keep current validâĂę
     ++current; // âĂę then increment
     it when it's_safe
     }
     ```

  3. but a better way is

     ```
     transform( data, data + max, // copy elements from data
     inserter(d, d.begin()), // to the front of
     ```

  4. With a complex algorithem, just use STL alogrithem+lambda

  5. but a better way is

```
for( vector<int>::iterator i = v.begin(); i != v.end(); ++i )
if( *i > x && *i < y ) break;

// This is better version now.
vector<int>::iterator i = find_if( v.begin(), v.end(), [x, y](int &i){i>x &&
```

But if you need a loop that dees something fairly simpler, but would require a confusing tangle of binders and adapters, just use loop.

```
list<Widget> lw;
type list<Widget>::iterator WI;
for_each(lw.begin(),lw.end(),
                  mem_fun_ref(&Widget::redraw) );

transform(data, data+10, inserter(deque, deque.begin()),
        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:

```
binary_search lower_bound upper_bound equal_range
set_union set_intersection set_difference
merge  inplace_merge  includes
unique  unique_copy
```

- Make sure destination ranges are big enough effective stl item 30

```
vector<int> values, result;
int doSth(int x); // function
// make destination range big enough
result.reserve(result.size()+values.size() );

transform(values.begin(), values,end(), result.end(),doSth );
// It's error. transform writes its result by making
//assignment. result.end() has no object at all

transform(values.begin(), values,end(),
  back_inserter(result), doSth)  //insert end

transform(values.rbegin(), values,rend(),
  front_insert(result), doSth)
  //result must be a list
  // 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.

```
vector<Foo> vf;
bool compare(const Foo& f1, constFoo& f2);
partial_sort(vf.begin(),vf.begin()+20, vf.end(),  compare);
sort(vf.begin(),vf.end(), compare);
nth_element(vf.begin(),vf.begin()+20,vf.end(), compare);

bool good(const Foo &f1)
partition(vf.begin(), vf.end(), good);
```

- Be wary of remove-like algorithm on container of pointers

```
void delAndNULL(Foo*& pf){ // use pointer reference here.
if(!pf->isCertified(){ delete pf; pf = nullptr}
}
for_each(v.begin(),v.end(), delAndNULL);
v.erase(remove(v.begin(),v.end(),
                    static_cast<Foo *>(0) ), v.end());
```

- **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.**

1.9.3.2 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

1.9.3.3 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.

```
void fun(int n){
    cout<<" "<<n;
} //all function definition end no semicolon

struct Sum{
    Sum(): sumEven{0} { }  //no semicolon here
    void operator()(int n) { if(n%2 ==0) ; sumEven += n; }
    int sumEven;
}; // type definition need semicolon

vector<int> nums{3, 4, 2, 8, 15, 267};
for_each(nums.begin(), nums.end(), fun); // no () here
Sum s = for_each(nums.begin(), nums.end(), Sum()); //have() here
for_each(nums.begin(), nums.end(), [sumEven](int n){
                                                    if(n%2==0) sumEven+=n;
                                                    });
```

- for_each need a functor, So when to use for_each eaual 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.

     ```
     struct GreatThanX{
         GreatThanX(int x): cutoff{x} { }  //no semicolon here
         bool operator()(int n) { if(n>x) ; return true; return false; }
         int cutoff;
     };

     vector<int> nums{3, 4, 2, 8, 15, 267};
     find_if(nums.begin(), nums.end(),GreatThanX(3));
     copy_if(nums.begin(), nums.end(),GreatThanX(7));
     .........
     ```

  4. **Only functor can be use a argument input to map or set container set**. See below examples.

     ```
     struct lex_compare {
         bool operator() (const int64_t& lhs, const int64_t& rhs) const{
             stringstream s1,s2;          s1 << lhs;  s2 << rhs;
             return s1.str() < s2.str();
         }
     };
     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

```
auto f = [](int x, int y){if(y>x) return true; return false;} ;
auto f1 = bind(f, 3, placeholders::_1) // GreatThanX(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 writed to o.

- If output is associate container, you need use inserter( ) function to get inserter iterator.

