C++ 03

Content

What are the following?

NVI	=	non virtual interface
CRTP	=	curiously recurring template pattern
DCLP	=	double checked locking pattern
NRVO	=	named return value optimization
TRTS	=	trailing return type syntax

A. Fundamental

A1. Fundamental principles

- abstraction
- encapsulation
- modularity
- reusability
- maintainability : anticipation of change
- extensibility : incremental development

1. Program flow :

```
if else, for, while, switch
```

2. Data type :

```
void / char / short / long / double / std::uint8_t, std::uint16_t, std::uint32_t, std::uint64_t
```

pointer based string

```
char string[] = {'A','B','C','D','E','\0'};
```

array ended with `NULL` character `'\0'`

```
char string[] = "ABCDE";
char *string  = "ABCDE";
```

3. What is enum?

```
class X
{
    enum        { good, norm = 10, poor = 20 };
    enum option { option_X, option_Y = 100, option_Z };
};
```

equivalent to

```
class X
{
    static const int good = 0;
    static const int norm = 10;
    static const int poor = 20;
    static const int option_X = 0;
    static const int option_Y = 100;
    static const int option_Z = 101;
};
```

4. What is typedef?

```
class X { typedef typename T<U>::V type; };
```

5. What stuffs must be initialized when declared?

- reference
- const object

6. What stuffs must be initialized in member initializer?

- reference
- const member
- direct initialization of base class
- direct initialization of members

A2. C++ Keywords

- about lifetime & mutability — `static / const / mutable`
- about external & internal linkage — `extern / static / inline / static inline / static constexpr` *(red = 2nd meaning)*
- about type casting — `explicit / const_cast / static_cast / dynamic_cast / reinterpret_cast` *(see later section)*
- about polymorphism — `template / virtual / override / final` *(see later section)*
- other — `friend / register / volatile / noexcept / nothrow`

About automatic and static

There are 2 concepts : lifetime and scope (of a variable). Lifetime is the period in between construction and destruction, while scope is where the object can be seen. Possible scopes of C++ identifiers are : namespace / class / function / function prototype / block / file. Automatic variable (non-static) has lifetime limited to their scopes, because they are destructed running out of scope. Static variable has lifetime extended from the first encounter to program termination, it can be declared by yeyword `static`.

Keyword `static` is versatile, it has different meaning different context. There are 5 cases when keyword `static` is used :

- `static` global variable - not relate to lifetime, it means function with internal linkage
- `static` global function - not relate to lifetime, it means function with internal linkage
- `static` local variable in function - the variable is constructed once only, it lives until program ends, it is a sticky variable
- `static` member variable - can be accessed without declaring object, it is class-wise member, not object-wise member
- `static` member function - can be accessed without declaring object, it is class-wise member, not object-wise member

`Static` member function can invoke other `static` member functions only. `Static` member function is initialized outside declaration :

```
// 1. declare static member inside class
class X { static T static_member; };

// 2. define static member outside class
X T::static_member = create_rvalue_instance_X();
```

About const

Constness is a declaration of no-change. There are 3 cases when keyword `const` is used:

- `const` local variable in function - we can invoke its `const` member function only
  - we can pass it to other functions as `const` input argument only
- `const` member variable - it cannot be modified once object is initialized
- `const` member function - it cannot modify any member, except `mutable` member

`Const` member function can invoke other `const` member functions only.

About mutable

Keyword `mutable` allows member to be modified inside `const` member function, particularly useful when member may be modified by a `get()` function, and it is not an object state contextually, such as `thread_safe_container::get() const` function, with a mutex lock in the container, the mutex lock is declared `mutable`.

Translation unit / Declaration vs Definition / One definition rule (ODR)

Symbols can be a function or a variable. Object is an instantiation of a class, object is also a variable.

```
namespace {
    // declaration of function
    void f0(const A&, const B&);                symbol = f0, type = void(const A&, const B&)
    void f1(const A&, const B&, const C&);      symbol = f1, type = void(const A&, const B&, const C&)
    // declaration of variable
    A a;                                        symbol = a, type = A
    B b;                                        symbol = b, type = B
}

// definition of function
void ns::f0(const A&, const B&)           { ... }
void ns::f1(const A&, const B&, const C&) { ... }
// definition of variable
ns::A a{1,2,3,4,5};
ns::B b{1,2,3,4};
```

C++ allows multiple declarations, however it does not allow multiple definitions. This is called One definition rule ODR.

```
int f();                        // multiple declaration is fine
int f();
int f();
int f() { return 123; }         // as long as there is single definition
```

In C++, here is the build process of an executable or a static / shared library :

- for each cpp file — preprocessor copy all header content (as well as macro, `#define`) forming a translation unit
- for each transaction unit — compiler convert it into object file *(1 to 1 mapping between translation unit and object file)*
- for all object file — linker links all of them to form an executable or library

Compilation vs linking :

- compilation by compiler — requires the declaration of functions / variables that a transaction unit depends on
- linking by linker — requires the definition of functions / variables that a transaction unit depends on

Declaration of functions / variables that this translation unit requires should be sequenced before the place of invocation :

- by forward declaration in the same translation unit *OR*
- by inclusion of header in the same translation unit

Definition of functions / variables that this translation unit requires, should be :

- found in current translation unit *OR*
- found in other translation unit *AND* declared as visible by other translation units (i.e. external linkage)

In short, suppose symbol of a variable or a function is declared in header `x.h` and defined in source file `x.cpp`, then :

- its declaration can be seen by other translation unit `y`, if `y.cpp` includes `x.h`
- its definition can be seen by other translation unit `y`, if the symbol is declared as external linkage in `x.h`
  its definition cannot be seen by other translation unit `y`, if the symbol is declared as internal linkage in `x.h`

Two remarks about header :

- `#include <header.h>` means — compiler to look for header in project setting
  `#include "header.h"` means — compiler to look for header in source code directory
- in order to avoid recursive inclusion of header, we need to wrap all header content inside the preprocessor :

```
#ifndef TIME_H
#define TIME_H
...
#endif
```

<u>Internal linkage vs External linkage</u>

Please read *Internal and External Linkage in C++, Peter Goldsborough*. Here are the rules that govern external / internal linkage :

- by default, variable declaration `const A a;` has an internal linkage
- by default, variable declaration `A a;` has an external linkage
- by default, function declaration `void f();` has an external linkage — *what we normal do for global functions*
- explicitly, variable declaration `static A a;` has an internal linkage
- explicitly, variable declaration `extern A a;` has an external linkage — *what we normal do for global variable*
- explicitly, function declaration `static void f();` has an internal linkage
- explicitly, function declaration `extern void f();` has an external linkage

- hence keyword `static` has a different meaning here, not relevant to lifetime
- Why is global variable evil? *Coupling among different classes. Concurrency issue.*

Here are different combinations :

- declaration and definition in separated files

```
0.  // header.h          // src0.cpp               // src1.cpp
    extern A a;          #include "header.h"       #include "header.h"
                         A a{1,2,3,4};             a.mem_fct();            ⇒ OK (This is how global variable works)

1.  // header.h          // src0.cpp               // src1.cpp
    int f();             #include "header.h"       #include "header.h"
                         int f() { return 1; }
                         f();                      f();                    ⇒ OK

2.  // header.h          // src0.cpp               // src1.cpp
    static int f();      #include "header.h"       #include "header.h"
                         int f() { return 1; }
                         f();                      f();                    ⇒ error, missing definition f in src1
```

- declaration and definition in header

```
1a. // header.h                     // src0.cpp
    int f() { return 1; }           #include "header.h"
                                    f();                                                   ⇒ OK

1b. // header.h                     // src0.cpp                    // src1.cpp
    int f() { return 1; }           #include "header.h"            #include "header.h"
                                    f();                           f();                    ⇒ error, f is defined as external link
                                                                                             in both src0 and src1
                                                                                             (duplicated definitions)
2.  // header.h                     // src0.cpp                    // src1.cpp
    static int f() { return 1; }    #include "header.h"            #include "header.h"
                                    f();                           f();                    ⇒ OK, f is defined as internal link
                                                                                             in both src0 and src1
3.  // header.h                     // src0.cpp                    // src1.cpp               (multi invisible definitions)
    inline int f() { return 1; }    #include "header.h"            #include "header.h"
                                    f();                           f();                    ⇒ OK, f is defined by substitution
                                                                                             (allow violation of ODR)
```

What are the differences among the above 3 cases (1a/b, 2, 3)? All 3 cases have `f` defined in header, so when that header is included in various `cpp` files, multiple definitions of `f` exist. This is how the 3 treatments differ :

1. external linkage (`extern`) — compilation error for duplicated definitions, unless only one `cpp` includes the header
2. internal linkage (`static`) — compilation ok, each `cpp` has its own definition `f`, each `cpp` cannot see the definition in other `cpp`
3. inline function (`inline`) — compilation ok, `inline` allows violating *One Definition Rule* as long as definitions are the same

Anonymous namespace

Anonymous namespace ensures all symbols declared inside namespace have internal linkage, hence it is an alternative to `static`.

```
2a. // header.h                     // src0.cpp                    // src1.cpp
    { int f() { return 1; } }       #include "header.h"            #include "header.h"
                                    f();                           f();                    ⇒ OK, f is defined as internal link
                                                                                             in both src0 and src1
                                                                                             (multi invisible definitions)
```

Inline / static inline / static constexpr

Please read *C++ Inline Variables and Functions, Pablo Arias*. The keyword `inline` has two meanings :

- `inline` function means eliminating cost of function call by copying function definition directly into caller's body
- `inline` function also allow violation of One Definition Rule (ODR) :
  - it allows symbol to be defined multiple times, but does NOT allow multiple definitions (i.e. all definition are the same)
  - it allows non-template function to be defined in header, a essential requirement for header-only library
- `inline` variable is introduced in C++17, mainly used for in-class initialization of `static` member

```
// header.h                     // src0.cpp                 // src1.cpp            // before c++17
struct X                        #include "header.h"         #include "header.h"
{
    static A a;                 A X::a{1,2,3};              X::a.mem_fct();        ⇒ OK
};  // declaration              // definition

// header.h                     // src0.cpp                 // src1.cpp            // post c++17
struct X                        #include "header.h"         #include "header.h"
{
    static inline A a{1,2};     X::a.mem_fct();             X::a.mem_fct();        ⇒ OK, since inline allows violating ODR
};  // declaration & definition
```

- besides, `constexpr` is implicitly `inline`, hence we can replace `static inline` with `static constexpr` (if it is compile-time constant)

```
// header.h                     // src0.cpp                 // src1.cpp
struct X                        #include "header.h"         #include "header.h"
{
    static constexpr A a{1,2};  X::a.mem_fct();             X::a.mem_fct();        ⇒ OK, since constexpr is inline
};  // declaration & definition
```

However `inline` function does not guarantee copying and eliminating function call. We can force `gcc` compiler to do so by :

```
inline __attribute__((always_inline)) void f(const A& a, const B& b) { ... }
```

About register and volatilty

Keyword `register` is a hint to compiler that a variable is heavily used, it's better to store it in register rather than memory, however compiler may not follow the hint. Finally, `volatile` is used in IO and multithreading, for variables which are seemingly constant in code, but actually keep updated by IO or other threads, they are likely to be optimised by compiler, leading to undesired results, if they are not declared as `volatile`.

A3. Pointer, array, reference and member pointer

	used in declaration	used as operator
`*`	declare pointer	dereference operator of pointer
`[]`	declare array	dereference operator of array
`&`	declare lvalue ref	address operator of lvalue variable
`&&`	declare rvalue ref	-
`.  .*`	-	member access operator
`-> ->*`	-	redirection operator

Pointer, array and reference

Array is a constant pointer to const or non-const object. Array is declared :

```
int a[] = {1,2,3,4};
int b[4] = {1,2,3,4};
int c[2,4] = {{1,2,3,4}, {5,6,7,8}};
int d[NZ][NY][NX];                          // then d + n ≡ d + n * NY * NX * sizeof(int)
```

Reference is a constant pointer, which can be auto-dereferenced without `*`. It must be initialised, cannot be reassigned.

```
int& refA = A;                              // reference to variable
void fct(int&);                             // reference as function argument
int& fct();                                 // reference as function return
```

What is the difference between array and pointer (as both of them refer to starting address)?

```
T obj;
char  ac[10];                               //  ac = &obj + sizeof(T)              (compile time const address)
char* pc = new char[10];                    // &pc = &obj + sizeof(T) + sizeof(ac)  (compile time const address)
                                            //  pc = address pointing to heap       (runtime determined address)
```

What is the difference between pointer and reference ?

- pointer can be null value, reference must point to an object
- pointer can be reassigned, reference must be initialized and cannot be reassigned (or rebinded)

→ reference can be initialized on object declaration or in member initializer list

Function pointer and member pointer

How to define / declare / initialize / invoke function pointer and member pointer? See `std::function` in C++11.doc

```
// function pointer                                                              correspondence ...
int (*f)(int, const std::string&) = &fct;   // declare and init
int (*f)(int, const std::string&) =  fct;   // declare and init
(*f)(123, "abc");                           // invoke

// data member pointer
int A::*dp = &A::data;                      // declare and init
objA  .*dp = 123;                           // invoke via object
ptrA ->*dp = 123;                           // invoke via pointer

// member function pointer
int(A::*fp)(int,int) = &A::function;        // declare and init
(objA .*fp)(123, "abc");                    // invoke via object
(ptrA->*fp)(123, "abc");                    // invoke via pointer
```

For data member pointer and member function pointer, template class `std::mem_fn` offers a nice and standard syntax as below :

```
// data member pointer using std::mem_fn
auto d_fn = std::mem_fn(&A::data);          // auto declare and init
d_fn (objA) = 123;                          // invoke via object
d_fn(*ptrA) = 123;                          // invoke via pointer

// member function pointer using std::mem_fn
auto f_fn = std::mem_fn(&A::function);      // auto declare and init
f_fn (objA, 123, "abc");                    // invoke via object   (no complicated  .* operator)
f_fn(*ptrA, 123, "abc");                    // invoke via pointer  (no complicated ->* operator)
```

where ... `template<typename RET, typename T> std::mem_fn(RET T::* x);`

Three things about functions

- function taking reference to array as input
- function namespace lookup by "Argument-dependent-lookup"
- macro function

```
struct A
{
    std::uint32_t x;
    std::uint32_t y;
    std::uint32_t z;
};
void fct(const A (&array)[10]); // pass fixed-size const array by reference

// *** Test *** //
A aa[5];

aa[0] = {1,2,3};
aa[1] = {11,12,13};
...
aa[9] = {91,92,93};
fct(aa);

std::cout << std::is_same<decltype(aa), A[4]>::value;  // false
std::cout << std::is_same<decltype(aa), A[5]>::value;  // true
std::cout << std::is_same<decltype(aa), A[6]>::value;  // false
```

```
A   variable [10] =  instance of array of 10 elements
A (&variable)[10] = reference to array of 10 elements
```

For global function `fct` declared in certain namespace `ns0`, we can invoke `fct` without specifying its namespace. Compiler will start a ***Argument-dependent-lookup***, which looks for `fct` in the same namespace of its arguments, that is `ns0::A`, `ns1::B` and `ns1::C`.

```
ns0::A a;
ns1::B b;
ns2::C c;
fct(a,b,c); // no need to call ns0::fct
```

Macro is simply text substitution. The following is a macro snippet, which is not a complete function nor a complete class :

```
#define fct(ARG0, ARG1)                    \
    S s;                                   \  //   ARG0 can be function, class, object, member etc
    s.ARG0 = create_##ARG0();;             \  // ##ARG0 can be "parts" of function, class, object, member etc
    s.ARG1 = create_##ARG1##_instance();   \  // ARG1## can be "parts" of function, class, object, member etc
                                           \
    std::map<std::string, std::uint32_t> m;  \
    m[#ARG0] = hash_fct(s.ARG0);           \  //  #ARG0 is the string, no expliciy double quote needed
    m[#ARG1] = hash_fct(s.ARG1);           \
```

When substitute with `fct(vec, str)`, it becomes :

```
S s;
s.vec = create_vec();
s.str = create_str_instance();

std::map<std::string, std::uint32_t> m;
m["vec"] = hash_fct(s.vec);
m["str"] = hash_fct(s.str);
```

IF those `#` and `##` are removed, ambiguity happens like follow :

```
S s;
s.vec = create_ARG0();          // It won't subsitute part of a word.
s.str = create_ARG1_instance(); // It won't subsitute part of a word.

std::map<std::string, std::uint32_t> m;
m[vec] = hash_fct(s.vec);       // Compile error if vec doesnt exist.
m[str] = hash_fct(s.str);       // Compile error if str doesnt exist.
```

Marco vs template

- macro : by preprocessor, a simple text substitution, compiler error on macro-expanded code (rather than on macro itself)
- template : by compiler, a turing-complete language, compiler error on type checking

Memory copying functions

Function `memcpy` copies the exact number of bytes from source to destination without checking null character. Function `strcpy` copies unknown number of bytes from source to destination until (including) null character is detected, it will crash when null character is not found. Function `strncpy` copies the exact number of bytes from source to destination, if null character is detected, it is copied too, however the rest of destination string is set zero. All three functions return destination pointer.

```
#include<cstring>
void* memcpy (void* dst, const void* src, size_t num_bytes);
void* memmove(void* dst, const void* src, size_t num_bytes); // involves 2 copies, from src to intermediate, then to dst
char* strcpy (char* dst, const char* src);
char* strncpy(char* dst, const char* src, size_t num_bytes);
```

A4. Stack vs Heap / Free store?

