Chapter 14 Transition to C++

C++ supports object oriented programming (OOP) : objects & classes

- class ~ type, object ~ variable
- objects are instances of classes.

■ Class

1. Syntax:

```
class MyClass {
    private:   // access control
        static int common;   // one location shared by all instances
        int x;
    public:
        void setX ( int new_x ) {  // member function / method.
            x = newX;
        }
        int getX ( ) const {   // const T * (this)
            return x;
        }
};
int MyClass :: common = 0;   // actually create the variable.
```

2. Access control

① Three levels of access / visibility:

• public	accessed by any piece of code	default in struct
• private	accessed by code within the class	default in class.
• protected	→ inheritance	

② Once an access specifier is explicitly written, the level of visibility persists until we write another one.

3. Encapsulation: member functions / methods.

↳ classes encapsulate the data and methods that act on them.

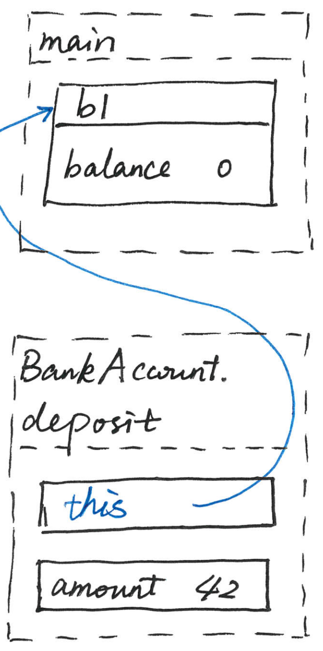
① Extra parameter passed to method: this.

"this" is a pointer to itself, i.e., the object that the method is inside.

as if: T * const this

⊗ • All references to fields in the object is implicitly this → field.

• pass "T* const this" along with any other parameters to method.

Actually	Implicitly	Memory
`class BankAccount {`	⋮	 main b1 balance 0 BankAccount.deposit this amount 42
`private:`		
`    double balance;`		
`public:`		
`    void deposit(d amount)`	`void deposit( BankAccount *`	
`    { balance += amount;`	`  this, double amount ){`	
`    }`	`  this -> balance += amount;`	
	`}`	
`    void initAcount(){`	`void initAccount(B.A * this){`	
`        balance = 0;`	`   this -> balance = 0;`	
`    }`	`}`	
`};`		
`int main(void){`	⋮	
`    BankAccount b1;`		
`    b1.initAcount();`	`b1.initAccount(&b1);`	
`    b1.deposit(42);`	`b1.deposit(&b1, 42);`	
`    return 0;`	⋮	
`}`		

Note: the functions are not actually inside the object in memory. The code for these functions is just placed with the rest of the code in memory.

② Const Methods

↳ • const T * const this

• cannot modify anything in the object "this" points at.

⋮

```
int MyMethod ( ) const {
        /* implementation */
}
```

Note: cannot envoke a non-const method on a const object.

4. Static members:

↳ All instances of a class share the same memory location for a particular field (~~variable~~ or method)

① put " static " keyword before that field.

② put the line " type MyClass :: static_member = value " at the global scope. i.e.,

- outside of any function or class
- execute before main.

which actually creates the box.

③ static methods cannot access non-static members of the class, as they have no " this " passed to them and they operate on no particular object.

5. Classes can contain other types.

① typedef of an existing type.

referenced outside of the class: OuterClass :: own_t

② inner class

- inner class declared as "private" cannot be used outside of the outer class.
- Methods of an inner class do NOT have direct access to non-static members of the outer class.
- Let the inner class reference the outer class: have a field in the inner class that is a pointer to the outer class type, and initialize it to point at the appropriate outer class instance.

```
class OuterClass {
  public:
      typedef int own_t;
      own_t var;
  private:
      class InnerClass {
          public:
              own_t x;
              int y;
              ~~void~~ own_t getX() {
                  return ~~x->~~ x;
              }
              OuterClass * p;
      };
      :
};
```

6. Plain old data (POD) type.

↳ classes or structs can be POD or non-POD.

① POD:
- have direct C analogs
- sequences of field in memory (can be malloced, freed, etc).
- Declaring functions inside of a class / struct won't make the type non-POD, unless they are virtual functions.

(classes in C++ & structs in C have the same layout in memory)

② non-POD: if you cannot write it in C, it is possibly not POD.
- Declaring field private.
- has members of a non-POD type. e.g., reference.
- Cannot work on the "raw" memory directly in a safe way; have to use C++ operators.

7. Naming conventions: classes are nons, methods are verbs.

■ References.

↳ conceptually, another name of a box.