```
std::transform(a.begin(), a.end(),
                                    std::inserter(set, set.begin()), modify);

std::transform(a.begin(), a.end(), b.begin(),
                                        a.begin(), plus<int>() );
// 1) you can make in place modification.
// 2) plus minus, multiplies, divides, modulus, negate, equal_to
// are often used in transform algorithms.
```

-

1.9.3.4 Bounding

(f,f) equal_range(f,f,&) (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,&) 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,&) f upper_bound(f,f,&,bpred)	Find the upper bound of a value within a range and return an iterator pointing to it.	

1.9.3.5 Comparing

bool equal(i1,i1last, i2) bool equal(i1,i1last,i2,bpred)	Check if the values in two ranges match.
bool lexicographical_compare (i1,i1last,i2,i2last) bool lexicographical_compare (i1,i1last,i2,i2last,bpred)	Compare two ranges lexicographically, and return true if the first range is less than the second; otherwise return false.
(i1,i2) mismatch(i1,i1last,i2) (i1,i2) mismatch(i1,i1last,i2, bpred)	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:

```
bool is_palindrome(const std::string& s){
    return std::equal(s.begin(), s.begin() + s.size()/2,
                      s.rbegin());
}
```

1.9.3.6 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:

```
for (int i=1; i<=5; i++)
    myvector.push_back(i*10);
    // myvector: 10 20 30 40 50

copy(myvector.begin(), myvector.end(), myvector.begin()+3)
//error, when source and targe is overlap,
//you have to use backward copy

copy_backward(myvector.begin(), myvector.end(), myvector.begin()+4)
// pay attention, copy_backward, *(--last) =
//if you want copy to position 3, you need to input position 4
//or you can input vector.end() as target,
//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 =.

1.9.3.7 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 find match.
n count_if(i,i,upred)	Count the items in a range that satisfy a predicate and return that count.	

1.9.3.8 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:

```
std::vector<int> v(5);
std::generate(v.begin(), v.end(), std::rand);
                    // Using the C function rand()

int n = {0};
std::generate(v.begin(), v.end(), [&n]{ return n++; });
//1, 2, 3, 4, 5
```

Filtering , should used on sorted container

f unique(f,f) f unique(f,f,bpred)	Collapse each group of consecutive duplicate values to a single value, and return an iterator pointing to the end of the modified range.
o unique_copy(i,i,o) o unique_copy(i,i,o,bpred)	Copy a range of values, performing the same action as unique above, and return an iterator pointing to the end of the new range.

```
int myints[] = {10,20,20,20,30,30,20,20,10};
  std::vector<int> myvector (myints,myints+9);

  // using default comparison:
  std::vector<int>::iterator it;
  it = std::unique (myvector.begin(), myvector.end());
  // 10 20 30 20 10 ? ? ? ?
```

Heap

make_heap(r,r) make_heap(r,r,bpred)	Make a range of values into a heap.	It nees random iterator. priority_queue use them inside. You don't use them directly
o pop_heap(r,r) o pop_heap(r,r,bpred)	Delete the first value from a heap.	
push_heap(r,r) push_heap(r,r,bpred)	Insert the last value of a range into a heap.	
sort_heap(r,r) sort_heap(r,r,bpred)	Sort a heap.	

1.9.3.9 Math

v accumulate(i,i,v) v accumulate(i,i,v,bfunc)	Add an initial value and the values in a range, return sum.
o adjacent_difference (i,i,o) o adjacent_difference (i,i, o, bfunc)	Calculate the difference between adjacent pairs of values, write the differences to an o, and return the end of that output range.
v inner_product (i1,i1last,i2,vInitial) v inner_product (i1,i1last,i2,v,bfunc1,bfunc2)	Calculate the inner product of two ranges and return that value plus vInitial.
o partial_sum(i,i,o) o partial_sum(i,i,o, bfunc)	Fill a range with running totals and return an iterator pointing to.