Stack (or call stack) is a region in computer memory for storing temporary variables created by active function, including :

- return address
- function's input arguments and
- function's local variables in a LIFO manner.

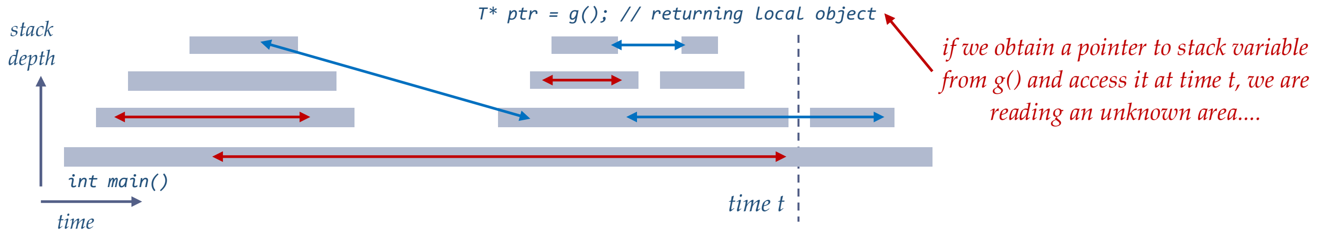
When an active subroutine comes to an end, call stack is popped, destructing local variables returning control to caller according to return address. Sequence of stack pop is called stack unwinding.

Heap (free store) is another region of computer memory for :

- dynamic allocation in runtime
- through calling `malloc/free` `new/delete` operator
- variables in heap can be accessed anywhere in the program, as long as the address or pointer is known.

Difference between heap and free store is conceptual (rather than physical) and depends on implementataion

- heap is for `malloc/free`, no constructor or destructor is called, common practice to `malloc` plus placement `new`
- free store is for `new/delete`, constructor and destructor are called



Stack	Heap / Free store
allocate on declaration of variable	allocate by `malloc/new` operator
deallocate on going out of scope	deallocate by `free/delete` operator
predetermined lifetime	runtime determined lifetime
predetermined allocated size	runtime determined allocated size
faster access (due to LIFO access pattern)	slower access (due to complicated bookkeeping)
each thread has its own stack	all threads share the same heap
no leakage	risk of leakage
no fragmentation	risk of fragmentation (due to repeated `new` and `delete`)

When to allocate on stack / heap (see figure in next page)?

- when variable lives within the same stack frame (including its children frames on top), use stack variable
- when variable lives across multiple stack frames, use heap variable
- such as creating learn-record (in ASM vision) by learn function, use the record else where by inspection function
- such as factory pattern, generating objects which are used somewhere else (these patterns happen a lot in *OOP*)
- however, there are different pointers available for managing different lifetime / ownership :
- for no ownership, but shared access to stack variable, use raw pointer
- for unique ownership to heap variable, use unique pointer
- for shared ownership to heap variable (i.e. pass it around), use shared pointer

How to prevent heap fragmentation?

Method 1 is to manage our own heap memory. There are two occasions in which we need dynamic allocation :

- objects that live across stack frames (blue arrows)
- polymorphic objects that live with in the same stack frame and below (red arrows)
- runtime variable size array that live within the same stack frame and below (red arrows)
- ► The second and third occasions can be done by customized memory manager using stack memory, such as `asm_malloc`.

Method 2 is to construct our own allocator for heap memory which is partitioned into regions with size of 2^N, it returns the smallest partition that is larger enough to satisfy user's request. It reduces the chance of fragmentation at the expense of extra memories.

Method 3 is to void the abuse of smart pointer and node based container. Heap allocation isnt *LIFO* is nature, overuse of which will inevitably results in fragmentation, it happens for `shared_ptr` and `std::list`.

B. Class and Object

B1. Copy construction and copy assignment

(1) Passing object as function argument by value invokes copy constructor, and copy constructor has to take reference to its type as input, or infinite recursion occurs. Make constructor private and implement factory pattern, if we want to restrict user to instantiate object using factories only. Make copy constructor and assignment operator private if we want to make a class non-copyable.

(2) In order to avoid self assignment, we have two solutions, self assignment checking and new-swap-delete approach *(this is not an official name)*. The former is a little bit faster when self-assignment happens, however it offers basic exception safety, while the latter offers strong exception safety, i.e. the latter retains its internal states when exception is thrown during memory allocation, and thus it is safer. The reason for such a difference is that the former invokes `delete` before `new`, whereas the latter invokes `new` before `delete`. Suppose there exists a resource class `RESOURCE` inside class `T` called `T::ptr` :

```
// method 1 : basic exception safety
T& T::operator=(const T& rhs)
{
    if (this!=&rhs)
    {   delete ptr; // firstly delete
        ptr = new RESOURCE(*rhs.ptr); // then new
        return *this;
    }
}
```

```
// method 2 : strong exception safety
T& T::operator=(const T& rhs)
{
    RESOURCE* tmp = new RESOURCE(*rhs.ptr); // firstly new
    std::swap(tmp, ptr);
    delete(tmp); // then delete
    return *this;
}
```

(3) Comparison of 6 fundamental members (`DC/CC/CA/MC/MA/DD`) among various containers

	T with ptr		mov-array<T>	unique-ptr<T>	shared-ptr<T>
		DC =	NEW	NEW	NEW
CA(nonsafe)	DEL NEW CPY RET	CC =	NEW CPY		LHS ++n
CA(safe)	NEW SWP DEL RET	CA =	DEL NEW CPY RET		--n LHS ++n RET
		MC =	LHS RHS	LHS RHS	LHS RHS
		MA =	DEL LHS RHS RET	DEL LHS RHS RET	--n LHS RHS RET
		DD =	DEL	DEL	--n
		CA+MA =	CPY SWP RET		
	C++03.doc		rvalue.doc	rvalue.doc	C++11.doc

(4) Rules governing these 6 functions and other member functions :

- a class must be either declared `final` (if it does not have derived) or its destructor declared `virtual` (if it may have derived)
- 6 basic functions must be in one of three cases : user-defined, `=default` or `=delete`, with public access for all three cases
- member functions should be declared `const` or `noexcept` when appropriate

(5) About default constructibility

- when a class has a constant member or reference member, custom constructor has to be provided
- when a class has a custom constructor, then its default constructor is automatically delete, that is :

`T::T()=default;` automatically becomes `T::T()=delete;`

(6) About keyword in function

Declaration in header	`static / const / noexcept`
	`explicit`
	`inline / virtual / override / final`

Definition in cpp	`static / const / noexcept` ~~`explicit`~~ `inline / virtual /` ~~`override`~~ `/` ~~`final`~~

Blue keywords are those that must be present in definition. Deleted keywords are those that must not present in definition. Others are just dont care.

B2. Conversion

Let's discuss one by one.

- Explicit cast vs implicit cast
- Casting operators
- Conversion constructor / conversion assignment / conversion operator
  Default initialization / direct initialization / copy initialization

(i) Explicit cast vs implicit cast

Object is stored as a sequence of bytes in memory, which equals to total size of all data members, excluding member functions and static members. It is **type** of the object which tells compiler how to interpret the byte stream. Changing the way compiler interprets the byte stream is called type conversion, it can be explicit or implicit. The former is done explicitly by casting operator `static_cast`, the latter is done automatically by compiler, when passing type `X` object to function accepting type `T` as argument and if there exists either non `explicit` conversion constructor or conversion operator, it will convert from `X` to `T`. Both explicit and implicit conversions will invoke conversion constructor or conversion operator with different precedence, please see below.

(ii) 4 casting operators, explicit keyword and downcasting

- `const_cast` adds or removes constness to or from a variable
- `static_cast` is explicit conversion that invokes conversion constructor / conversion operator
- `dynamic_cast` is used in polymorphism, which casts pointer or reference to base class **down** in the same inheritance hierarchy
- `reinterpret_cast` reinterprets the byte sequence as new type, you can get the original object by casting it backward
- `explicit` forbids compiler from using particular conversion constructor or conversion operator in implicit conversion

Name one usage of const cast

Suppose we have constant member, which is big and needed to be initialized with a separate `init()` function, we would :

- declare it `const` for 90% if its accesses, but `const_cast` inside `init()` function, rather than ...
- declare it `non-const` for 100% of its accesses, without evil `const_cast`, regardless of it is a constant by nature
- `const_cast` can be implemented as pointer or as reference

```
class algo
{
public:
     void init(const OBJ& input) // once and for all initialization
     {
          OBJ* px = const_cast<OBJ*>(&x); px = input; // please verify this syntax in gcc
          OBJ& ry = const_cast<OBJ&>(y);  ry = input; // please verify this syntax in gcc
     }

     void fct_accessing_const_xy() const { ... } // main usage

private:
     const OBJ x;
     const OBJ y;
};
```

Name one usage of downcast

Downcasting of pointer to base to pointer to derived is not that evil, if you can do it in a safe way, for example, including `type_id` as class member, initialize it in every derived class constructor, each assigns a different value, downcast before checking the `type_id`. In this case, we can either perform a `static_cast` of pointer, which crashes if it is downcasted to an incorrect derived class, or perform a `dynamic_cast` of pointer, which does runtime check to see if the downcast is valid, and returns `NULL` if it is downcasted to an incorrect derived class, hence `dynamic_cast` takes more time. In case we have `type_id` as class member, we prefer `static_cast` of pointer.

```
enum TYPE { EVENT_TYPE_A, EVENT_TYPE_B, EVENT_TYPE_C };
class event                { event (TYPE type_id)  : id(type_id)         { ... } };
class eventA : public event { eventA(X x, Y y, Z z) : event(EVENT_TYPE_A) { ... } };
class eventB : public event { eventB(X x, Y y)      : event(EVENT_TYPE_B) { ... } };
class eventC : public event { eventC(X x)           : event(EVENT_TYPE_C) { ... } };
```

For the sake of low latency and safety, we make a tradeoff : do `dynamic_cast` for debug only (`assert` is skipped in production).

```
//   #define NDEBUG // uncomment this line for production
     void process_eventA(event* ptr)
     {
          assert(dynamic_cast<eventA*>(ptr)!=nullptr); // if ptr points to eventB or eventC in debug mode, there will be runtime error
          eventA* ptrA = static_cast<eventA*>(ptr);
//        eventA* ptrA = (eventA*)(ptr); // Is this even faster?

          ptrA->mem_specific_to_A_only = ...
     }
```

(iii) 6+3 basic members / 6 invocations

Consider class, apart from 6 basic members `(DC CC CA MC MA DD)`, we add 3 more for conversion from `X` to `T` :

- conversion constructor of type `T` is a unary constructor with type `X` as argument
- conversion assignment of type `T` is an assignment operator with type `X` as argument
- conversion operator of type `X→T` is is a nullary operator named as target type `T`, having no return type but return statement

```
struct X;
struct T // only CC and CA are shown below
{
    T(const T&)            { std::cout << “copy constructor”;                      }
    T& operator=(const T&) { std::cout << “copy assignment”;       return *this; }
    T(const X&)            { std::cout << “conversion constructor”;                } // function 1 for conversion
    T& operator=(const X&) { std::cout << “conversion assignment”; return *this; } // function 2 for conversion
};

struct X
{
    operator T()           { std::cout << “conversion operator”;   return T{};   } // function 3 for conversion
};

void alg(const T&);

int main()
{
    X x;
    std::cout << “test0  Direct initialization”;    T t0(x);
    std::cout << “test1  Copy   initialization”;    T t1 = x;
    std::cout << “test2  Assignment ”;                t1 = x;
    std::cout << “test3  Explicit cast ”;           T t2 = static_cast<T>(x);
    std::cout << “test4  Explicit cast ”;             t2 = static_cast<T>(x);
    std::cout << “test5  Implicit cast ”;           alg(x);
}

// output
test0  Direct initialization    > conversion constructor
test1  Copy   initialization    > conversion operator    > default constructor
test2  Assignment               > conversion assignment
test3  Explicit cast            > conversion constructor
test4  Explicit cast            > conversion constructor > copy assignment
test5  Implicit cast            > conversion operator    > default constructor

// output, when conversion operator is removed
test0  Direct initialization    > conversion constructor
test1  Copy   initialization    > conversion constructor
test2  Assignment               > conversion assignment
test3  Explicit cast            > conversion constructor
test4  Explicit cast            > conversion constructor > copy assignment
test5  Implicit cast            > conversion constructor

// output, when conversion constuctor is removed
test0  Direct initialization    > conversion operator > default constructor
test1  Copy   initialization    > conversion operator > default constructor
test2  Assignment               > conversion assignment
test3  Explicit cast            > conversion operator > default constructor
test4  Explicit cast            > conversion operator > default constructor > copy assignment
test5  Implicit cast            > conversion operator > default constructor

// output, when both conversion constuctor and conversion operator are removed
compile error for test 0,1,3,4,5
```

How to construct T with X ?	Initialization	explicit vs implicit cast
`T t(x);`	direct initialization	
`T     t = x;`	copy initialization	
`T t; t = x;`	conversion assignment	
`T     t = static_cast<T>(x);`		explicit cast
`T t; t = static_cast<T>(x);`		explicit cast and copy assignment
`alg(x); // alg(const T&)`		implicit cast

Main difference between direct-initialization and copy-initialization is that they trigger different resolution mechanisms :

- direct initialization and explicit cast prefer conversion constructor to conversion operator
- copy initialization and implicit cast prefer conversion operator to conversion constructor
- conversion assignment is not involved in the resolution

C. Operator

Arity (num of operands), associativity (priority when cascading the same operator), precedence (priority among different operators) of an operator can't be changed. It can be declared as member function or global function, when declared as member function, `this` pointer is assumed to be the first argument. Operator cannot be static, as object is necessary to invoke operator.

operators are invoked by either	`obj>>x;` `obj.operator>>(x);`	`// for member function`
	`obj>>x;` `operator>>(obj,x);`	`// for global function`
operators that don't allow overload	`.  .*  ::  ?:`	
operators that must be member fct	`+= -= *= /= () [] -> =`	
operators that must be global fct	`+  -  >> <<`	

prefix increment (returns lvalue)	`T& T::operator++()     {`	`incre(*this); return *this; }`
postfix increment (returns rvalue)	`T  T::operator++(int) { T copy(*this);`	`incre(*this); return  copy; }`
prefix increment is invoked by	`++x; or x.operator++();`	`// we do   have (++x)+=3`
postfix increment is invoked by	`x++; or x.operator++(0);`	`// we dont have (x++)+=3`

function call operator (objects = functor)	`R T::operator()() {}`	`// nullary`
	`R T::operator()(A arg) {}`	`//   unary`
	`R T::operator()(A1 arg1, A2 arg2) {}`	`//  binary`
conversion operator (from `T` to `U`)	`T::operator U() {}`	`// nullary, return nothing, not even void`
conversion operator is invoked by	`objU = static_cast<U>(objT);`	
	`objU = objT.operator U();`	

New and delete operator

What are the differences between `new` and `malloc`?