Code with References

```
int f ( int & x, int y) {
        int z = x + y ;
        x = z ; // use
        return z - 2 ;
}
int g ( int x ) {
        int z = f ( x, 3 ) ;
        int & r = z ; // initialization
                          declaration
        r ++ ;
        return z * x ;
}
```

Equivalent Code with Pointers

```
int f ( int * const x, int y) {
        int z = ( * x ) + y ;
        ( * x ) = z ;
        return z - 2 ;
}
int g ( int x ) {
        int z = f ( & x, 3 ) ;
        int * const r = & z ;
        ( * r ) ++ ;
        return z * x ;
}
```

1. Differences between references and pointers:

① const: a reference cannot be changed to refer to a different box.

② must be initialized when it is declared.

③ automatically dereferenced (don't need * or & when used)

④ cannot have NULL references.

⑤ cannot have a "reference to reference"

⑥ cannot have a pointer to a reference.

e.g., int * & x (✓), int & * x (X)

⑦ cannot perform "reference arithmetic".

2. * or & is not needed when used.

⇓

a call to a function which returns a (non-const) reference may be used as an lvalue.

e.g.,
```
int & getX (T* p){
        return p->x;
}
   :
getX (p) = 3;
```

```
int * const getX (T*p){
      return &(p->x);
}
*(getX(p)) = 3
```

3. Translation between references and pointers have the following exceptions:

① a const reference may be initialized from a non-lvalue.

e.g.,
```
void someFunc ( const int & x ) {   // only legal if it's const.
        /* implementations */
}
   :
someFunc (3);
```

What actually happen is that the compiler creates a temporary "const int" variable, and passes the address of that variable.

② pointers and references are distinct types.

We can define an operator which operates on two references, but not on two pointers.

■ Namespaces

In C :

- functions and types names are visible throughout the entire program.
- declaring a function as "static" restricts it visibility to the compilation unit, and may not be used in any others.
- Problem : name collisions

In C++ :

- introduces namespaces that create named scopes.

1. Declaration :

   wrapping the declarations with namespace somename { }

2. C++ standard library declares in "std" namespace; other libraries provide various namespaces.

3. Usage :

   ① scope resolution operator :: to reference them in their fully qualified names.

   e.g., std::vector

   ② using keyword to open the namespace

   e.g., using namespace std;

   ```
   using namespace std;
        ⋮
   vecter
   ```

   or : using std::vector; // preferable.

   ```
   using std::vector;
        ⋮
   vector
   ```

■ Overloading.

I. Function overloading

↳ allow multiple functions with the same name.

1. Only legal if the functions can be distinguished by their parameter types.
2. The compiler must also be able to determine an unambiguous best choice.

   e.g.,
   ```
   int f(int, double)
   double f(double, int)
   f(3,3)
   ```
3. Guidelines:
   - Only overload functions when they perform the same task, but on different types.
   - Only in such a way that what the "best choice" is straight-forward.

II. Name mangling

↳ adjust the function names to encode the parameter type information, as well as the class and name space.

- it happens in C++, but not in C.
- For C/C++ mixed code, declare the functions inside of

  ```
  extern "C" {
        /* function prototype. */
  }
  ```

  to disable name mangling, i.e., compile with a C compiler.
- "main" is treated specially (as if in extern "C")

III. Operator Overloading.

↳ define the behavior of the operators when at least one user-defined type is involved.

Example:

```
class Point {
private:
    int x;
    int y;
public:
    Point operator+ (const Point & rhs) const {
        Point ans;
        ans. x = x + rhs. x;
        ans. y = y + rhs. y;
        return ans;
    }
};
```

Example:

```
Matrix & operator += (const Matrix & rhs) {
        ⋮
    return *this;
}
```

1. Two operands: ① this (implicitly)

   ② const $\underbrace{\text{reference}}$ to the right-hand operand

   ↓ pass const instead of the object to avoid copying.

2. In case of operators which modify the object,
   e.g., +=,
   they return a reference to that object: *this (T&)
   where the address is implicitly taken.

3. Overloading operators should obey common sense.

4. const versus non-const functions / operators constitute valid overloadings.

Headers

1. Standard headers : starts with c & does not end with .h

   e.g., cstdlib ~ stdlib.h, cstdio ~ stdio.h.

2. class declarations typically occur in header files, with very short methods directly written in. e.g., accessor method.

   Then write the remaining implementation in the .cpp file. with the fully qualified name (::)

Default values

- default values can be specified for some or all of the parameters.

  e.g., int f (int x, int y=3, $\underset{\sim}{\text{bool}}$ b= false)

  ↳ new C++ type for true / false.

- Arguments for certain parameters can be omitted if default values are desired.

  e.g., f(9), f(9, 6), f(9, 6, true)

- The omitted arguments must be the rightmost arguments of the call.

- Default values must be specified in the prototype in the header file.

Chapter 15 Object Creation & Destruction.