- An example about partial_sum

```
std::vector<int>v(10, 2);  // new initialize method
std::partial_sum(v.begin(), v.end(),
            std::ostream_iterator<int>(std::cout, " "));
    //2 4 6 8 10 12

std::partial_sum(v.begin(), v.end(), v.begin(),
                                 std::multiplies<int>());
std::cout << "The first 10 powers of 2 are: ";
 for (auto n : v) {
      //2 4 8 16 32
    }
```

- An example about accumulate

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = std::accumulate(v.begin(), v.end(), 0);
int product = std::accumulate(v.begin(), v.end(), 1,
                                        multiplies<int>());

std::string s = std::accumulate(v.begin(), v.end(),
                                    std::string{},
                                [](const std::string& a, int b) {
                return a.empty() ? to_string(b)
                                 : a + '-' + to_string(b); });
                            to_string is in C++11
```

- An example about adjacency_difference

```
v = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
    std::adjacent_difference(v.begin(), v.end() - 1,
                             v.begin() + 1, std::plus<int>());
          //fibonacci
```

- difference some algorithms

1. For accumulate, input a init, ->calculate(init, element)->return it to init.
2. For adjacent_difference: first -> dfirst, f(first+1, first)->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.

1.9.3.10 Merging

inplace_merge(b,b,b) inplace_merge(b,b,b,bpred)	Merge two sorted ranges, in place, into a single sorted range.
o merge(i1,i1,i2,i2,o) o merge(i1,i1,i2,i2,o,bpred)	Merge two sorted ranges into a single sorted range.

Min/Max

& min(&,&) & min(&,&,bpred)	Find the minimum of two values and return a reference to that value.
& max(&,&) & max(&,&,bpred)	Find the maximum of two values and return a reference to that value.
f min_element(f,f) f min_element(f,f,bpred)	Find the minimum value in a range and return an iterator pointing to that value.
f max_element(f,f) f max_element(f,f,bpred)	Find the maximum value in a range and return an iterator pointing to that value.

1.9.3.11 Partitioning

nth_element(r,r,r) nth_element(r,r,r,bpred)	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.
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:

```
std::vector<int> v = {0,1,2,3,4,5,6,7,8,9};
auto it = std::partition(v.begin(), v.end(),
                         [](int i){return i % 2 == 0;});
 std::copy(std::begin(v), it,
               std::ostream_iterator<int>(std::cout, "␣"));
 std::copy(it, std::end(v),
               std::ostream_iterator<int>(std::cout, "␣"));
 \\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.

1.9.3.12 Permuting

bool next_permutation(b,b) bool next_permutation (b,b,bpred)	Change a range of values to the next lexicographic permutation of those values, and return true, or false if no next permuation exists.
bool prev_permutation (b,b) bool prev_permutation (b,b, bpred)	Change a range of values to the previous lexicographic permutation of those values, and return true, or return false if no previous permuation exists.

A example:

```
int a[] = {1,2,3};
do {
    cout << a[0] << '␣' << a[1] << '␣' << a[2] << '\n';
} while ( std::next_permutation(a,a+3) );
1 2 3 // 1 3 2 //2 1 3
2 3 1 //3 1 2  //3 2 1
```

1.9.3.13 Random/shuffing

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.

1.9.3.14 Removing

remove(f,f,&) remove_if(f,f,upred)	Remove from a range of values all values that match a give value or satisfy a predicate
remove_copy(i,i,o,&) remove_copy_if(i,i,o,upred)	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.

1.9.3.15 Replacing

replace(f,f,&,&) replace_if(f,f,upred,&)	Replace, within a range of values, one specified value(satisfies a predicate) with another value.
replace_copy(i,i,o,&,&) replace_copy_if(i,i,o,upred,&)	Copy replacing one specified value with another specified value.

1.9.3.16 Reverse

reverse(b,b)	Reverse the order of all values in a range of values.
reverse_copy(b,b,o)	Reverse and copy

1.9.3.17 Rotating

rotate(f,f,f)	Rotate a range of values by n positions.
rotate_copy(f,f,f,o)	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.

```
for (int i=1; i<10; ++i) myvector.push_back(i);
// 1 2 3 4 5 6 7 8 9
rotate(myvector.begin(),myvector.begin()+3,
                                    myvector.end());
                              // 4 5 6 7 8 9 1 2 3
```

1.9.3.18 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) f adjacent_find(f,f, bpred)	the first pair of equal adjacent values in a range and return an iterator pointing to the first value of the pair.
i find(i,i,&)	return an iterator pointing to the value or end
i find_if(i,i,upred)	satisfies a predicate and return an iterator pointing to the first such value, or to the end

3: Searches for a single element from a range

f1 find_first_of(f1,f1,f2,f2) f1 find_first_of(f1,f1,f2,f2,bpred)	std::find_first_of searches for a single element from a range within another range.