- `new` operator invokes constructor while `malloc` does not
- `new` operator throws when not enough memory while `malloc` returns `NULL`

What are differences between `new` and array `new`? Lets do an experiment.

```
char buffer[1024];
struct my_class
{
    my_class() { std::cout << "constructor"; }
   ~my_class() { std::cout << " destructor"; }
    my_class(std::uint32_t A, std::uint32_t B, std::uint32_t C) : a(A), b(B), c(C) { std::cout << "constructor-ABC"; }

    void* operator new      (size_t n)  { std::cout << "new      operator, size=" << n; return buffer; }
    void* operator new[]    (size_t n)  { std::cout << "new[]    operator, size=" << n; return buffer; }
    void  operator delete   (void* ptr) { std::cout << "delete   operator "; }
    void  operator delete[] (void* ptr) { std::cout << "delete[] operator "; }
    void  debug() const                 { std::cout << a << b << c; }

    std::uint32_t a=1;
    std::uint32_t b=2;
    std::uint32_t c=3;
};

auto* p0 = new my_class;                // new operator, size=12 >> constructor
p0->debug();                            // 1,2,3
delete p0;                              // destructor >> delete operator

auto* p1 = new my_class[1]{{11,12,13}}; // new[] operator, size = 20 >> constructor-ABC
p1->debug();                            // 11,12,13
delete[] p1;                            // destructor >> delete [] operator

auto* p2 = new my_class[4]{{11,12,13},{21,22,23},{31,32,33},{41,42,43}}; // new[] operator, size = 56 >> constructor-ABC x 4times
for(int n=0; n!=4; ++n) p2[n].debug();                                  // 11,12,13,21,22,23,31,32,33,41,42,43
delete[] p2;                                                            // destructor x 4times, delete [] operator
```

- Why are the allocation sizes inconsistent?
  There are 8 bytes **meta-data** at the beginning used to indicate the size of array. Thus `8+12*1=20` and `8+12*4=56`.
- How do `delete` and `delete[]` know the number of times destructor should be called?
  operator `delete` corresponds to operator `new`, it releases memory starting from `T* ptr` with size `sizeof(T)`
  operator `delete[]` corresponds to operator `new[]`, it releases memory starting from `ptr-8` with size `sizeof(T)*N+8` and invoke destructor for `N` times, corresponding to `sizeof(T)*n+8` for all `n=[0,N-1)` where `N=*(ptr-8)` is **meta-data**
  it is developer's responsibility to match `new` with `delete` and `new[]` with `delete[]`, otherwise crash

Let's verify my speculation by overwriting the size of array. Please pay attention to the **ADDRESS !!!**

```
auto* p3 = new my_class[4];                   // new[] operator, size = 56 >> constructor-ABC x 4times
for(int n=0; n!=4; ++n) p3[n].debug();        // 1,2,3,1,2,3,1,2,3,1,2,3
std::cout << std::hex << (std::uint64_t)buffer; // 7fa28604b160  (8 bytes) = address returned by operator new[]
std::cout << std::hex << (std::uint64_t)(p3);   // 7fa28604b168 (12 bytes) = address stored by p3
std::cout << std::hex << (std::uint64_t)(p3+1); // 7fa28604b174 (12 bytes)
std::cout << std::hex << (std::uint64_t)(p3+2); // 7fa28604b180 (12 bytes)
std::cout << std::hex << (std::uint64_t)(p3+3); // 7fa28604b18c (12 bytes)
*(reinterpret_cast<std::uint64_t*>(buffer)) = 3;
delete[] p3;                                  // destructor x 3times, delete [] operator
```

What are the differences between no-throw `new` and placement `new`?

- no-throw `new` returns `nullptr` instead of throwing exception in case of failure
- no-throw `delete` does what normal delete does
- placement `new` allows construction object on pre-allocated heap memory or even stack memory, besides ...
- \- it decouples memory allocation from construction
- \- it decouples deallocation from destruction
- placement `delete` does nothing at all (it does not invoke destructor, nor free memory)

```
// nothrow new
T* p0 = new (std::nothrow) T;                 // no dash inside std::nothrow
T* p1 = new (std::nothrow) T(x,y,z);
delete p0;
delete p1;

// placement new for stack mem and
// placement new for heap mem and
char stack_buf[3*sizeof(T)];                  // only stack memory allocation, no invocation of T constructor
char* heap_buf = new char[3 * sizeof(T)];     // only  heap memory allocation, no invocation of T constructor
T* p0 = new (stack_buf + 0 * sizeof(T)) T;    // invocation of T constructor is delayed to here ...
T* p1 = new (stack_buf + 1 * sizeof(T)) T(x,y,z);
T* p2 = new (stack_buf + 2 * sizeof(T)) T(*p1);
T* pA = new ( heap_buf + 0 * sizeof(T)) T;
T* pB = new ( heap_buf + 1 * sizeof(T)) T(x,y,z);
T* pC = new ( heap_buf + 2 * sizeof(T)) T(*pB);
p0->print(); p1->print(); p2->print();
pA->print(); pB->print(); pC->print();
p0->~T();    p1->~T();    p2->~T();           // requires explicit invocation of destructor
pA->~T();    pB->~T();    pC->~T();           // no need to call : delete p0, delete p1 (as memory are not from heap)
delete[] heap_buf;
```

Both no-throw new and placement new allow array-form, constructor is invoked for multiple times :

```
// constructor is invoked for 10 times
T* p0 = new T[10];
T* p1 = new (std::nothrow) T[10];
T* p2 = new (stack_buf) T[10];
delete [] p0;
delete [] p1;
for(int n=0;n!=10;++n) p2+n->~T();
```

There are 6 new operators and 6 delete operators in total. All of them can be overloaded.

Please be reminded that the address returned by `operator new[]` differs from the address kept by pointer variable (like `p2` below) by 8 bytes for storing the number of elements.

```
void* T::operator new     (std::size_t num_bytes) throw (std::bad_alloc);
void* T::operator new     (std::size_t num_bytes, const std::nothrow_t&) throw();
void* T::operator new     (std::size_t num_bytes, void* ptr) throw();
void* T::operator new[]   (std::size_t num_bytes) throw (std::bad_alloc);
void* T::operator new[]   (std::size_t num_bytes, const std::nothrow_t&) throw();
void* T::operator new[]   (std::size_t num_bytes, void* ptr) throw();
void T::operator delete   (void* ptr) throw();
void T::operator delete   (void* ptr, const std::nothrow_t&) throw();   // does the same thing as delete
void T::operator delete   (void* ptr, void* ptr2) throw();              // does nothing
void T::operator delete[] (void* ptr) noexcept;
void T::operator delete[] (void* ptr, const std::nothrow_t&) noexcept;  // does the same thing as delete []
void T::operator delete[] (void* ptr, void* ptr2) noexcept;             // does nothing

// example
struct T
{
    T() { std::cout << "\nT::T()"; }
    void* operator new  (size_t sz) { std::cout << "\nT::new   " << sz << “bytes”; } // static by default
    void* operator new[](size_t sz) { std::cout << "\nT::new[] " << sz << “bytes”; } // no static declaration
    unsigned short m;
};
T* p1 = new T;          // print T::new    2 bytes, T::T()
T* p2 = new T[5];       // print T::new[] 10 bytes, T::T() T::T() T::T() T::T() T::T()
```

What are the differences between placement new and reinterpret cast?

- placement `new` and `reinterpret_cast` are low latency techniques for conversion between *byte array* and *aggregate structure*
- placement `new` allows construction of *aggregate structure* on already *allocated byte array* without extra copy nor move
- `reinterpret_cast` allows getting/setting *byte array* as if it is an *aggregate structure* without extra copy nor move
- both methods work no matter if there is zero padding in the *aggregate structure*, recall the use of `#pragma pack(push 1)`

```
struct S
{
    S(std::uint8_t N) : size(N)
    {
        std::uint8_t n = N % 3;
        if      (n == 0) memcpy(ac, "abcd", 5); // '\0' is added
        else if (n == 1) memcpy(ac, "ijkl", 5);
        else             memcpy(ac, "wxyz", 5);
    }
    std::uint8_t size;
    char ac[5];
};

#pragma pack(push, 1) // ensure no padding
struct T
{
    T(std::uint8_t N) : n1(N), n2(n1*n1), n4(n2*n2), as{ S(N),S(N+1),S(N+2) } {}
    std::uint8_t  n1;
    std::uint16_t n2;
    std::uint32_t n4;
    S as[3];
};
#pragma pack(pop)

template<typename X> void print(const X& x)
{
    std::cout << "\nn1=" << x.n1 << x.n2 << x.n4 << x.as[0].ac << x.as[1].ac << x.as[2].ac;
}

char impl[100];
new (impl) T(0); print(*reinterpret_cast<T*>(impl));
new (impl) T(1); print(*reinterpret_cast<T*>(impl));
new (impl) T(2); print(*reinterpret_cast<T*>(impl));
new (impl) T(3); print(*reinterpret_cast<T*>(impl));
```

Literal operator

Literal operator offers a convenient way to construct classes. In the following example, objects are constructed from literals :

```
struct distance
{
    distance(double x) : m(x){}
    distance operator+(const distance& rhs) { return distance{ m+rhs.m }; }
    distance operator-(const distance& rhs) { return distance{ m-rhs.m }; }
    distance operator*(double n)            { return distance{ m*n }; }
    distance operator/(double n)            { return distance{ m/n }; }
    double m;
};

distance average(const std::initializer_list<distance>& list)
{
    distance sum{0};
    for(const auto& x:list) sum = sum + x;
    return sum / list.size();
}

distance operator""_cm(long double x) { return distance(x) / 100;  }
distance operator""_m (long double x) { return distance(x);        }
distance operator""_km(long double x) { return distance(x) * 1000; }

std::ostream& operator<<(std::ostream& os, const distance& x)
{
    os << "distance " << x.m << " meters";
    return os;
}

// in practice, please :
// 1. add namespace and
// 2. make distance constructor explicit (avoid mistakenly convert double as distance)
std::cout << "\n" << 1.23;                    -> 1.23 (just a double)
std::cout << "\n" << 1.23_cm;                 -> 0.0123 meters
std::cout << "\n" << 1.23_m;                  -> 1.23 meters
std::cout << "\n" << 1.23_km;                 -> 1230 meters
std::cout << "\naverage = " << average({ 1.23, 1.23, 1.23 });            -> 1.23 meters (mistakenly construct as distance)
std::cout << "\naverage = " << average({ 1.23_cm, 1.23_m, 1.23_km });      -> 410.414 meters
```

I try to use `double` instead of `long double` in all literal operators, however it results in compilation error. Is this a must?

D. Exception

D1. What is exeption?

Exception is :

- a signalling mechanism for runtime error that cannot be handled locally
- it separates normal code from error handling code, thus avoiding spaghetti code, improving readability and maintainability
- it is particularly useful for RAII

D2. Using exception

It includes 3 parts :

- defines our exception classes,
- throws exception object when runtime error that cannot be handled locally is detected and
- invokes functions that may throw exception in try-block, handles exception in catch-block

```
// (Step 1) define exception class
class my_ex0 : public std::exception;
class my_ex1 : public std::exception;
class my_ex2 : public std::exception;

// (Step 2) throw exceptions that it cannot handle
void alg0()
{
    if (...) throw my_ex0{};
    if (...) throw my_ex1{};
    if (...) throw my_ex2{};
}
void alg1() throw(my_ex0, my_ex1) // remark 1
{
    if (...) throw my_ex0{};
    if (...) throw my_ex1{};
}
void alg2() noexcept; // remark 2

// (Step 3) implement try-and-catch block
try
{
    f();
}
catch(const my_ex0& e) { throw; }                                // remark 3 : rethrow
catch(const my_ex1& e) { std::cout << "exception1"; }
catch(...)             { std::cout << "exception2 or others"; }  // remark 4 : catch all exceptions
```

D3. Don't throw when …

- do not throw if error can be solved locally
- do not catch if error cannot be solved in catch block
- do not use exception like goto, nor as a way to pass objects across call stack
- do not use exception to detect coding error, use assertion instead, which terminates program when condition is false

```
assert(bool_condition);
```

D4. Stack unwinding

Exception throwing mechanism works like this :

- when a function throws exception, call stack is poped, local objects are destructed, program control goes to the caller
- if caller lies in try-block, rest of the try-block is skipped, local objects are destructed, program control goes to catch block
- if exception matches specification in catch-block, it is then handled in catch-block
- if caller does not lie inside try block, or if exeption does not match :
- keep unwinding until it is handled in a catch-block, `std::terminate()` is invoked with no one does

This process is known as stack unwinding.

- `std::terminate()` is invoked when another exception is thrown during stack unwinding
- `std::unexpected()` is invoked when exception specification is violated
- callback for both `terminate()` and `unexpected()` can be customized by :

```
void (*)() set_terminate (void (*)());
void (*)() set_unexpected(void (*)());
```

D5. Remarks

FAQ1 : Exception in constructor and destructor

Since constructor does not return, exception is the only way to signal an error during construction, it is particularly useful for RAII. However, when constructor throws, destructor will not be invoked (as construction of object is incomplete). The partly constructed object should be undone manually. How about destructor? Negative, if some destructors do throw, then they may be called during stack unwinding (if they have instances as local objects in the function that starts the unwinding), as a result, one exception leads to exponential growth in exceptions, thus in order to avoid this from happening, `std::terminate()` is invoked when another exception is thrown during stack unwinding. What should we do when destructor fails? Write a message to log file and terminate the process.

- constructor does throw (the only way to notify caller in case of error)
- destructor never throw (result in multiple exceptions)

FAQ2 : What is exception safety?

Exception safety of a class or a function describe its behaviours in error-handling. There are four levels : (1) no-throw guarantee (i.e. class that always succeeds and never throws, it's the strongest level), (2) strong exception safety (i.e. it may fails, returning errors or throwing exceptions, yet its states are unchanged), (3) basic exception safety or no-leak guarantee (i.e. it may fail, and its states may be overwritten, but it is still a valid state, besides, there should be no memory nor resource leakages) and (4) no exception safety (i.e. there are leakages when it fails).

FAQ3 : How does exception separate normal code (good code) from error handling (bad code)?

Suppose `function1` may return `error1&2&3`, `function2` may return `error2&3&4`, `function3` may return `error3&4&1`, `function4` may return `error4&1&2`. Please note in this test, `exception1,2,3,4` are classes, while `error1,2,3,4` are integers.

```
// The exception way
try  {    function1();
          function2();
          function3();
          function4();
}
catch(const exception1& e1) { /* handle e1 */ }
catch(const exception2& e2) { /* handle e2 */ }
catch(const exception3& e3) { /* handle e3 */ }
catch(const exception4& e4) { /* handle e4 */ }

// The IF statement way
status = function1();
if(status == error1) { /* handle e1 */ return; }
if(status == error2) { /* handle e2 */ return; }
if(status == error3) { /* handle e3 */ return; }
status = function2();
if(status == error2) { /* handle e2 */ return; }
if(status == error3) { /* handle e3 */ return; }
if(status == error4) { /* handle e4 */ return; }
status = function3();
if(status == error3) { /* handle e3 */ return; }
if(status == error4) { /* handle e4 */ return; }
if(status == error1) { /* handle e1 */ return; }
status = function4();
if(status == error4) { /* handle e4 */ return; }
if(status == error1) { /* handle e1 */ return; }
if(status == error2) { /* handle e2 */ return; }
```

FAQ4 : Why do I have too many try-and-catch block?