■ Object creation – Initialization

$\Uparrow$

Constructor :
- automatically called by C++ when the "box" for the object is created (not including the creation of a pointer)
- cannot be called by the programmer, only during the object creation / declaration.

1. Types of constructor – overloading
   - default : no arguments.
   - single argument : add explicit keyword
   - can be overloaded by multiple constructors

2. Initialization : initializer list

   ① syntax :

   $$\text{ClassName} :: \text{ClassName} (\dots) : \text{name} (\overset{\overset{\text{initialization}}{\downarrow}}{\text{values}}), \ \text{name} (\text{values}) \overset{\overset{\text{assignment.}}{\downarrow}}{\{\}}$$

   ② initialization v.s. assignment
   - { init : creating
     | assign : creating → overwriting (slower. unpredicted)

     C++ ensures all fields have some forms of initialization before { } .
   - If not specified in the initializer list, the field is default initialized.
   - Must initialized in the initializer list : { a reference
     | a const field.

   ③ initializing order = the order in which they are declared.

	Syntax	Default	Trivial if ...
methods	double FuncName (args) {…}		

constructor	① ClassName ( ) : init lists {…}
	name (value), name (value)
	② ClassName (args) : init lists {…}
	* explicit ClassName (one arg) …
		ClassName ( ) { }	∘ Not declared ~~by~~ explicitly (automatically supplied)
			∘

destructor	~ClassName ( ) {
	/* if any special treatment, e.g., delete */
	Also written in copy constructors /=
	}
		~ClassName ( ) { }	∘ Automatically supplied
			∘ all fields have trivial destructor
			∘ actually does nothing

Copy Constructor	ClassName (const T & …) : init lists {…}	ClassName (const T& …) : every-field { }	∘ Automatically supplied ∘ all fields have trivial copy constru ∘ simply copy the bytes in memory

Assignment operator	T & operator= (const T &…) {…
	if (this != & rhs) {
	…
	}
	return * this ;
	}
		T & operator= (const T & …)
		{ /* Do not check for self assignment , just copy each field in the class */
		}
			∘ Not overloaded explicitly
			∘ copy each field in the struct.
			∘ do not check for self assignment.

Class is not-PoD if constructor / destructor / copy constructor is defined.

④ Type of initialization.

value initialization	default initialization
`ClassName * ptr = new ClassName();` // parenthesis	`ClassName *ptr = new ClassName ;` // no parenthesis
A class with a default constructor ↳ initialized by its default constor. A class without any constructor ↳ every field value initialized Non-class types ↳ zero-initialized Arrays ↳ their elements value initialized.	non-POD types ↳ initialized by their default constructors POD types ↳ uninitialized. Arrays ↳ their elements default initialized.

3. Object creation

① As local variables :

```
ClassName  ObjName ( ) ;   // using default constructor
ClassName  ObjName ( arguments ) ;   // using overloaded constructor
```

② As unamed temporaries :

```
ClassName ( arguments ) ;  // destructed after ;
```

③ Dynamic Allocation :

- `ClassName * ObjPtr = new ClassName ( arguments ) ;  // delete ObjPtr`
- `ClassName * ObjArray = new ClassName [n] ( ) ;  // delete[] ObjArray`

  can only use the default constructor

-
  ```
  ClassName ** ObjPtrArray = new ClassName * [n] ( ) ;
  for ( int i = 0; i < n; i++ )
      ObjPtrArray [i] = new ClassName ( arguments ) ;
  ```

new / delete v.s. malloc / free :

- new returns an object * pointer, malloc returns a void * pointer
- If an array is allocated, new[] / delete[] should be used, as they return an (object × n) spaces, not like new / delete. malloc / free records the allocated spaces anyway, so ~~the~~ no ~~specific~~ special treatment is needed.

4. What to do :
    - Make the class default ~~d~~ constructible.
    - Use initializer lists to initialize your class's fields.
    - Explicitly initialize every field in the initializer list.
    - Initialize the fields in the order they are declared.
    - Use new and new[], not malloe.

General dynamic allocation in C++ :

```
ptr = new type ;                          delete ptr;
ptr = new type [ num_of_elements ] ;      delete [] ptr;
```

■ Object Destruction

⇑

destructor : • automatically invoked whenever the "box" for the object is about to be destroyed.

1. Structure of the destructor :
   - Must no parameter list
   - Typically contains memory deallocation (delete) of its fields. (Rule of Three)

2. The destruction happens when :

   ① the memory for the object is deallocated through (if created by allocation):

   delete ObjPtr or delete[] ObjArray

   ② (if the object is a local variable) the object is going out of scope.

   e.g., the object is created in a { }, and the execution arrow goes out of that { } (can be a for loop, if/else, function, etc.)

   ③ the enclosing object that contains it is destroyed.