4: Below 3 algorithms searche for a whole range of elements within another range

f1 search(f1,f1,f2,f2) 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 find_end(f1,f1,f2,f2) f1 find_end(f1,f1,f2,f2,bpred)	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)
f search_n(f,f,n,&) f search_n(f,f,n,&,bpred)	For a contiguous sequence of n values each equal to &, return iterator to the first of those values, or the end of the range

1.9.3.19 set

bool includes(i1,i1,i2,i2) bool includes(i1,i1,i2,i2,bpred)	Search for **all values from the second range in the first range** and return true if found, or false
o set_difference(i1,i1,i2,i2,o) o set_difference (i1,i1,i2,i2,o,bpred)	in the first range but not in the second range and return the end of that output range.
o set_intersection(i1,i1,i2,i2,o) o set_intersection (i1,i1,i2,i2,o,bpred)	in the first range and also in the second range and return the end of that output range.
o set_union(i1,i1,i2,i2,o) o set_union(i1,i1,i2,i2,o,bpred)	either in the first range or in the second range and return the end of that output range.
o set_symmetric_difference (i1,i1,i2,i2,o) o set_symmetric_difference (i1,i1,i2,i2,o,bpred)	not common to both ranges and return the end of that output range.

- All algorithm need two ranges should be sorted.

1.9.3.20 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.

1.9.3.21 sort

partial_sort(r,r,r) partial_sort(r,r,r,bpred)	Sort all values till first part of range is in sorted order.
r partial_sort_copy(i,i,r,r) r partial_sort_copy (i,i,r,r,bpred)	Partially sort a range of values (as above) and copy **as many values as will fit into an output range.**
sort(r,r) sort(r,r,bpred)	Sort a range of values.
stable_sort(r,r) stable_sort(r,r,bpred)	Sort and maintaining the same relative order of duplicate values.

1.9.4 Function object

1.9.4.1 Basic

1. In C language, we use function pointer.

```
int(*pf)(int, int); //declare a pointer function
int fun1(int i,int j);
pf = fun1 // or pf=&fun1; we usually skip &.
(*pf)(1,2) //or pf(1,2);
```

2. In C++, Some generic algorithms can also accept function.

```
void fun(int i) {
  //do stuff
}
//here just input fun name, it's function pointer usage
for_each(a.begin(), a.end(), fun);
```

3. But passing function has shortcomings 1) Can't be inline. 2) sometimes It can't be compile due to different compiler implementation. 3) 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.

```
class functor { // also call function objects
public:
  void operator()(int i);
};
//here, use functor, (class object);
//overload operator();

for_each(a.begin(), a.end(), functor());
// You must use functor()
//with () 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.

```
// you can use class instead struct, but you need to make
//operator() public, in struct, default is public, so struct is better!
struct less_than_7 : std::unary_function<int, bool>{
    bool operator()(int i) const { return i < 7; }
};

class less_than_value : std::unary_function<int, bool>{
    less_than_value(int x) :value(x) {};
    bool operator()(int i) const { return i < value; }
    private:
    int value;
};

std::count_if(v.begin(), v.end(), std::not1(less_than_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 you 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.

```
class yanCompare{
    bool operator()(string &s1, string &s2){....;}
};

set<string, yanCompare> setDic;
//just yanCompare, no() follow it. pass into a type, not obj


remove_if(...yanComare() );
//yanCompare(), 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 and value,

   (c) expression has no name but has value,

   (d) value can be divided by three categories, can overload move ctor.

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.

```
[]->bool(int){return x<7};
// if only return, you can omit ->bool (return type);
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:

1.9.4.2 Adaptable

- Four adapter is not1, not2, bind1st and bind2nd. It can change you 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.

```
int array[] = {10,20,30,40,50,10}; int cx;
cx = count_if (array, array+6, bind1st(equal_to<int>(),10));
```

- In order to use these four adapters. You functor should be inheriated from unary_funciton or binary_function. It make you function object has typedef information, which not1 or not2 will use them.

- Usage of ptr_fun, If you want to adapter a current function, (not function object), because function doesn't have any typedef information, so you have to use ptr_fun firstly.

```
bool IsBad(Foo& f);

find_if(vect.begin(), vect.end(),
  not1(IsBad);  //error! don't compile;

find_if(vect.begin(), vect.end(),
  not1(ptr_fun(IsBad ) ); //OK, now:

//recommend to use c++11 new feature
//   std::not1(std::cref(isvowel)));
//   std::not1(std::function<bool(char)>(isvowel)));
```

- Usage of mem_fun, mem_fun_ref

```
list<Foo*> lpf
for_each(lpf.begin(),lpf.end(),mem_fun_ref(&Foo::testFun) );

list<Foo> lf
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: depracated 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**

1.9.4.3 functor tips

- functor can be template.