This is because you are not sticking with the rule that : try-and-catch only when you know how to handle it.

D6. Signal handling in linux

6.1 What is signal?

- Signal is a software interrupt delivered from the OS to any process, usually used for error handling.
- Signal is a limited form of interprocess communication, asynchronous notification of occurred event.
- Each signal is 32 bit integer, thus it can represent 32 different signals.
- Some signals can be generated by keying input in controlling terminal of the process.
- Some signals can be caught and handle, while some cannot.

6.2 signal		generated by	caught and handled
`01. SIGHUP`	hang-up	when controlling terminal is closed / disconnected	yes
`02. SIGINT`	interrupt	enter `ctrl-c` in controlling terminal	yes
`03. SIGQUIT`	quit signal	enter `ctrl-d` or `ctrl-\` in controlling terminal	yes
`08. SIGFPE`	floating point exception	division by zero / overflow	yes
`09. SIGKILL`	last resort terminate process	command `kill -SIGKILL pid` or `kill -9 pid`	cannot be handled
`11. SIGSEGV`	segmentation violation	access outside virtual address space VAS	yes
`15. SIGTERM`	graceful terminate process	command `kill -SIGTERM pid` or `kill pid` default kill	yes
`18. SIGCONT`	continue	command `kill -SIGCONT pid`	yes
`20. SIGTSTP`	suspend to background	command `kill -SIGTSTP pid` or `ctrl-z` (can goto `fg` again)	`fg` = foreground

- we handle `2` and `15` in `YLib`
- we cannot catch and handle `9`, as it guarantees to kill a process

6.3 We can register callback to signal so that callback will be invoked in case signal occurs. This is how we register callback :

```
#include <csignal>

// C stype callback
void my_sig0_callback(int signal_id) { ... }
void my_sig1_callback(int signal_id) { ... }
void my_sig2_callback(int signal_id) { ... }

// C++ stype callback (does not work in my test)
class my_logger
{    public:
     static void sig0_callback(int signal_id) { ... }
     static void sig1_callback(int signal_id) { ... }
     static void sig2_callback(int signal_id) { ... }
}

// register callback
signal(SIGINT,  my_sig0_callback);
signal(SIGSEGV, my_sig1_callback);
signal(SIGTERM, my_sig2_callback);
signal(SIGINT,  my_logger::sig0_callback);
signal(SIGSEGV, my_logger::sig1_callback);
signal(SIGTERM, my_logger::sig2_callback);
```

Which thread invoke the signal handler?

If signal is raised by a thread in the process, the same thread will invoke the signal handler, like `boost::signal`.

If signal is raised by standard input like `ctrl-C`, *the main thread is interrupted, paused and invokes the handler.*

```
// send signal
std::raise(SIGINT);
std::raise(SIGSEGV);
std::raise(SIGTERM);
```

D7. Runtime assert and compile time static-assert

The following are runtime `assert` (requires `#include<assert.h>`) and compile time `static_assert` respectively :

- `assert` is a global function (all `assert` can be disabled by adding `#define NDEBUG` before `#include<assert.h>`)
- `static_assert` is a macro (not a function, no return value)

```
    #include<assert.h> // for assert
7.1 void   assert(int expression);           // if expression is 0 in runtime, cause runtime error : Abort (core dumped)
7.2 static_assert(bool_constexpr, message); // if expression is FALSE in compile time, cause compile failure
```

Here is an example (tested in `gcc`) :

```
//  #define NDEBUG // disble all assert if this macro is added before "assert.h"
    #include<assert.h>
    std::uint32_t* ptr = nullptr;
    assert(ptr != nullptr); // output = Aborted (core dump)

//  static_assert(std::is_same<std::vector<std::uint32_t>::value_type, std::uint16_t>::value, "not the same"); // compile error
    static_assert(std::is_same<std::vector<std::uint32_t>::value_type, std::uint32_t>::value, "not the same");
```

7.3 Practical application of `assert`

- when we deal with pointer which is 100% non-null, then a null checking is probably a waste of time ...

```
// For safety, we do all necessary checking
void fct(RESOURCE* ptr)
{
     if (ptr!=nullptr)
     {
          ptr->mem = ...
     }
}
```

```
// Yet for low latency production, we check for debug mode only :
void fct(RESOURCE* ptr)
{
     assert(ptr!=nullptr);

     ptr->mem = ...

}
```

E. Inheritance

E1. What is Inheritance?

Build a class on top of another by obtaining members from an ancester. As opposed to composition …

- inheritance is a ***is-a*** relationship, composition is ***has-a*** relationship
- inheritance can access public and protected members, while composition can access only public members
- inheritance allows polymorphism, while composition does not.

E2. Three levels of access privileges

	inside base class definition		inside derived class definition		access via object
	as `this`	as `rhs`	as `this`	as `rhs`	
public base mem	yes [1]	yes [4]	yes [7]	yes [10]	yes [13]
protected base mem	yes [2]	yes [5]	yes [8]	yes [11]	no [14]
private base mem	yes [3]	yes [6]	no [9]	no [12]	no [15]

```
class base
{
    void fct(const base& rhs)
    {
        std::cout << x;       // [1]
        std::cout << y;       // [2]
        std::cout << z;       // [3]
        std::cout << rhs.x;   // [4]
        std::cout << rhs.y;   // [5]
        std::cout << rhs.z;   // [6]
    }
    public:     int x;
    protected:  int y;
    private:    int z;
};
class derive : public base
{
    void fct(const derive& rhs, const base& rhs0)
    {
        std::cout << x;       // [7]
        std::cout << y;       // [8]
    //  std::cout << z;       // [9]
        std::cout << rhs.x;   // [10]
        std::cout << rhs.y;   // [11]
    //  std::cout << rhs.z;   // [12]
        std::cout << rhs0.x;  // [13]
    //  std::cout << rhs0.y;  // [14]
    //  std::cout << rhs0.z;  // [15]
    }
};
```

```
private constructor makes a class non-constructible
private constructor makes a class non-inheritable
private  assignment makes a class non-copyable
```

E3. Three levels of inheritance

Level of inheritance determines the scopes having <u>right to know</u> about the inheritance, thus assigning base class pointer to object of derived class inside scope having no <u>right to know</u> will result in compile error. Suppose we have :

`class B : public A`	all classes know about inheritance `B:A` (polymorphism works in this mode)
`class B : protected A`	only `B` itself and its derived classes know about inheritance `B:A`
`class B : private A`	only `B` itself knows about inheritance `B:A`

Assigning protected or private derived class `B` to base pointer `A` outside the inheritance tree will result in compile error.

```
struct A   {};
struct B : xxx     A { void fct() { A* p = new B; } };
struct C : public B { void fct() { A* p = new B; } };
                      void fct() { A* p = new B; }
```

xxx =	public	protect	private
line x	ok	ok	ok
line y	ok	ok	error
line z	ok	error	error

E4. How access privilege progagates through inheritance?

	public inheritance	protect inheritance	private inheritance
public in base	public in derive	protect in derive	private in derive
protect in base	protect in derive	protect in derive	private in derive
private in base	private in derive	private in derive	private in derive

The following example shows how access privilege proceeds through inheritance hierarchy.

```
    class base {  public:     void display1() { std::cout << "\nbase public    member";}
                  protected:  void display2() { std::cout << "\nbase protected member";}
                  private:    void display3() { std::cout << "\nbase private   member";} };
    class derived1 : public    base { public: void test() { display1(); display2(); }};
    class derived2 : protected base { public: void test() { display1(); display2(); }};
    class derived3 : private   base { public: void test() { display1(); display2(); }};
//  class derivedA : public    base { public: void test() { display1(); display2(); display3(); }}; // compile error
//  class derivedB : protected base { public: void test() { display1(); display2(); display3(); }}; // compile error
//  class derivedC : private   base { public: void test() { display1(); display2(); display3(); }}; // compile error

    // All functions below are invoked via objects, hence no polymorphism kicks in.
    base     base_object;
    derived1 derived_object1;               derived_object1.test();
    derived2 derived_object2;               derived_object2.test();
    derived3 derived_object3;               derived_object3.test();

    derived_object1.display1();
//  derived_object1.display2();             // compile error : cannot access protected member
//  derived_object1.display3();             // compile error : cannot access private   member
//  derived_object2.display1();             // compile error : cannot access protected member
//  derived_object2.display2();             // compile error : cannot access protected member
//  derived_object2.display3();             // compile error : cannot access private member
//  derived_object3.display1();             // compile error : cannot access private member
//  derived_object3.display2();             // compile error : cannot access private member
//  derived_object3.display3();             // compile error : cannot access private member
```

E3. Application of protected and private inheritance

According to Alu, here are the usages of different inheritance :

- public inheritance is for is-a relationship
- protected inheritance is for implementator, with virtual functions defined in derived class
- private inheritance is for implementator, with virtual functions defined in derived-derived class
  (however I cannot search the word implementator in the web ... woo ... beware)

For public inheritance, we want the inheritance relation to be known by outside scope, we can declare base class pointer pointing to derived objects, we then operate the derived objects through the base class pointer, as if they are all the base class, as a result we are implementing a is-a relationship. On the contrary, for protected/private inheritance, we do not disclose the inheritance relationship to the outside world, hence disabling the is-a relationship. Logically, this is the same as composition. When do we choose protected or private inheritance to composition? We choose the former, when there is some virtual functions to be implemented in base class. The following two codes are logically the same ...

```
    // use protected or private inheritance
    class implementation : private interface
    {
        virtual void fct() override;
    };

    // use inheritance (when there is virtual functions in base class)
    class implementation
    {
        void fct() { impl.fct(); }
        interface_impl impl;
    };

    class interface_impl : public interface
    {
        virtual void fct() override;
    };
```

F. Polymorphism

F1. What is polymorphism?

1 Polymorphism is changing form of objects, its about resolving to different implementations according object's dynamic type.
2 Polymorphism is triggered by : (1) `virtual` declaration, (2) function overiding and (3) base class pointer to derived class.
3 Polymorphism is implemented by : (1) `vtable` in class, (2) `vptr` in object and (3) dynamic binding to resolve `virtual` functions.

When should we use polymorphism?

- when we have a container of base class pointers to different derived objects, i.e. heterogenous container, we then either :
- invoke pure `virtual` function, which is defined in concrete classes (called non-virtual interface *NVI*)
- invoke downcasting followed by concrete-class-specific-function, it is safe to do so if we provide `type()`, like `yubo::event`

```
struct event  { virtual int type() { _type = 0; }     int _type;              };
struct eventA { virtual int type() { _type = 1; }     void run_A_specific(); };
struct eventB { virtual int type() { _type = 2; }     void run_B_specific(); };

std::vector<std::unique_ptr<event>> vec;
for(auto x : vec)
{   // when used properly, like storing derived type as member, downcast can still be a safe move. Yet this is not polymorphism
    if (x->type() == 1) std::dynamic_pointer_cast<eventA>(x)->run_A_specific();
    if (x->type() == 2) std::dynamic_pointer_cast<eventB>(x)->run_B_specific();
}
```

F2. Static type and dynamic type

- Static type of an identifier is the type in its declaration, it is known and fixed in compile time.
- 2.1 Dynamic type of stack-allocated identifier is the same as static type.
- 2.2 Dynamic type of heap-allocated identifier is the type allocated by `new` operator.

```
base     b;                              // static_type(b)   = base       dynamic_type(b)   = base
derived  d;                              // static_type(d)   = derived    dynamic_type(d)   = derived
base*    pb0 = new base;                 // static_type(pb0) = base*      dynamic_type(pb0) = base*
base*    pb1 = new derived;              // static_type(pb1) = base*      dynamic_type(pb1) = derived*
derived* pb2 = new derived;              // static_type(pb2) = derived*   dynamic_type(pb2) = derived*
```

Given inheritance hierarchy `base→derived0→derived1`, having multiple member functions, which may either be `virtual` or `non_virtual`, recursively calling each other, forming a long invocation sequence. What are the static type and dynamic type in each step?

```
base* ptr = new derived1;
ptr->fct0();                             // static_type(ptr) = base*      dynamic_type(ptr) = derived1*
```

2.3 If we trace the call stack and reach the following line, the static type and dynamic type of `this` pointer are :

```
void derived0::fct0() { this->fct1(); }    // static_type(this) = derived0*     dynamic_type(this) = derived1*
```

Static type changes along the call sequence, depending on `this` pointer.
Dynamic type is fixed, depending on `vptr` in dynamic allocation.

F3. Static binding and dynamic binding

- Static binding is compile-time function resolution according to static type (declared type) of an identifier.
- Dynamic binding is runtime function resolution according to dynamic type (`vptr` type) of an identifier.

When is static binding / dynamic binding is triggered?

- 3.1 invoke `non_virtual` function via object ⇒ static binding
- 3.2 invoke `virtual` function via object ⇒ static binding
- 3.3 invoke `non_virtual` function via pointer or reference ⇒ static binding
- 3.4 invoke `virtual` function via pointer or reference ⇒ dynamic binding

F4. Dynamic binding algorithm with `vtable` and `vptr`

1 Compiler generates `vtable` for classes (not for objects) having `virtual` functions
- `vtable` is a map of `virtual` member function pointers to their implementations, it doesn't record `non_virtual` function
- `vtable` is inherited (exactly copied) to derived class
- `vtable` in derived class is updated if virtual member is overriden

2 `vptr` is assigned to object, which points to `vtable` of its dynamic type, `vptr` lives with the object until it is destructed

3 dynamic binding is triggered when `virtual` function is invoked via pointer, via two redirections :
- from `vptr` to `vtable` and
- from `vtable` to function pointer

F5. Overload, hiding rule, redefine and override

Overload means multiple functions sharing the same identifier, but having different signatures (including constness and excluding return type) and different implementations. Compiler resolves the overload according to caller arguments and overload signatures. Template function is just a general form of overload, with template type deduction working like overload resolution.

Overloading a base class member function with a function with same identifier but <u>different</u> signature in derived class will hide the base class function. **Hiding** happens regardless of `virtual` or `non-virtual` declaration. For example :

```
    class base                  { public: void fct(int x)        { std::cout << "A"; }};
    class derive : public base  { public: void fct(int x, int y) { std::cout << "B"; }};

    derive d;
//  d.fct(1);         // compile error
    d.fct(1,2);       // OK, print B
    d.base::fct(1);   // OK, print A
```

- **Redefine** means re-implementation of `non_virtual` base class member function with same identifier and same signature.
- **Override** means re-implementation of `virtual` base class member function with same identifier and same signature.
- As only `virtual` functions are recorded in `vtable`, redefine does not trigger polymorphism, while override does.

-	given `f(X,Y)`	then implementation of `f(X,Y,Z)` is overload
-	given `base::f(X,Y)`	then implementation of `derived::f(X,Y,Z)` is hidding
-	given `non_virtual base::f(X,Y)`	then implementation of `derived::f(X,Y)` is redefine
-	given `virtual base::f(X,Y)`	then implementation of `derived::f(X,Y)` is override
-	given `virtual base::f(X,Y)`	then no re-implementation means simple inheritance of `base::f(X,Y)`

F6. Default parameter in virtual function

Unlike virtual function resolution, which depends on dynamic type of caller, default parameter depends on static type of caller :

```
// Difference in resolution mechanism in virtual function and in default parameter is confusing, so try to avoid ...
struct B            { virtual void fct(int n=10) { std::cout << "B" << n; } };
struct D : public B { virtual void fct(int n=20) { std::cout << "D" << n; } };

B b;
D d;
B* pb0 = &b; pb0->fct(); // print B10 resolve "fct" according to dynamic type, resolve "n" according to static type
B* pb1 = &d; pb1->fct(); // print D10
D* pd  = &d; pd ->fct(); // print D20
```

Test 1 – Invocation via object vs invocation via pointer

```
    class base
    {
        public:    base()                                {    std::cout << "\nconstructor in base";          }
        virtual   ~base()                                {    std::cout << "\ndestructor in base";           }
                  void fct0()                            {    std::cout << "\nfct0 in base";                 }
        virtual   void fct1()                            {    std::cout << "\nfct1 in base";                 }
    };

    class derived : public base
    {
        public:    derived()                             {    std::cout << "\nconstructor in derived";       }
        virtual   ~derived()                             {    std::cout << "\ndestructor in derived";        }
                  void fct0()                            {    std::cout << "\nfct0 in derived";              }
        virtual   void fct1()                            {    std::cout << "\nfct1 in derived";              }
    };

    class derived_x2 : public derived
    {
        public:    derived_x2()                          {    std::cout << "\nconstructor in derived_x2";    }
        virtual   ~derived_x2()                          {    std::cout << "\ndestructor in derived_x2";     }
                  void fct0()                            {    std::cout << "\nfct0 in derived_x2";           }
        virtual   void fct1()                            {    std::cout << "\nfct1 in derived_x2";           }
    };

    base          baseObj,*basePtr = &baseObj;           // constructor in base
    derived       der1Obj,*der1Ptr = &der1Obj;           // constructor in base & derived
    derived_x2    der2Obj,*der2Ptr = &der2Obj;           // constructor in base & derived & derived_x2

//  [invoke fct via object]                              // The following are all static bind.
    baseObj.fct0();                                      // fct0 in base
    baseObj.fct1();                                      // fct1 in base
    der1Obj.fct0();                                      // fct0 in derived
    der1Obj.fct1();                                      // fct1 in derived
    der2Obj.fct0();                                      // fct0 in derived_x2
    der2Obj.fct1();                                      // fct1 in derived_x2

//  [invoke base fct via object]                         // The following are all static bind.
    der1Obj.base::fct0();                                // fct0 in base
    der1Obj.base::fct1();                                // fct1 in base
    der2Obj.derived::fct0();                             // fct0 in derived
    der2Obj.derived::fct1();                             // fct1 in derived
    der2Obj.base::fct0();                                // fct0 in base
    der2Obj.base::fct1();                                // fct1 in base
    der2Obj.derived::base::fct0();                       // fct0 in base
    der2Obj.derived::base::fct1();                       // fct1 in base
//  der2Obj.base::derived::fct0();                       // compile error
//  der2Obj.base::derived::fct1();                       // compile error

//  [invoke fct via pointer]
    basePtr->fct0();                                     // fct0 in base             static bind (fct0 is not in vtable)
    basePtr->fct1();                                     // fct1 in base             dynamic bind
    der1Ptr->fct0();                                     // fct0 in derived          static bind (fct0 is not in vtable)
    der1Ptr->fct1();                                     // fct1 in derived          dynamic bind
    der2Ptr->fct0();                                     // fct0 in derived_x2       static bind (fct0 is not in vtable)
    der2Ptr->fct1();                                     // fct1 in derived_x2       dynamic bind

//  [invoke base fct via pointer]
    der1Ptr->base::fct0();                               // fct0 in base
    der1Ptr->base::fct1();                               // fct1 in base
    der2Ptr->derived::fct0();                            // fct0 in derived
    der2Ptr->derived::fct1();                            // fct1 in derived
```

The above is not about polymorphism, it should be done with pointer to base class :

```
//  [polymorphism]
    base* basePtr1 = &der1Obj;
    base* basePtr2 = &der2Obj;
    basePtr1->fct0();                                    // fct0 in base             static bind (fct0 is not in vtable)
    basePtr1->fct1();                                    // fct1 in derived          dynamic bind
    basePtr2->fct0();                                    // fct0 in base             static bind (fct0 is not in vtable)
    basePtr2->fct1();                                    // fct1 in derived_x2       dynamic bind
    basePtr1->base::fct1();                              // fct1 in base             static bind (resolution is done explicitly)
    basePtr2->base::fct1();                              // fct1 in base             static bind (resolution is done explicitly)
//  basePtr2->derived::fct1();                           // compile error            ptr to base cannot see derived class

//  [invoke fct by downcasting]
    dynamic_cast<derived*>(der2Ptr)->fct0();             // fct0 in derived          static bind (change static type to derived)
    dynamic_cast<derived*>(der2Ptr)->fct1();             // fct1 in derived_x2       dynamic bind (change static type to derived)
    dynamic_cast<base*>   (der2Ptr)->fct0();             // fct0 in base             static bind (change static type to base)
    dynamic_cast<base*>   (der2Ptr)->fct1();             // fct1 in derived_x2       dynamic bind (change static type to base)
```

The dynamic casts mean changing the static type of `der2Ptr` to `derived` and `base` respectively, while dynamic cast cannot change the dynamic type of `der2Ptr` as it is fixed by `vtable` assignment in object construction during heap allocation.

Test 2 – Member function invoking another member function (more generic case, verified in both *MSVS* and *online gcc*)

```
struct base
{
             void fct0()   { std::cout << "\nbase::fct0";   fct2();   fct3(); }
    virtual  void fct1()   { std::cout << "\nbase::fct1";   fct2();   fct3(); }
             void fct2()   { std::cout << "\nbase::fct2";   }
    virtual  void fct3()   { std::cout << "\nbase::fct3";   }
};

struct derivedA : public base
{
             void fct0()   { std::cout << "\nderA::fct0”;   fc2();    fct3(); }
             void fct2()   { std::cout << "\nderA::fct2";   }
};

struct derivedB : public base
{
    virtual void fct1()    { std::cout << "\nderB::fct1";   fct2();   fct3(); }
    virtual void fct3()    { std::cout << "\nderB::fct3";   }
};

struct derivedC : public base
{
             void fct0()   { std::cout << "\nderC::fct0";   fct2();   fct3(); }
    virtual  void fct1()   { std::cout << "\nderC::fct1";   fct2();   fct3(); }
             void fct2()   { std::cout << "\nderC::fct2";   }
    virtual  void fct3()   { std::cout << "\nderC::fct3";   }
};

base      B;       base* pB  = new base;
derivedA  DA;      base* pDA = new derivedA;
derivedB  DB;      base* pDB = new derivedB;
derivedC  DC;      base* pDC = new derivedC;

B .fct0();         B .fct1();         B .fct2();         B .fct3();
DA.fct0();         DA.fct1();         DA.fct2();         DA.fct3();
DB.fct0();         DB.fct1();         DB.fct2();         DB.fct3();
DC.fct0();         DC.fct1();         DC.fct2();         DC.fct3();

pB ->fct0();       pB ->fct1();       pB ->fct2();       pB ->fct3();
pDA->fct0();       pDA->fct1();       pDA->fct2();       pDA->fct3();
pDB->fct0();       pDB->fct1();       pDB->fct2();       pDB->fct3();
pDC->fct0();       pDC->fct1();       pDC->fct2();       pDC->fct3();
```

Here are the results, a 8×8 matrix.

```
base::fct0    base::fct2    base::fct3    base::fct1    base::fct2    base::fct3    base::fct2    base::fct3
derA::fct0    derA::fct2    base::fct3    base::fct1    base::fct2    base::fct3    derA::fct2    base::fct3
base::fct0    base::fct2    derB::fct3    derB::fct1    base::fct2    derB::fct3    base::fct2    derB::fct3
derC::fct0    derC::fct2    derC::fct3    derC::fct1    derC::fct2    derC::fct3    derC::fct2    derC::fct3

base::fct0    base::fct2    base::fct3    base::fct1    base::fct2    base::fct3    base::fct2    base::fct3
base::fct0    base::fct2    base::fct3    base::fct1    base::fct2    base::fct3    base::fct2    base::fct3
base::fct0    base::fct2    derB::fct3    derB::fct1    base::fct2    derB::fct3    base::fct2    derB::fct3
base::fct0    base::fct2    derC::fct3    derC::fct1    derC::fct2    derC::fct3    base::fct2    derC::fct3
```

The resolution is as follows :

- for member function calling other member function, prefix all function all by `this` pointer
- the static type of `this` pointer is the class where `this` locates
- the dynamic type of `this` pointer is the class pointed by `vptr`
- if the function is not `virtual`, then do static binding, that is resolve by the invocation object / invocation pointer / `this` pointer
- if the function is `virtual`, then do dynamic binding, that is resolve by the type pointed by `vptr`

Test 3 – Function taking reference to class hierarchy as argument

When multi function overloads taking reference to base object and derived objects as argument like example below, polymorphism does not kick in. In fact it is got nothing to do with dynamic binding and `vtable`, it is simply overload resolution, it prefers to match with the declared type of input argument.

```
struct B {};
struct D0 : public B {};
struct class D1 : public B {};

void fct(const B&)          { std::cout << "\nfct-B";  }     // bind to B, D0, D1
void fct(const D0&)         { std::cout << "\nfct-D0"; }     // bind to D0 only
void fct(const D1&)         { std::cout << "\nfct-D1"; }     // bind to D1 only

B  b;                        B& rb  = b;
D0 d0;                       B& rd0 = d0;
D1 d1;                       B& rd1 = d1;

fct(b);  // fct-B            fct(rb);  // fct-B
fct(d0); // fct-D0           fct(rd0); // fct-B              hence it prefers fct(const B&) to fct(const D0&)
fct(d1); // fct-D1           fct(rd1); // fct-B              hence it prefers fct(const B&) to fct(const D1&)
```

The invoked `fct` above are global functions, now let's make them member functions inside another inheritance hierarchy.

```
struct BASE
{
    virtual void fct(const B&)  { std::cout << "\nBASE-B";  }
    virtual void fct(const D0&) { std::cout << "\nBASE-D0"; }
    virtual void fct(const D1&) { std::cout << "\nBASE-D1"; }
};

struct DERIVE : public BASE
{
    void fct(const B&)          { std::cout << "\nDERIVE-B";  }
    void fct(const D0&)         { std::cout << "\nDERIVE-D0"; }
    void fct(const D1&)         { std::cout << "\nDERIVE-D1"; }
};

B  b;                        B& rb  = b;
D0 d0;                       B& rd0 = d0;
D1 d1;                       B& rd1 = d1;
BASE   base;                 BASE*  pbase2base   = new BASE;
DERIVE derive;               BASE*  pbase2derive = new DERIVE;

base.fct(b);                 BASE-B                          base.fct(rb);                BASE-B
base.fct(d0);                BASE-D0                         base.fct(rd0);               BASE-B
base.fct(d1);                BASE-D1                         base.fct(rd1);               BASE-B
derive.fct(b);               DERIVE-B                        derive.fct(rb);              DERIVE-B
derive.fct(d0);              DERIVE-D0                       derive.fct(rd0);             DERIVE-B
derive.fct(d1);              DERIVE-D1                       derive.fct(rd1);             DERIVE-B
pbase2base->fct(b);          BASE-B                          pbase2base->fct(rb);         BASE-B
pbase2base->fct(d0);         BASE-D0                         pbase2base->fct(rd0);        BASE-B
pbase2base->fct(d1);         BASE-D1                         pbase2base->fct(rd1);        BASE-B
pbase2derive->fct(b);        DERIVE-B                        pbase2derive->fct(rb);       DERIVE-B
pbase2derive->fct(d0);       DERIVE-D0                       pbase2derive->fct(rd0);      DERIVE-B
pbase2derive->fct(d1);       DERIVE-D1                       pbase2derive->fct(rd1);      DERIVE-B
```

The above result confirms our conclusion : polymorphism happens between `BASE` and `DERIVE`, not among `B`, `D1` and `D1`.

FAQ for polymorphism

Q1. How do vtable and vptr work?

Suppose a base class having virtual functions `vir0`, `vir1` and `vir2`, a derived class overrides the first two functions :

```
class base
{
    base() : _vptr(base::_vtable) { ... }
    static fct_ptr_type _vtable[3] =
    {
        &base::vir0, &base::vir1, &base::vir2,
    };
    fct_ptr_type* _vptr; // memory cost for inheritance = extra 4 bytes per object
};

class derived : public base
{
    derived() : _vptr(derived::_vtable) { ... }
    static fct_ptr_type _vtable[3] =
    {
        &derived::vir0, &derived::vir1, &base::vir2
    };
};
```

- `vtable` is created as static member, one for base class and one for derived class (`base::_vtable ≠ derived::_vtable`)
- `vptr` is created as non-static member for each object, which points to `base::_vtable` if when `base` is constructed, and vice versa
- virtual function invocation `pbase->vir1(arg);`
  will then becomes `pbase->_vptr[1](arg);`

Q2. Virtual constructor, virtual assignment, virtual destructor

Never declare constructor nor copy constructor `virtual` because :

- `virtual` function requires dynamic binding according to dynamic type, which exists only when construction is done
- `virtual` function requires dynamic binding through `vptr`, however `vptr` is initialized in construction, chicken or egg

⇒ solve by using `virtual` create function (i.e. factory)

We can declare copy assignment `virtual`, but beware, as it is error prone

- `base::operator=(const base&)` is overriden by `derived::operator=(const base&)`
- `base::operator=(const base&)` is NOT overriden by `derived::operator=(const derived&)`
- given `base_ptr0 = base_ptr1`, where both pointers may point to `base` or `derived`, dynamic binding is determined by `base_ptr0`

```
class B
{
    // Please note that default auto-generated assignment operator is non-virtual. Hence we need to declare explicitly.
    virtual B& operator=(const B& rhs) { cout << "B::B"; x = rhs.x;            return *this; }
    T x;
};
class D : public B
{
    virtual B& operator=(const B& rhs) { cout << "D::B"; x = rhs.x;            return *this; }
            D& operator=(const D& rhs) { cout << "D::D"; x = rhs.x; y = rhs.y; return *this; } // not for dynamic-bind
    T y;
};

B b,b_rhs;
D d,d_rhs;
b = b_rhs; // static binding, B::B
b = d_rhs; // static binding, B::B
d = b_rhs; // static binding, D::B
d = d_rhs; // static binding, D::D

B &rbb = b;
B &rbd = d;
rbb = b_rhs; // dynamic binding, B::B
rbb = d_rhs; // dynamic binding, B::B
rbd = b_rhs; // dynamic binding, D::B
rbd = d_rhs; // dynamic binding, D::B (D::D is not picked, as it does not appear in vtable, possible object slicing)
```

Execution of line `rbd = d_rhs` will result in a mixture of `d` and `d_rhs` inside object `d`. If there are multiple derived classes, like different messages (`D0,D1,D2...`) in a protocol (`B`), then we can add a ID member in `B` and perform downcast in `B& D::operator=(const B& rhs)`.

```
if (rhs.id == typeD0)
{
    const auto& rhs_ref = dynamic_cast<const D0&>(rhs); x = rhs_ref.x; y = rhs_ref.y;
}
```

Always declare destructor `virtual` because :

- different derived class may different resources that have to be deallocated properly using its own destructor

Q3. Virtual function invoked in constructor and member function

If `virtual` function is invoked inside constructor, dynamic binding is forbidden, because `derived::vir_fct` may access uninitialized member prior to completion of `derived` construction. Thats why C++ forbits it.

```
struct base
{
    base()                          {   vir_fct();              }
    void fct()                      {   vir_fct();              }
    virtual void vir_fct()          {   std::cout << "\nbase fct";      }
    std::string str{ "uninit str" };
};

struct derived : public base
{
    derived()                       {   str = "initialized str";                    }
    virtual void vir_fct()          {   std::cout << "\nderived fct accessing " << str;     }
};

void main() { derived d; }          // output = base fct
```

If `virtual` function is invoked inside member function, this is a useful pattern called non-virtual interface.

```
class algo_base
{
public:
    void run()                      // This is non-virtual interface.
    {
        step1();
        step2();
        step3();
    }

private:                            // We make these implementations private on purpose.
    virtual void step1();           // It is perfectly legal to override private virtual function in derived class.
    virtual void step2();
    virtual void step3();
};
```

Q4. Virtual private, virtual static and virtual overloads

Should we have virtual private? Of course, this is exactly non-virtual interface, where base class defines non-virtual public interface, whereas derived class defines virtual private implementation. This is a decoupling between interface and implementation.