```
template<typename Type>
class TooBig{
Type cutoff;
bool operator()(const T&v){ return v>cutoff};
}
```

- For template functor, You can get value type from iterator

```
template<typename T>
struct Average{
T operator()(T t1, T t2){t1+t2/2;}
};

transfor(beg1, end1, beg1,
Average<typename iterator_traits<beg1>::value_type>() );
// 1) I also can extract value_type information from container.
// 2) This information can be used in template function object.
```

- Any functor type should consistent with container type. Pay attention to the second example, you just input Foo to binary_function template, no const and &.

```
list<Foo*>
struct PtrFooCompare:binary_function
                <const Foo*, const Foo*, bool>{
bool operator(const Foo*, const Foo*){...}
}

list<Foo>
struct FooCompare:binary_function
                <Foo, Foo, bool>{
bool operator(const Foo&, const Foo&){...}
}
```

- 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.

```
class Predicate: public unary_function<Foo, bool>{
public:
bool operator()(const Foo& f) const{ // two const
 cutoff++; // this statment will not compile.
}
private:
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.

- Generator no argument, unary functions with one argument and binary function with two arguments

- A long story,part 1 , begin from a program, a function can be declare with parameter name.

```
int fun (double); //It's ok when declare,
int fun (double(d) );
// it's ok, () is superfluous and are ignored.
//but when you define fun, you must give parameter name.

int fun (double() );// fun take a function pointer as parameter
// I omit the parameter name,
//pointer point a function return double and take nothing.
```

- part2, when you call a function, you can pass a non-name temporary arguments. here, you don't need give a name of parameter, the const reference will prolong the temporary argument life until the end of function, You have to use const, because you can't change a temporary arguments.

```
fun(const  Foo &fc);

fun(ftemp());  //ftemp return a Foo
fun(Foo()); //un-name Foo temporary obj;
fun( fc1+fc2); // just like ftemp
fun(1) // implicit change from 1 to Foo
```

- part3, When you define a object of a class, you may make a mistake.

```
Foo fo; //define a obj
Foo fo(); //error decalre a function fo
//fo take nothing and return Foo.
```

- part4, Now let see this example. I want to build a list obj, name is data. But compiler think that I declare a function, name is data, take two arguments, and return a list<int>.

```
list<int> data(istream_iterator<int>(dataFile),
   //think as double(d); such as double d.
   //in fact, it's istream_iterator<int> temp (dataFile)
    istream_iterator<int> ()
```

```
    //think as a function pointer.
            );
```

- part5, you need to write below code to define a list obj. Detail can be seen in effective STL item 6.

```
istream_iterator<int> beg (dataFile);
istream_iterator<int> end;

list<int> data(beg, end);
```

1.10 concurrent

1.10.1 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++.

```
op++;

move mem reg
// Thread1 change to thread 2 here.
add reg 1
move reg mem
```

- In visual studio. You can use debug view assemble language produced by compiler.

1.10.2 synchronization

1.11 New Feature in C++14/C++17

1.11.1 New Type

- Add three new data types " long long", "char16_t" and "char32_t".
- Going by the standard:
  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.**

```
int8_t and uint8_t
int16_t and uint16_t
int32_t and uint32_t
int64_t and uint64_t
```

- The fixed-width integers have two downsides: First, they may not be supported on architectures where those types canâĂŹt 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

- integer best practices:

  int should be preferred when the size of the integer doesnâĂŹt matter (e.g. the number will always fit within the range of a 2 byte signed integer). For example, if youâĂŹre asking the user to enter their age, or counting from 1 to 10, it doesnâĂŹt matter whether int is 16 or 32 bits (the numbers will fit either way). This will cover the vast majority of the cases youâĂŹre likely to run across.

  If you need a variable guaranteed to be a particular size and want to favor performance, use std::int_fast_t.