Should we have virtual static function? No definitely :

- static allows invocation without object, while
- virtual allows invocation (resolving) depending on object's dynamic type, i.e. needs to invoke with object.

Virtual overloads is error-prone as it mixes overloads with overrides together, so when derived class overrides one of the overloads, it may hide the other overloads, like the following, please try to avoid it.

```
struct base
{
    virtual  void f(int)            { std::cout << "\nB1"; }
    virtual  void f(int,int)        { std::cout << "\nB2"; }
    virtual  void f(int,int,int)    { std::cout << "\nB3"; }
};
struct derived : public base
{
    void f(int x)                   { std::cout << "\nD1"; }
};

derived d;
d.f(1);
d.f(1,1);                           // no matching call to 'derive::f(int,int)'        This is hiding.
d.f(1,1,1);                         // no matching call to 'derive::f(int,int,int)'    This is hiding.
```

virtual constructor (copy constructor)	*no*	*chicken or egg*
virtual copy assignment	*yes*	*but beware you know what you are doing*
virtual destructor	*yes*	*for specfic deallocation*
virtual function invoked in constructor	*no*	*access uninitialized variable*
virtual function invoked in member function	*yes*	*this is NVI idiom*
virtual private	*yes*	*this is NVI idiom*
virtual static	*no*	*dynamic binding needs dynamic type of object*
virtual overload	*no*	*hiding*

Q5. Private inheritance, protected inheritance and disable inheritance

Private inheritance is used to :

- achieve implement-as-relationship (not is-a relationship)
- forbid polymorphism (cannot set `base* p = new derived` as inheritance is unknown to others)
- it is useful for defining attributes / policy for class, such as `non_constructable` or `non_copyable`
- it is intuitive, it cannot be achieved by composition

Protected inheritance is rarely used, unless :

- we want inheritance relationship exposed to derived classes only

Disable inheritance

- How can we make a class non-inheritable? Declare the class with `final` keyword.
- How can we make a class method non-overridable? Declare the class method with `final` keyword.

Q6. Virtual inheritance and Diamond problem

6.1 Diamond problem happens in multi-inheritance, in which a class is derived from multiple basis. Consider the following :

```
struct base                                          { virtual void fct() { cout << "\nbase";      }};
struct derived0 : public base                        { virtual void fct() { cout << "\nderived0";  }};
struct derived1 : public base                        {};
struct derived2 : public base                        {};
struct concrete : public derived0, public derived1, public derived2    {};

base* p0 = new derived0; p0->f();
base* p1 = new derived1; p1->f();
base* p2 = new derived2; p2->f();
base* p3 = new concrete; p3->f(); // compile error : ambiguous conversion from concrete* to base*
```

There is compile error as `concrete` inherits a copy `base` of through `derived1` and a copy of `base` through `derived2`, this is a duplication, which is known as the diamond problem. It can be solved by adding `virtual` keyword to all `derived` (but not for `concrete`).

```
struct base                                          { virtual void fct() { cout << "\nbase";      }};
struct derived0 : public virtual base                { virtual void fct() { cout << "\nderived0";  }};
struct derived1 : public virtual base                {};
struct derived2 : public virtual base                {};
struct concrete : public derived0, public derived1, public derived2    {};

base* p0 = new derived0; p0->f(); // derived0::fct() is invoked
base* p1 = new derived1; p1->f(); //     base::fct() is invoked
base* p2 = new derived2; p2->f(); //     base::fct() is invoked
base* p3 = new concrete; p3->f(); // derived0::fct() is invoked
```

Why is it so? Whenever come across `virtual` inheritance :

- most-derived class (i.e. `concrete`) invokes `base` class constructor directly
- most-derived class does not initialized `base` class in `concrete` member initializer list, `base` default constructor is called
- with `virtual` inheritance, `base` class constructor is not called again in `derived0::derived0()` when it has been called
- `derived0::fct` is pointed in vtable of `concrete`
- compile error "ambiguous inheritance" happens if we also overide `derived1::fct` or `derived2::fct`

We need to declare all `derived` inheritances `virtual`, hence the following example doesn't work, it is here to demonstrate invocation sequence of constructors 6.3 :

```
struct base                             {    base()                { cout << "\nbaseA";    }
                                             base(int)             { cout << "\nbaseB";    } };
struct derived0 :  public virtual base  {    derived0():base(123)  { cout << "\nderived0"; } };
struct derived1 :  public virtual base  {    derived1():base(123)  { cout << "\nderived1"; } };
struct derived2 :  public base          {    derived2():base(123)  { cout << "\nderived2"; } };
struct concrete :  public derived0,
                   public derived1,
                   public derived2      {    concrete()            { cout << "\nconcrete"; } };

derived0 d0;        invocation sequence :  base(int) > derived0()
derived0 d1;        invocation sequence :  base(int) > derived1()
concrete x;         invocation sequence :  base() > derived0() > derived1() > base(int) > derived2 > concrete
```

Practical applications of polymorphism in YLib

We are going to go through 3 closely-related applications of polymorphism in YLibrary :

- C style polymorphism of POD
  C style protocol composing and parsing with POD
- message handling using self-made `vtable`
- event handling using visitor pattern

In this section, we will apply the following C++ techniques :

- `enum class` and `std::underlying_type_t`
- `reinterpret_cast` of POD for polymorphism
- `std::memcpy` of POD for protocol composing and parsing
- invocation of member pointers
- passing array by reference
- `assert` in debug mode
- throw `std::runtime_error` and string literals

Application 1 - C style polymorphism / Composing and parsing

When there is no inheritance in C language, how can we implement polymorphism in C? The answer is :

- for `POD` structure, we can `reinterpret_cast` a pointer to the `POD` into a pointer to ***first member*** of the `POD` (interchangably)
- for `POD` structure, we use a `enum class` as the first member to indicate the derived class type (it looks like protocol message)

```
enum class TYPE : std::uint8_t
{
    DERIVED0, DERIVED1, DERIVED2, NUM_OF_DERIVED
};

struct base
{
    TYPE type;
    inline void vir_fct() const noexcept; // *** Approach 2 *** //
};

struct derived0
{
    base header; std::uint32_t x; std::uint32_t y; std::uint32_t z;
    inline void vir_fct() const noexcept
    {
        std::cout << "\nderived0 = " << x << y << z
                  << " size = " << sizeof(derived0)
                  << " size = " << sizeof(header) + sizeof(x) +sizeof(y) + sizeof(z); // There are paddings.
    }
};

struct derived1
{
    base header; char s0[8]; char s1[8];
    inline void vir_fct() const noexcept
    {
        std::cout << "\nderived1 = " << std::string(s0,8) << std::string(s1,8)
                  << " size = " << sizeof(derived1)
                  << " size = " << sizeof(header) + sizeof(s0) +sizeof(s1);
    }
};

struct derived2
{
    base header; char s[64]; std::uint32_t len;
    inline void vir_fct() const noexcept
    {
        std::cout << "\nderived2 = " << std::string(s,len)
                  << " size = " << sizeof(derived2)
                  << " size = " << sizeof(header) + sizeof(s) +sizeof(len);
    }
};
```

There are two approaches to resolve those emulated-virtual-functions :

- resolution in another global function `invoke()`
- resolution in base class `base::vir_fct()`

As shown in the next page, the implementation inside `invoke()` and `base::vir_fct()` are exactly the same.

```
// *** Approach 1 *** //
void invoke(const base* msg)
{
    if      (msg->type == TYPE::DERIVED0) {  const derived0* derived = reinterpret_cast<const derived0*>(msg);
                                                              derived ->vir_fct();                               }
    else if (msg->type == TYPE::DERIVED1) {  const derived1* derived = reinterpret_cast<const derived1*>(msg);
                                                              derived ->vir_fct();                               }
    else                                  {  const derived2* derived = reinterpret_cast<const derived2*>(msg);
                                                              derived ->vir_fct();                               }
}

// *** Approach 2 *** //
void base::vir_fct() const noexcept
{
    if      (type == TYPE::DERIVED0)      {  const derived0* derived = reinterpret_cast<const derived0*>(this);
                                                              derived ->vir_fct();                               }
    else if (type == TYPE::DERIVED1)      {  const derived1* derived = reinterpret_cast<const derived1*>(this);
                                                              derived ->vir_fct();                               }
    else                                  {  const derived2* derived = reinterpret_cast<const derived2*>(this);
                                                              derived ->vir_fct();                               }
}
```

Here is the testing program. We try to invoke the virtual functions through objects or pointers etc.

```
test::derived0 d0{test::TYPE::DERIVED0, 11, 22, 33};
test::derived1 d1{test::TYPE::DERIVED1, "abcdefg", "ABCDEFG"};
test::derived2 d2{test::TYPE::DERIVED2, "This is a pen. This is a man.", 18};

// Invocation from object
d0.vir_fct();
d1.vir_fct();
d2.vir_fct();

// C style polymorphism, invoked by outside function (Approach 1)
std::vector<test::base*> vec;
vec.push_back(reinterpret_cast<test::base*>(&d0));
vec.push_back(reinterpret_cast<test::base*>(&d1));
vec.push_back(reinterpret_cast<test::base*>(&d2));
for(const auto& x:vec) test::invoke(x);

// C style polymorphism, invoked by itself (Approach 2)
for(const auto& x:vec) x->vir_fct();
```

Application 1 (continue) Composing and parsing

Apart from C-style polymorphism, PODs are good for composing and parsing datafeed protocol, like the following :

```
test::derived0 source      { test::TYPE::DERIVED0, 11, 22, 33 };
test::derived0 destination { test::TYPE::DERIVED0, 77, 88, 99 };
std::array<char, sizeof(test::derived0)> datafeed;

source.vir_fct(); destination.vir_fct(); // 11 22 33 77 88 99
std::memcpy(&buffer.front(), &source, sizeof(test::derived0));       // from source   to datafeed    (simulate composing)
std::memcpy(&destination, &buffer.front(), sizeof(test::derived0));  // from datafeed to destination (simulate parsing)
source.vir_fct(); destination.vir_fct(); // 11 22 33 11 22 33
```

With these two properties, PODs are very useful for datafeed handler for processing protocol messages :

- C style polymorphism which forwards different messages to different handlers
- fast composing and parsing of messages without copying

Application 2 - Message handling

From the above section, we know that PODs are good for implementing datafeed message handler, now lets implement it.

```
// Same enum class, base and derived classes as previous section,
// simply remove vir_fct, which is irrelevant to this use case.
enum class TYPE : std::uint8_t { DERIVED0, DERIVED1, DERIVED2, NUM_OF_DERIVED };

struct base     {   TYPE type;    };
struct derived0 {   base header;  std::uint32_t x; std::uint32_t y; std::uint32_t z;    };
struct derived1 {   base header;  char s0[8];      char s1[8];                          };
struct derived2 {   base header;  char s[64];      std::uint32_t len;                   };
```

Here is the message handler, we can also define derived handler to override some of the handling functions.

```
class handler
{
public:
    using idx_type = std::underlying_type_t<TYPE>;
    using fct_type = void (handler::*)(const base*) const noexcept;

    handler()
    {
        vtable[static_cast<idx_type>(TYPE::DERIVED0)] = &handler::fct0;
        vtable[static_cast<idx_type>(TYPE::DERIVED1)] = &handler::fct1;
        vtable[static_cast<idx_type>(TYPE::DERIVED2)] = &handler::fct2;
    }

    void handle(const base* msg) const noexcept
    {
        auto ptr = vtable[static_cast<idx_type>(msg->type)];
        (this->*ptr)(msg); // Bracket around (this->*ptr) is a must.
    }

private:
    // Handler's standard implementation. Each has 3 steps :
    // 1. assert in debug
    // 2. reinterpret_cast
    // 3. invoke member pointer
    void fct0(const base* msg) const noexcept
    {
        assert(msg->type == TYPE::DERIVED0);
        const derived0* derived = reinterpret_cast<const derived0*>(msg);
        std::cout << "original handler for derived0 : " << ...
    }

    void fct1(const base* msg) const noexcept
    {
        assert(msg->type == TYPE::DERIVED1);
        const derived1* derived = reinterpret_cast<const derived1*>(msg);
        std::cout << "original handler for derived1 :" << ...
    }

    void fct2(const base* msg) const noexcept
    {
        assert(msg->type == TYPE::DERIVED2);
        const derived2* derived = reinterpret_cast<const derived2*>(msg);
        std::cout << "original handler for derived2 :" << ...
    }

protected:
    std::array<fct_type, static_cast<idx_type>(TYPE::NUM_OF_DERIVED)> vtable;
};

// Test program
std::vector<message::base*> vec;
vec.push_back(reinterpret_cast<message::base*>(&d0));
vec.push_back(reinterpret_cast<message::base*>(&d1));
vec.push_back(reinterpret_cast<message::base*>(&d2));

message::handler h;
for(const auto& x:vec) h.handle(x);
```

Here is a derived handler. For example, base handler is for adminstrative messages, derived handler is for business messages.

```
class derived_handler : public handler
{
public:
    using idx_type = handler::idx_type;
    using fct_type = handler::fct_type;

    derived_handler() : handler{}
    {
        // static_cast from derived::mem_ptr to base::mem_ptr is necessary, otherwise compile error
        handler::vtable[static_cast<idx_type>(TYPE::DERIVED0)] =
                        static_cast<fct_type>(&derived_handler::over_fct0);
        handler::vtable[static_cast<idx_type>(TYPE::DERIVED1)] =
                        static_cast<fct_type>(&derived_handler::over_fct1);
    }

private:
    void over_fct0(const base* msg) const noexcept
    {
        assert(msg->type == TYPE::DERIVED0);
        const derived0* derived = reinterpret_cast<const derived0*>(msg);
        std::cout << "override handler for derived0 :" << ...
    }

    void over_fct1(const base* msg) const noexcept ...
```

Application 3 - Event handling

As events in YLibrary are not PODs, instead they form a real inheritance tree. Visitor pattern is used in event handling, which :

- separate event handling from event structure
- event handling is done by `handler::on_xxx_event()`
- event notifies handler by `event::accept(handler&)`
- terminology correspondence between event system and visitor pattern

pattern	*example1*	*example2*
`element`	`shape`	`event`
`element::accept(visitor&)`	`shape::accept(visitor&)`	`event::accept(handler&)`
`visitor`	`area_visitor, perimeter_visitor`	`pricer, hitter, quoter`
`visitor::visit(element&)`	`area_visitor::visit(shape&)`	`pricer::handle(event&)`

Here is the inheritance hierarchy :

```
struct base
{
    virtual void accept(handler& h) const = 0;
};

// The following are not PODs, customized constructor must be provided.
struct derived0 final : public base
{
    std::uint32_t x; std::uint32_t y; std::uint32_t z;
    void accept(handler& h) const override {  h.on_derived0_event(*this); }
};

struct derived1 final : public base
{
    derived1(const char (& s0_)[8], const char (& s1_)[8]) // Passing array as reference
    {
        std::memcpy(s0, s0_, 8);
        std::memcpy(s1, s1_, 8);
    }
    char s0[8]; char s1[8];
    void accept(handler& h) const override {  h.on_derived1_event(*this); }
};

struct derived2 final : public base
{
    derived2(const char* ptr, std::uint32_t str_len)
    {
        using namespace std::string_literals;
        if (str_len > 64) throw std::runtime_error("too long : "s + std::to_string(str_len));

        std::memcpy(s, ptr, str_len);
        len = str_len;
    }
    char s[64]; std::uint32_t len;
    void accept(handler& h) const override {  h.on_derived2_event(*this); }
};


class handler
{
public:
    // Default implementation : do nothing ...
    virtual void on_derived0_event(const derived0& event) noexcept { }
    virtual void on_derived1_event(const derived1& event) noexcept { }
    virtual void on_derived2_event(const derived2& event) noexcept { }
};

class pricer : public handler
{
    void on_derived0_event(const derived0& event) noexcept override { /* please implement */ }
    void on_derived1_event(const derived1& event) noexcept override { /* please implement */ }
};

class hitter : public handler
{
    void on_derived1_event(const derived1& event) noexcept override { /* please implement */ }
    void on_derived2_event(const derived2& event) noexcept override { /* please implement */ }
};

std::vector<event::base*> vec;
vec.push_back(reinterpret_cast<event::base*>(&d0));
vec.push_back(reinterpret_cast<event::base*>(&d1));
vec.push_back(reinterpret_cast<event::base*>(&d2));

event::pricer pricer;
for(const auto& x:vec) x->accept(pricer);
event::hitter hitter;
for(const auto& x:vec) x->accept(hitter);
```

G. Generic programming

G1. Template specialization

Function template and function template specialization are declares as the following, `T` and `U` are type template parameters, while `N` is non-type template parameter. Non-type template parameter can only be integer (no floating point, nor string). There exist default template parameter (in template parameter list), template partial specialization and template complete specialization.

Function template

template parameter list

```
template<typename T, typename U> void function(const T& t,     const U& u)      { ... }
template<typename T>             void function(const T& t,     const MOMENT& u) { ... } // this is NOT partial specialization
template<>                       void function(const GAUSS& t, const MOMENT& u) { ... } // this is NOT complete specialization
                                                                                         // the above are overloading
```

Class template

Main difference between template function and template class, no need to specify `<T,U>` for template function specialization.

```
template<typename T, typename U> class algo { ... };
template<typename T, typename U> algo<T,U>::f0() { ... }
template<typename T, typename U> algo<T,U>::f1() { ... }
template<typename T, typename U> algo<T,U>::f2() { ... }

template<typename T> class algo<T,MOMENT> { ... }; // partial specialization
template<typename T> algo<T,MOMENT>::f0() { ... }
template<typename T> algo<T,MOMENT>::f1() { ... }
template<typename T> algo<T,MOMENT>::f2() { ... }

template<> class algo<GAUSS,MOMENT> { ... }; // complete specialization
template<> algo<GAUSS,MOMENT>::f0() { ... }
template<> algo<GAUSS,MOMENT>::f1() { ... }
template<> algo<GAUSS,MOMENT>::f2() { ... }
```

Template function is a generalized function overload, it can be invoked without specifying template parameters, as type deduction is triggered. However, template class must be instantiated with template parameters specified, such as `std::vector<bool>`, as there is no type deduction for template class.

```
function(gauss_object, moment_object);    // instantiation of function template
class algo<GAUSS,MOMENT,3> algo_object;   // instantiation of class template
```

Member function template

```
// 1. Declaration
template<typename T> struct algo
{
    template<typename U> void fct() { std::cout << "template member"; }
};

// 2. Definition
template<typename T>
template<typename U>
void algo<T>::fct<U>() { std::cout << "template member"; }

// 3. Usage
template<typename T> void invoke_algo(algo<T>& x)
{
//  x.fct<deal_type>();      // compile error, compiler does not know it is a template member
    x.template fct<input>(); // keyword "template" is needed to invoke template member
}

algo<model> x;
invoke_algo(x);
```

Passing member pointer to template function in two approaches :

- approach 1 - pass as non-template parameter
- approach 2 - pass as template parameter

```
// given data struct
struct simple_pod           { std::uint32_t x{0}; std::string s{""}; };
void init(std::uint32_t& x) { x = 123;   }
void init(std::string& s)   { s = "abc"; }

template<typename T> void pass_as_non_template_para(simple_pod& pod, T simple_pod::* mem_ptr)
{
    init(pod.*mem_ptr);
};

template<typename M> void pass_as_template_para(simple_pod& pod, M mem_ptr)
{
    init(pod.*mem_ptr);
};

// syntax in caller-side is the same for both approaches
simple_pod pod0;
pass_as_non_template_para(pod0, &simple_pod::x);
pass_as_non_template_para(pod0, &simple_pod::s);
simple_pod pod1;
pass_as_template_para(pod1, &simple_pod::x);
pass_as_template_para(pod1, &simple_pod::s);
```

Abbreviated function template

- it involves `auto` to declare argument
- it involves `decltype` to forward universal reference (see below)
- it involves `concepts` to constrain argument

```
void fct(const auto& arg0, auto&& arg1, my_concept auto& arg2)
{
     impl(std::forward<decltype(arg1)>(arg1));
}

// which is equivalent to ...
template<typename T, typename U, my_concept V>
void fct(const T& arg0, U&& arg1, V& arg2)
{
     impl(std::forward<U>(arg1));
}
```

Variable template and alias template are similar. Both are shortcuts to traits.

- variable template can be specialized, variable template offers shortcut to value-traits by forwarding
- alias template cannot be specialized, but alias template offers shortcut to type-traits by forwarding

Variable template

```
// define value-traits (with specialization) by variable template
template<typename T> inline constexpr int value_traits           = 0;
template<>           inline constexpr int value_traits<my_classA> = 1;
template<>           inline constexpr int value_traits<my_classB> = 2;
template<>           inline constexpr int value_traits<my_classC> = 3;

// forward value-traits by variable template
template<typename T> inline constexpr int value_traits_v = any_value_traits<T>::value;
```

Alias template (cannot be specialized)

```
// define type-traits (WITHOUT specialization) by alias template
template<typename T> using type_traits = my_classA; // i.e. all input-types map to the same output-type

// forward type-traits by alias template
template<typename T> using type_traits_t = typename any_type_traits<T>::type;
```

G2. Template traits

Template traits is a mapping of types (i.e. `f:type->type` or `f:type->value`) :

- keyword `enum` defines constant integer
- keyword `typedef` defines shortcut to a type (we can also use `using` instead)

```
template<typename C> struct container_traits
{
    enum { size = 1024; }
    static const bool value = true;

    typedef C::value_type    value_type;
    typedef C::value_type* pointer_type;
    using   value_type2 = C::value_type;   // equivalent to above
    using pointer_type2 = C::value_type*;  // equivalent to above
};

// keyword typename is used to tell compiler that it is a type instead of static member
typename container_traits<my_vector>::value_type x;
typename container_traits<my_vector>::pointer_type ptr;
```

There are different ways to create value-traits and type-traits using template specialization.

```
// define value-traits (with specialization) by class template
template<typename T> struct type_traits { static const bool value = false; };
template<> struct type_traits<A>        { static const bool value = true;  };
template<> struct type_traits<B>        { static const bool value = true;  };
template<> struct type_traits<C>        { static const bool value = true;  };

// define type-traits (with specialization) by class template
template<typename T0,typename T1,typename T2,int N> struct v2t_traits { typedef v2t_traits<T0,T1,T2,N-1>::type type; };
template<typename T0,typename T1,typename T2>       struct v2t_traits<T0,T1,T2,0>  { typedef T0 type; };
template<typename T0,typename T1,typename T2>       struct v2t_traits<T0,T1,T2,10> { typedef T1 type; };
template<typename T0,typename T1,typename T2>       struct v2t_traits<T0,T1,T2,20> { typedef T2 type; };

// define value-traits (with specialization) by variable template
template<typename T> inline constexpr int value_traits           = 0;                    // copied from 1.5 above
template<>           inline constexpr int value_traits<my_classA> = 1;
template<>           inline constexpr int value_traits<my_classB> = 2;
template<>           inline constexpr int value_traits<my_classC> = 3;

// forward value-traits by variable template
template<typename T> inline constexpr int value_traits_v = any_value_traits<T>::value;   // copied from 1.5 above

// define type-traits (WITHOUT specialization) by alias template
template<typename T> using type_traits = my_classA;                                      // copied from 1.6 above

// define type-traits by alias template
template<typename T> using type_traits_t = typename any_type_traits<T>::type;            // copied from 1.6 above
```

G3. Template template (for template class)

Template-template is a template class or function that takes template class (like container) as template parameter. The default value `ASSOCIATIVE_CONTAINER_TYPE = std::map` is fine with `c++17` and beyond only, for old version, `std::map` is considered as template with four template parameters, hence does not match the requirement and generate compilation error.

```
template<typename T,
        typename U,
        template<typename,typename> class ASSOCIATIVE_CONTAINER_TYPE = std::map, // default value is ok
        template<typename> class TRAITS_TYPE = bool_traits>                     // default value is ok
class algo
{
    ASSOCIATIVE_CONTAINER_TYPE<T,U> my_map;
    TRAITS_TYPE<T>::is_primitive my_bool = ...;
    ...
};

algo<boost::posix_time::ptime, std::string> algo1;
algo<boost::posix_time::ptime, cdoi::string, cdoi::map, cdoi::traits> algo2;
```

G4. Variadic template (for template function)

Variadic template is template having variable number of arguments, it is usually defined in a recursive manner. Ellipsis operator … denotes a pack, it happens in three ways :

- a template parameter set with variable size in template parameter list as `typename...`
- an argument set in function prototype as `Ts...` (very often, it comes with universal reference) and
- an argument set inside function definition as `args...` (very often, it is forwarded with `std::forward<Ts>`)

```
// one-arg boundary case
template<typename T>
void log(const T& arg)
{
    std::cout << arg;
}

// general case
template<typename T, typename... Ts>
void log(const T& arg, Ts&&... args)
{
    log(arg);
    log(std::forward<Ts>(args)...);
}
```

OR

```
// empty boundary case
// non-template
void log()
{
    // do nothing
}

// general case
template<typename T, typename... Ts>
void log(const T& arg, Ts&&... args)
{
    std::cout << arg;
    log(std::forward<Ts>(args)...);
}
```

```
// invoked as
log(123, algo1, gauss, moment1, moment2);
log(456, algo2, poisson, waiting_time);
log(789, algo3, node, edge, graph);
```

G5. Basic metaprogramming

`std::true_type` and `std::false_type` are in fact instantiations of a template class `std::integral_constant`.

```
template<typename T, T N> struct integral_constant
{
    typedef T value_type;
    static const T value = N;
};

typedef integral_constant<bool,  true>  true_type;
typedef integral_constant<bool, false> false_type;
```

With `std::integral_constant`, we can start template metaprogramming. Always remember to provide boundary case for recursion.

```
template<int N> struct factorial    : integral_constant<int, N*factorial<N-1>::value> {};
template<>      struct factorial<1> : integral_constant<int, 1> {};
template<int N> struct fibonacci    : integral_constant<int, fibonacci<N-1>::value> + fibonacci<N-2>::value> {};
template<>      struct fibonacci<1> : integral_constant<int, 1> {};
template<>      struct fibonacci<2> : integral_constant<int, 1> {};
```

G6. Template class as base class

If a class is derived from a template base class, there is compile error in `gcc` (but not in `MSVC`) if we access protected data members of base class. Please read stackoverflow thread 50321788. There are 3 solutions :

- add `this` pointer for each access in derived class
- add `base<T>::mem` resolution for each access in derived class
- add short cut (alias) `using base<T>::mem` at the beginning of derived class

```
// This problem happens in (1) template base class AND (2) gcc compiler.
template<typename T> class base
{
protected: int x; int y;
};

template<typename T> class derived : public base<T>
{
public:
    using base<T>::x; // solution : alias
    using base<T>::y; // solution : alias
    void fct()
    {
        x = 1; // compile error : x is not declared in this scope (if there is no alias)
        y = 2; // compile error : y is not declared in this scope (if there is no alias)
    }
};
```

G7. Type erasure pattern

The objective is to design non-template class `type_erase` with a template constructor so that it can be constructed from any unrelated class, wraps and housekeeps the latter, for invocation of common interfaces. Like `std::any` and `std::function`, yet they are templates.

```
struct A
{
    A(int x, int y);
    void f(const std::string& str) const;
};

void invoke(const type_erase& any)
{
    any.f("hello world");
}
```

```
struct B
{
    B(int x, int y, int z);
    void f(const std::string& str) const;
};

std::vector<type_erase> vec;
vec.push_back(A{1,2});
vec.push_back(B{3,4,5});
for(const auto& x:vec) invoke(x);
```

Can we solve it with template `invoke` (without using `type_erase`)? No, as we cannot instantiate a vector storing different classes :

```
template<typename T> void invoke(const T& any)
{
    any.f("hello world");
}
```

```
std::vector<???> vec;
vec.push_back(A{1,2});
vec.push_back(B{3,4,5});
for(const auto& x:vec) invoke(x);
```

Can we solve it with polymorphism (without using `type_erase`)? No, as `A` and `B` are not derived from same base class :

```
void invoke(const base* any_ptr)
{
    any_ptr->f("hello world");
}
```

```
std::vector<base*> vec;
vec.push_back(new A{1,2});
vec.push_back(new B{3,4,5});
for(const auto& x:vec) invoke(x);
```

Given the requirement above, how can we implement `type_erase`?

- `type_erase` is non template, it can wrap different types, hence `type_erase` must have a template constructor
- `type_erase` internally should keep one instance of the wrapped object (as a pointer) and allows polymorphism
- `type_erase` can resolve into un-related classes in runtime, hence we need to *"create an inheritance"* internally

Step 1 : Create a base class

```
class object_base
{
    virtual ~object_base() {}
    virtual void f(const std::string& str) const = 0;
};
```

Step 2 : Create a set of derived class

```
template<typename T> class object_wrapper : public object_base
{
    object_wrapper(const T& object) : _object(object) {}
    void f(const std::string& str) const override { return _object.f(str); }

    private: T _object;
};
```

Step 3 : Put everything inside `type_erase` and implement a template constructor

```
struct type_erase
{
    class object_base;                                              // we can move the implementation to here
    template<typename T> class object_wrapper : public object_base;        // we can move the implementation to here

    template<typename T> type_erase(T&& object) : base_ptr(new object_wrapper<T>(std::forward<T>(object))) {}
    inline void f(const std::string& str) const { return base_ptr->f(str); }

private:
    std::unique_ptr<object_base> base_ptr;
};
```

If we replace member `void f(const std::string&)` by `void operator()(const std::string&)`, then we can do something like :

```
std::vector<base*> vec;
vec.push_back(new A{1,2});
vec.push_back(new B{3,4,5});
vec.push_back(function);                       // where void function(const std::string&) is global function
vec.push_back(+[](const std::string&){ ... });      // where + sign converts lambda into function pointer
```

Possible applications of type erasure include : deleter in `std::shared_ptr`, `std::function` and `std::any`.

H. Template containers and algorithms

H1. Containers

	sequential	associative	container adaptor	other

array based	`array<T,N>`
	`vector`
	`deque`
	`string`
		`unordered_set`
		`unordered_map`
		`unordered_multiset`
		`unordered_multimap`
			`stack`
			`queue`
			`priority_queue`
				`mpmcq (circular buffer)`
				`disruptor`

node based	`(doubly) list`
	`(singly) forward_list`
		`set`
		`map`
		`multiset`
		`multimap`
				`Btree`
				`graph`
				`direct_acyclic_graph`
				`multi_partite_graph (network)`

Common iterators for container :

- input iterator
- output iterator
- forward iterator
- backward iterator
- random access iterator
- back inserter
- istream iterator
- ostream iterator

```
std::istream_iterator<std::string> is_iter(std::cin); // block and wait for input here
std::ostream_iterator<std::string> os_iter(std::cout, " <- echoed output");

auto s0 = *is_iter;    *os_iter = s0;        ++is_iter; // block and wait for input here
auto s1 = *is_iter;    *os_iter = s1;        ++is_iter; // block and wait for input here
auto s2 = *is_iter;    *os_iter = s2;
```

Common member functions for container :

- modify `insert, emplace, erase, clear, push_back, pop_back` — `array<T,N>` does not have modifiers
- access `[], at, front, back` — `array<T,N>` does have accessors
- iterator `begin, end, rbegin, rend`
- capacity `empty, size, capacity, reserve`

Performance of common functions (note `insert(iter,x)` function means adding item `x` right before `iter`) :

- `insert` function for most containers supports user-provided iterator as a ***hint iterator*** for faster insertion

	insert/erase	search	iterator
vector	O(1) back O(n) middle O(n) reallocate	O(n) unsorted O(logn) sorted	not persistent (may resize)
unordered_set	O(1) O(n) collision O(n) rehashing	O(1) O(n) collision	not persistent (may resize and rehash)
list	O(1)	O(n)	persistent
set	O(logn)	O(logn)	persistent

For `std::map<K<V>` we have the following alias

```
std::map<K<V>::key_type      = K
std::map<K<V>::mapped_type   = V
std::map<K<V>::value_type    = std::pair<K,V>
```

Requirements on T for STL containers 1.5

Firstly should `T` be default-constructible and copy/move constructible? Lets consider *possible* implementations of array and vector.