  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:

  Unsigned types, unless you have a compelling reason. The 8-bit fixed-width integer types. Any compiler-specific fixed width integers – for example, Visual Studio defines __int8, __int16, etcâĂę

- 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

```
struct X { int id; /* ... */ };
bool operator==(X a, X b) { return a.id == b.id; }

struct MyHash {
  size_t operator()(const X& x) const { return std::hash<int>()(x.id); }
};

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:

```
namespace std {
  template <> struct hash<Foo>{
    size_t operator()(const Foo & x) const{
      /* your code here, e.g. "return hash<int>()(x.value);" */
    }
  };
}

std::unordered_set<Foo> s;
```

- A good std::hash example can be found in cppreference.com

- 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.

```
  int val = 0b11110000;
  std::cout << 0b1000'0001'1000'0000;
␣␣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.

```
class [[deprecated]] flaky {
};
[[deprecated("Consider␣using␣something␣other␣than␣cranky")]]
int cranky(){
   return 0;
}
// below two statements produce compiler warning.
  flaky f;
  return cranky();
```

1.11.1.1 std::array

- 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.

```
//Making a new array via copy
auto a3 = a2;

//This works too:
auto a4(a2);

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.

```
void carr_func(int * arr, size_t size){
    std::cout << "carr_func - arr: " << arr << std::endl;
}

//Error:
//carr_func(a2, a2.size());

//OK:
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.

```
#include <iostream>
#include <array>
void printLength(const std::array<double, 5> &myarray){
    myarray.sort();
    std::cout << "length: " << myarray.size();
}

std::array<double, 5> myarray { 9.0, 7.2, 5.4, 3.6, 1.8 };
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.**

1.11.2 range base

- A range-based for loop is introduced in C++11. Use `auto&` to aoid 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>.)

     ```
     for (const auto& elem : container)
     // 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.

     ```
     for (const auto elem : container)
     // 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:

     ```
     for (auto& elem : container)
     // 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 reccommend using it.

     ```
     //make sure Container contain pointer
     for(auto *ptr: Container){ ptr->change();}
     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.

```
int fun(){...};
int& a = fun(); // reference can't bind to temporary prvalue.

vector<bool> v = {true, false, false, true};
for (auto& x : v)
x = !x;
```

- If you want to read, you can use two below methods:

```
for(auto i: bv)

for(auto const & i: bv)
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>

```
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.

1.11.3 callable

1.11.3.1 std::function

- std::function can be called as **wrapper**, it use **type eraser** technology. It's a variadic template. A basic example of std::function is below:

```
//list of callable objects
void print_num(int i){  //1) common function
    std::cout << i << '\n';
}
struct PrintNum {  //2) functor
    void operator()(int i) const{ std::cout << i ;}
    };

std::function<void(int)> f_display;

f_display = print_num;  //plain function
f_display = PrintNum(); //functor
f_display = [](int) { print_num(42); }; //lambda
// you can call fun inside of lambda

f_display(-9); // call function.

std::function<void()> f_display_3 = std::bind(print_num, 3);
```

- It copy and save a callable object. see code below.

```
int main() {
    int value = 5;
    typedef std::function<void()> fun;
    fun f1 = [=]() mutable { std::cout << value++ << '\n' };
    fun f2 = f1;
    f1();                          // prints 5
    fun f3 = f1;
    f2();                          // prints 5
    f3();                          // prints 6 (copy after first increment)
}
```

- 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:

```
std::function<int ()> func;
.......
if ( func ) { // if we did have a function, call it
     func();
}
```

- 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.

```
void run_within_for_each(std::function<void (int)> func){
     vector<int> numbers{ 1, 2, 3, 4, 5, 10, 15, 20, 25 };
     for_each(numbers.begin(), numbers.end(), func);
}

void fun(int x){ }  //1) function pointer
  cout<<x<<endl;
};
auto lambda1 = [](int y){  //4) lamba
    cout << y <<endl;
};

run_within_for_each(fun);
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?"