- `std::array<T,N>` has no constructor, but it is an aggregate and thus can be aggregate-initialized
- `std::array<T,N>` is implemented as raw array using stack memory
- `std::vector<T>` is implemented as raw pointer using heap memory

There are different template requirements on type `T` depending on how we construct/use the container.

- `std::array<T,N>` requires `T` to be default constructible if we default-initialize the array
- `std::array<T,N>` requires `T` to be copyable / movable if we aggregate-initialize the array
- `std::vector<T>` does ***NOT*** require `T` to be default constructible
  as `std::vector<T>` does ***NOT*** call `T::T()` to avoid double init, it calls emplace (inplace construction, no copy nor move) instead
- `std::vector<T>` requires `T` to be copyable / movable (if `T` is ***NOT***, we would declare `std::vector<std::unique_ptr<T>>`)

```
template<typename T, std::uint32_t N> class array final
{
    // no constructor, however it is aggregate, so it can be aggregate-initialized
    T impl[N];
};

template<typename T> class vector
{
    void push_back(const T& x)
    {
        if (size == capacity) { capacity *= 2; ptr = realloc(ptr, capacity); }
        new (&ptr[size]) T(x); // invoke T::T(const T&)
        ++size;
    }
    T* ptr;
};

std::array <my_class, 3> a0;              //     require my_class to be default-constructible
std::array <my_class, 3> a1 = { x,y,z }; //     require my_class to be copable or movable
std::vector<my_class>    v0;              // NOT require my_class to be default-constructible, as no construction of T happens here
std::vector<my_class>    v1 = { x,y,z }; //     require my_class to be copable or movable
```

Secondly can we have reference as element type? May element type `T` contain reference member (`T` is non-default-constructible)?

- `std::vector<T&>` results in compilation error on declaration ***because vector requires element to be assignable***
- `std::vector<T>` results in compilation error on allocation of *N* elements, which invokes `T::T()`
- `std::vector<T>` is fine, as long as it does not invoke deleted members of `T`

`T::T()` and two assignments are auto-deleted
`T::T(cons T&)` and `T::T(T&&)` are retained

```
struct T
{
    T(const int& ref) : r(ref){}
    const int& r;
};

std::vector<T&> v0;     // compile error on declaration of T& vector
std::vector<T>  v1(10); // compile error on declaration of T vector, with allocation of 10 T elements
std::vector<T>  v2;
v2.push_back(T{a});
v2.push_back(T{b});
v2.push_back(T{c});
for(const auto& x:v2) std::cout << x.r;
```

Thirdly can we have incomplete type for container and iterator? All containers need to allocate `T` either in stack or in heap, hence it needs to know `sizeof(T)` either during container construction or container insertion hence complete type of `T` is necessary. Although some compilers may compile on container construction with forward declaration only, there will be compilation errors on insertion of element afterwards.

```
struct my_class;                                    // forward declaration without class definition, so sizeof(class) is unknown
std::vector<my_class> vec;                          // compilation OK
vec.push_back(T{a,b,c});                            // compilation error, need to know complete type
std::vector<my_class>::iterator i = vec.begin();    // compilation OK
```

Yet, most STL iterators are implemented as raw pointers only, thus most likely, they do not require complete type. `std::unordered_map` is the only exception, we need to provide complete type of `K` and `V` to construct `std::unordered_map<K,V>::iterator`. In short :

- For STL containers, complete type for `T` is necessary
- For STL iterators, complete type for `T` is *NOT* necessary, except for `std::unordered_map`

Requirements on `T` for `STL` maps

Requirements on key of hashmap

We can use custom key for `std::unordered_map`, as long as we provide the following for the key :

- hash function of the key
- spaceship operator of the key
- so that when multiple keys hashed into the same bucket (i.e. collision), we can differentiate the keys
- both constraints are necessary, otherwise there will be compile error

For example, using `pod` as the key :

```
struct pod
{
    char i;
    char j;
    char k;

    auto operator<=>(const pod& rhs) = default; // i.e. memberwise spaceship operator
};

struct pod_hash
{
    std::size_t operator()(const pod& x)
    {
        return (i<<16) | (j<<8) | k;
    }
};

std::unordered_map<pod, std::string, pod_hash> map;
pod x{'a','b','c'}; map[x] = "abcde";
pod y{'b','c','d'}; map[y] = "bcdef";
pod z{'c','d','e'}; map[z] = "cdegh";
```

Mutability of key

We can change the value for `std::set` or `std::map`, but not the key (as it is constant) :

```
if (auto iter = my_map.find(key); iter != my_map.end())
{
    iter->first  = new_key;                        // compile error
    iter->second = new_value;                      // This is fine.
}
```

We have to do it through `find`, `erase` and `insert`, which involves either copy or move of value.

```
if (auto iter = my_map.find(key); iter != my_map.end()) // involve O(logN) search
{
    auto const value = std::move(iter->second);        // involve one copy or move of V
    my_map.erase(iter);
    my_map.insert({ new_key, std::move(value) });      // involve another copy or move of V
}
```

C++17 introduces function `extract`, which combines `find` and `erase` together. There is no copy nor move of value in the process.

```
if (auto node = my_map.extract(key); !node.empty())     // involve O(logN) search, map size is reduced by 1 after extract
{
    node.key() = new_key;
    my_map.insert(std::move(node));                    // move of node is easier than move of value
}
```

H2. Algorithms

The unary functors take reference to `std::iterator_traits<ITER>::value_type` as input argument, while returning bool.

```
// 2.1 Non modifying sequence operation
template<typename ITER, typename FCT> bool std::all_of(ITER i0, ITER i1, FCT fct)
{
     for(ITER i=i0; i!=i1; ++i) if (!fct(*i)) return false;
     return true;
}
std::cout << std::boolalpha << std:: all_of(begin_iter, end_iter, unary_functor);
std::cout << std::boolalpha << std:: any_of(begin_iter, end_iter, unary_functor);
std::cout << std::boolalpha << std::none_of(begin_iter, end_iter, unary_functor);

// 2.2 Non modifying sequence operation
template<typename ITER, typename FCT> void std::for_each(ITER i0, ITER i1, FCT fct)
{
     for(ITER i=i0; i!=i1; ++i) fct(*i);
}

// 2.3 Copying across different containers
template<typename ITER, typename OITER> void std::copy(ITER i0, ITER i1, OITER j)
{
     for(ITER i=i0; i!=i1; ++i, ++j) *j=*i;
}

// 2.4 Modifying sequence operation
template<typename ITER, typename OITER, typename FCT> void std::transform(ITER i0, ITER i1, OITER j, FCT fct)
{
     for(ITER i=i0; i!=i1; ++i, ++j) *j=fct(*i);
}

// 2.5 std::max and std::max_element are different
template<typename ITER> ITER std::max_element(ITER i0, ITER i1)
{
     ITER j = i0;
     for(++i0; i0!=i1; ++i0) { if(*j<*i0) j=i0; }
     return j;
}

// 2.6 Assume items are sorted, find the first element above given lower bound (using logN binary search)
template<typename ITER> ITER std::lower_bound(ITER i0, ITER i1, const typename std::iterator_traits<ITER>::value_type& x)
{
     while(dist > 0)
     {
          auto dist = std::distance(i0,i1);
          ITER mid = i0; std::advance(mid, dist/2);
          if (*mid < x) i0 = ++mid;
          else          i1 =   mid;
     }
     return i0;
}

// We can store (i0,dist) instead of (i0,i1) to avoid repeated calculation of distance.
template<typename ITER> ITER std::lower_bound(ITER i0, ITER i1, const typename std::iterator_traits<ITER>::value_type& x)
{
     auto dist = std::distance(i0,i1);
     while(dist > 0)
     {
          ITER mid = i0; std::advance(mid, dist/2);
          if (*mid < x) { i0 = ++mid; dist = dist-(dist/2+1); }
          else          {             dist = dist/2;          }
     }
     return i0;
}

// 2.6 Application of std::lower_bound is to build a sorted std::list by finding right insertion position on every iteration.
template<typename T> struct sorted_list final
{
    void insert(const T& x)
    {
        auto i = std::lower_bound(impl.begin(), impl.end(), x);
        impl.insert(i, x);
    }
    std::list<T> impl;
};
```

Algorithms for sorting

```
// 2.7 Elements returning TRUE on fct are sorted before elements returning FALSE on fct. Return iter to 1st false element.
template<typename ITER> ITER std::partition(ITER i0, ITER i1, FCT fct)
{
    --i1;
    while(i0!=i1)
    {
        if (fct(*i0)) ++i0;
        else { std::swap(*i0,*i1); --i1; }
    }
    // latest i0 has not been considered yet
    if (f(*i0)) return i1;
    else return i0;
}
```

Algorithms in numerics library

```
// 2.8 Accumulation
template<typename ITER> T std::accumulate(ITER i0, ITER i1, T init)
{
    for(ITER i=i0; i!=i1; ++i) init += *i;
    return init;
}

// 2.9 Greek "I-O-TA" denotes a function that generates a sequence. DON'T confuse with itoa.
template<typename ITER> void std::iota(ITER i0, ITER i1, T init)
{
    for(ITER i=i0; i!=i1; ++i) { *i = init; ++init; }
}
```

Algorithms in parallel

```
// 2.10 Auto-spawn threads, as many threads as the core has
std::vector<int> v; for(int n=0; n!=1000; ++n) v.push_back(rand());
std::for_each(std::execution::par, std::begin(v), std::end(v), [](auto& x) { cout << this_thread::get_id(); }); // print diff id
std::for_each(std::execution::seq, std::begin(v), std::end(v), [](auto& x) { cout << this_thread::get_id(); }); // print same id
```

Back-inserter

For all the above algo, back-inserter is needed as the output iterator when output container has a size smaller than the input.

```
std::vector<int> vec{1,2,3,4,5,6,7,8,9,10};
std::list<int> list; // empty
std::copy(vec.begin(), vec.end(), std::back_inserter(list));
```

What back-inserter does ...

```
list_back_inserter<T>& list_back_inserter<T>::operator++()
{
    if (pos == list.size()-1) list.resize(...);
    advance(pos);
}
```

H3. String and string algorithms

Both `std::string::c_str()` and `std::string::data()` return `char*` representing the content of `std::string`.

- `c_str()` terminates with null character
- `data()` does not terminate with null character

Find and substring

```
// for trimming
size_t pos0 = str.find_first_not_of(“.,:- \t\n”); // find_first_of
size_t pos1 = str.find_last_not_of (“.,:- \t\n”); // find_last_of
auto str0 = str.substr(pos0, pos1-pos0+1);
// for tag-searching
size_t pos = str0.find(“tag”, offset);
if (pos != std::string::npos) auto str1 = str0.substr(pos, size);
```

Conversion to lower / upper case

```
#include <ctype.h>
std::string str(“This Is A Test.”);
for(auto& x:str) x = tolower(x); std::cout << “\nlower ” << str;
for(auto& x:str) x = toupper(x); std::cout << “\nupper ” << str;
```

Conversion from integer to ASCII and reverse

```
// from yyyy_mm_dd to integers
int y = std::stoi(str.substr(0,4));
int m = std::stoi(str.substr(5,2));
int d = std::stoi(str.substr(8,2));
auto str = std::to_string(3.141592654);
```

String can be concaternated in two ways :

```
std::stringstream ss; ss << "This is " << x << ".";

using namespace std::string_literals; // Don't miss this.
std::string str;
str += "This is "s += std::to_string(x) += ".";
```

Naive string implementation with movability and SSO (Short String Optimization).

```
// SSO means using stack memory for short string.
class string
{
public:
    // all 83 member functions, woo ...

private:
    std::array<char, 16>    m_sso;  // for string shorter than 16 bytes
    std::unique_ptr<char[]> m_data; // for longer string, with unique_ptr, move-construction and move assignment can be default.
    size_type m_size;
    size_type m_capacity;
};
```

H4. Hashmap

Comparison between binary search tree (for traversal) and hashmap (for fast searching)

advantages of binary search tree	advantages of hashmap
memory efficient	cache friendly, low latency at the expense of memory
no rehashing, guaranteed search time O(logN)	fast search / insert / erase, O(1) but not guaranteed
no collision, guaranteed search time O(logN)	
offer ordered tranversal *(using inorder-DFS)*	
offer fast range-search or nearest-target-search	
offer simple concurrency on different subtrees	

Hash function offered by STL is a functor that maps `T` into `int` :

```
std::hash<int> h0;
std::hash<std::string> h1;
for(int n=0; n!=100; ++n) { auto b = rand_byte();  std::cout << "\nhash of " << b << h0(b); }
for(int n=0; n!=100; ++n) { auto s = rand_str();   std::cout << "\nhash of " << s << h1(s); }
```

We can thus create our own hash function for `std::unordered_set` and `std::unordered_map` :

```
struct simple_hash_byte // For single byte : XOR sequence of left-shift and right-shift
{
    int operator()(unsigned char x)
    {
        x = x ^ (x << 13);
        x = x ^ (x >> 17);
        x = x ^ (x << 5);
        return x;
    }
};

struct simple_hash_string // For multi bytes : adding hash of each byte
{
    int operator()(const std::string& s)
    {
        int x = 0;
        for(const auto& c:s) x += 31 * x + simple_hash_byte()(c);
        return x;
    }
};

std::unordered_map<int, std::string, simple_hash_byte> m0;
std::unordered_map<std::string, int, simple_hash_string> m1;
```

How to iterate through all buckets in `std::unordered_map`?

```
for(int n=0; n!=100; ++n) map[rand_str()] = rand_int();
for(int n=0; n!=map.bucket_count(); ++n)
{
    std::cout << "[Bucket " << n << "] ";
    for(auto iter = map.begin(n); iter != map.end(n); ++iter) cout << iter->first << ":" << iter->second;
}
```

How to iterate through all values associated to a key in `std::unordered_multimap`?

```
    std::unordered_multimap<std::string, int> map;
//  map["abc"] = 1;         // compile error, operator [] can't resolve between insert-new-entry & modify-existing-entry
//  map.insert("abc", 1);   // compile error
    map.insert(std::make_pair("abc", 1));     map.insert(std::make_pair("xyz", 4));
    map.insert(std::make_pair("def", 2));     map.insert(std::make_pair("abc", 5));
    map.insert(std::make_pair("def", 3));     map.insert(std::make_pair("abc", 6));

    for(const auto& x:map) cout << x.first << " : " << x.second;
    auto range = map.equal_range("abc");
    for(auto iter = range.first; iter!=range.second; ++iter) cout << iter->first << " : " << iter->second; // abc:1  abc:5  abc:6
```

How to iterate through all buckets in `std::unordered_map` and eliminate entries return true given a predicate?

```
// This is wrong as iterator may be invalidated after erase
for(auto iter = unordered_map.begin(); iter!=unordered_map.end(); ++iter)
{
    if (predicate(iter->first)) map.erase(iter);
}
// This is ok as erase returns next valid iterator
auto iter = unordered_map.begin();
while (iter!=unordered_map.end())
{
    if (predicate(iter->first)) iter = map.erase(iter);
    else ++iter;
}
```