1.11.3.2 member function

- std::function can be used for member function.

```
struct Foo {
    Foo(int num) : num_(num) {}
    void print_add(int i) const { std::cout << num_+i << '\n'; }
```

```
    int num_;
};

std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
const Foo foo(3);
f_add_display(foo, 1); // store a call to a member function
//method 1, use std:ffunction directly.


std::function<void(int)> f_add_display2=
                          std::bind( &Foo::print_add,  foo,  _1 );
f_add_display2(2); // a call to a member function and object


std::function<void(int)> f_add_display3=
                         std::bind( &Foo::print_add, &foo, _1 );
f_add_display3(3); // a call to a member function and object ptr

//method 2, use bind
```

- If you want to input obj directly, Use a lambda that captures the foo instance and invokes print_add on it. Another option is to use std::bind to bind the foo instance:

```
typedef std::function<void(int)> fp;

void test(fp my_func){
  my_func(5);
}
Foo foo;
test([&foo](int i){ foo.print_add(i); });


test(std::bind(&Foo::print_add, &foo, std::placeholders::_1));
```

- 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:

```
struct A {
    int x;
    int getX() { return x; }
    int add(int y) { return x+y; }
};

A a{2};

auto get1 = std::mem_fn(&A::getX);
auto get2 = std::bind(&A::getX, _1);
```

```
get1(a); // yields 2
get2(a); // same

auto add1 = std::mem_fn(&A::add);
auto add2 = std::bind(&A::add, _1, _2);

add1(a, 5); // yields 7
add2(a, 5); // same
```

- Three ways to use member function.1) use std::function, 2) use bind, 3) use mem_fun. All the detail can be found in the previous example. **Prefer mem_fn than bind because it's verbose, Perfer mem_fn than std::function, because it lightweight.**

- mem_fn(C++11) and mem_fun. **mem_fn is new, don't use mem_fun any longer**

  1. mem_fn is faster than bind. so prefer mem_fn first.
  2. std::mem_fun 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_fun and std::mem_fun_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.

```
struct Foo {
    void display_number(int i) {
        std::cout << "number: " << i << '\n';
    }
    int data = 7;
};

Foo f;
auto print_num = std::mem_fn(&Foo::display_number);
print_num(f, 42);
// for mem_fn, you have to give a object:f
```

- **bind is deprecated in C++11, and will be given up in C++14. perfer to use lambda than bind, detail can be found in effective modern c++ item**

1.11.3.3 lambda

- lambda basic syntax and an example.

```
[ captures ] (parameters) -> returnTypesDeclaration { ..; }
int sum = 0, divisor = 3;
vector<int> vc { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45 };
for_each(vc.begin(), vc.end(), [divisor, &sum] (int y){
      if (y % divisor == 0){
         cout << y << endl;
```

```
            sum += y;
        }
    });
 cout << sum << endl;
```

- Lambda is used on complex type container. You can use typedef.

```
typedef std::vector< std::pair<int,std::string> > Record_t;
// typedef is your good friend, reuse it below
Record_t k1;

int find_it(std::string value, Record_t const& stuff){
  auto fit = std::find_if(stuff.begin(), stuff.end(),
              [value](Record_t::value_type const& vt) -> bool
                   { return vt.second == value; });
  if (fit != stuff.end())
            return fit->first;
}
```

- 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):

```
auto factorial = [](int i, int j) {return i * j;};

int arr{1,2,3,4,5,6,7,8,9,10,11,12};
long res = std::accumulate(arr, arr+12, 1, factorial);
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:

```
typedef int (*func)();
func f = [] () -> int { return 2; };
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 MessageSizeStor().**

```
class MessageSizeStore{
    public:
    MessageSizeStore () : _max_size( 0 ) {}
    void getMaxMessage (const std::string& message ) {...}
    int getSize (){...}
private:
    int _max_size;
};
```

```
MessageSizeStore size_store;
for_each(s.begin(),s.end(),
        [&] (const std::string& message) {
                size_store.checkMessage( message );
                });
```

- Why functor is better than function pointer? because functor can be inline.
- Why lambda is bettern 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.

```
int count3 = 0;
std::count_if(v.begin(), v.end(),
[&count3](int x){count3 += (x%3 == 0)} );
//how many number can be divided by 3.
```

1.11.3.4 summary

- 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	std::function	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 (ugly type)	yes	(yes)

1.11.4 Other New Feature

1.11.4.1 delctype

- decltype and Trailing Return type(C++11 new feature)

```
template<typename T1, typename T2>
void f(T1 x, T2 y){
decltype(x+y) xpy = x+y;
}

//with return value;
template<typename T1, typename T2>
auto f(T1 x, T2 y) ->decltype(x+y)
{
x+y;
}
```

- How to make your template class has more generic use for client, "exceptional C++" item 5 introduces more idea in this topics.

1.11.4.2 constexpr

- Unlike templates and preprocessor macros, constexpr allows for loops and recursion at compile-time without extreme boilerplate.
- 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 question is : when should you use constexpr? The answer is the same than with const: as much as you can.
- The most powerful thing about constant expressions, is that they enable you to do meta-programming without resorting to templates.
- A good artical is "Demystifying constexpr".

1.11.4.3 alias declaration

- Alias declaration is better than typedef when you declare a function pointer, It's more verbose.

```
// FP is a synonym for a pointer to a function taking an int
// and a const std::string& and returning nothing
typedef void (*FP)(int, const std::string&);

using FP = void (*)(int, const std::string&);
// alias declaration
```

- C++11 doesn't support typedef template directly, only can be put inside a struct.

```
template<typename T> // MyAllocList<T>
using MyAllocList = std::list<T, MyAlloc<T>>;
// is synonym for std::list<T, MyAlloc<T>>
MyAllocList<Widget> lw; // client code

template<typename T>
struct MyAllocList {
typedef std::list<T, MyAlloc<T>> type;
};
// MyAllocList<T>::type is synonym for std::list<T,  MyAlloc<T>>
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.

```
template<typename T>
class Widget { // Widget<T> contains
private: // a MyAllocList<T>
typename MyAllocList<T>::type list; // as a data member
};
```

- A better way is to use Alias name

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>; // as before

template<typename T>
class Widget {
private:
MyAllocList<T> list; // no "typename" and no "::type"
};
```

- Another advantage of Alias Name is below:

```
template<typename T>
struct type{ typedef std::vector<T> sometype; }

template<typename TT>
void someFunction( typename type<TT>::sometype& myArg );
//You need typename here, because sometype is depended T.
//typename indicates type<TT>::sometype is a type.

std::vector<int> a;
someFunction(a); // error, cannot deduce 'TT'
someFunction<int>(a);


template<typename T>
using sometype = std::vector<T>;

template<typename T>
void someFunction(sometype<T> &myArg );

std::vector<int> a;
someFunction(a);
```

- C++14 offers alias templates for all the C++11 type traits transformations.

```
template <class T>
using remove_const_t = typename remove_const<T>::type;
template <class T>
using remove_reference_t = typename remove_reference<T>::type;
template <class T>
using add_lvalue_reference_t =
typename add_lvalue_reference<T>::type;
```

1.11.4.4 scoped enums

1. C++98-style enums are now known as unscoped enums.

```
enum Color { black, white, red };
// black, white, red are in same scope as Color
auto white = false; // error! white already

enum class Color { black, white, red };
// black, white, red are scoped to Color
auto white = false; // fine, no other

Color c = white; // error!
Color c = Color::white; // fine
auto c = Color::white; // also fine (and in accord
// 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.

```
enum class Color { black, white, red }; // enum is now scoped
Color c = Color::red; // as before, but
```

```
... // with scope qualifier
if (c < 14.5) { // error! can't compare
// 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.

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
basic_ios(const basic_ios& ) = delete;
basic_ios& operator=(const basic_ios&) = delete;
};

bool isLucky(int number); // original function
bool isLucky(char) = delete; // reject chars
bool isLucky(bool) = delete; // reject bools

template<typename T>
void processPointer(T* ptr);
template<>
void processPointer<void>(void*) = delete;
//you can't not use void to instantiate
```

1.11.4.5 noexcept

1. When an exception is thrown from an noexcept function, std::terminate gets triggered. compiler will not check it for you.

2. noexcept is part of a function's interface, and that means that callers may depend on it.

3. noexcept functions are more optimizable than non-noexcept functions.

4. noexcept is particularly valuable for the move operations, swap, memory deallocation functions, and destructors.

5. Most functions are exception-neutral rather than noexcept.

1.11.4.6 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.

```
template<typename T>
T adder(T v) {
  return v;
}

template<typename T, typename... Args>
T adder(T first, Args... args) {
  return first + adder(args...);
}
////////////////////////////
template<typename T>
show_list(const T& value){cout<<value<<endl;}

template<typename T, typename... Args>
show_list(const T& value, const Args&... args){
cout<<value<<endl;
show_list(args...);
}
```

- clang++ -S -emit-llvm main.cpp -o out.bc