3. What are destroyed :

   ① default destructor — fields inside of the class.
   - class : the destructor for that class is invoked before destroyed.
   - pointer to an ~~class~~ object : that destructor will NOT be destructed.

   ② actions specified in the destructor will be performed.

4. How the destruction happens :

   ① Each of the fields is destroyed in the reverse order in which they were initialized.

② (If destructed by delete[ ]), the elements of the array have their destructor invoked in the decreasing order of index.

③ (If ~~are~~ created as local variables) If multiple variables go out of scope at the same ~~time~~ point, their destructors are invoked in the opposite order from the order in which they were constructed.

As a general rule, whenever construction & destruction occur in a group, the order of destruction is the opposite of the order of construction.

■ Object Copying

Do not :
- make a shallow copy by "p1 = p2" without overloading = .
- directly pass an object to a function by "func(p)", object in which will be destructed after the function returns.

copy constructor	copying assignment operator
copying during initialization ↓ a new object is created as a copy of an old one.	copying during assignment ↓ changes the value of an object which already exists (and is initialized) to be a copy of another object.

Declaration :
`ClassName ( const ClassName & rhs )`
`: initializer_list {`
`    /* making deep copy */`
`}` ~~create new object~~.
	Declaration :
	`ClassName & operator= ( const ClassName & rhs ){`
	`    if ( this != & rhs ) {`
	`        /* make deep copy, create new (tmp) field ~~(objects)~~ */`
	`        /* delete original memory (objects), and assign ~~use~~ the tmp one */`
	`    }`
	`    return * this;`
	`}`
	↓ put all assignment, including non-dynamically-allocated fields, at the very end, to avoid using new parameters to delete old ones.

Usage :
`ClassName ObjA ( ObjB );`
or
`ClassName ObjA = ObjB; // not prefered`
	Usage :
	`ObjA = ObjB;`

Rule of Three : if a class needs special copying behavior (e.g., making deep copy), it almost certainly have resources that need to be cleaned up by a destructor.

■ Unamed Temporaries

↳ values created as the result of an expression, but not given a name.

1. Declaration of an object with no name :

    ClassName (argument) ;

2. Deallocation of an unnamed temporary object :

    at the end of the full expression.

3. Application :

① Parameter passing : Fuction (ClassName(argument)) ;

Note : It's generally preferable to have functions take (const & ClassName)

Compared to creating a temp object then passing it, in this way the compiler is allowed to optimize the copy (directly create it in the function frame)

② Return values : return ClassName(1) + ClassName(2) * ClassName (3) ;

return value optimization : directly initialize the unnamed temporary object within the expression (function)

③ Implicit conversion : a simplification of parameter passing.

```
SomeFunc ( classname ( arg )) ;
        ⇓
SomeFunc ( arg ) ;  // only valid for single-argument constructor
                       that are NOT declared as explicit
```

As a general rule, all of the single-argument constructor should be declared as explicit except the copy constructor.

Chapter 16 Strings and IO Revisited.

■ String Class

↳ std::string, #include<string>

Some features:

1. constructor

default	string();
from C-string	string(const char * s), string(const char* s, size_t n) → the first n chars
fill	string(size_t n, char c)
copy	string(const string & str)
substring	string(const string & str, size_t pos, size_t len)

2. properties

size() / length() :	return length in byte (size_t)
clear() :	erase the contents and make it empty
empty() :	return true if empty

3. modify

push_back(char c) :	append c to the end of string.
append(......) :	append char/str/substr to the end of string acceptable argument type = constructor
insert(size_t pos, ...) :	insert char/str to the position pos.
swap(string & str) :	exchange the contents with str.
pop_back() :	erase the last character.

4. Access

substr(size_t pos, size_t len) :	get substring. default (0, npos)
find(..., size_t pos = 0) :	find the first accurance of char/str from position pos.

5. operaters

= : str = " " / ' ' / anotherstr.

+= : append " " / ' ' / &str to string.

[ ] : get character

Performance issue:

- It's easy to write code that copy values excessively, and creates / destroy many temp objects.

  e.g., passing objects ~~as~~ arguments, using substr( ), etc.

- Considering : - pass parameters by reference instead of value.
  - eliminate creating temporary objects.

■ Streams

↳ std:: ostream, std:: istream, # include < iostream >
  - ostream ↓ cout, cerr
  - istream ↓ cin

  std :: ifstream, std:: ofstream, std:: fstream, # include < fstream >
  - ifstream ↓ read
  - ofstream ↓ write
  - fstream ↓ r/w.

  std :: stringstream, # include < sstream >

1. Output :

$$\text{std::cout}\ \underbrace{\texttt{<<}}_{\text{Stream insertion operator}}\ \cdots\ ;$$

or std :: cerr << … ;

I • return type : left operand, i.e., std::ostream & (cout will be changed by each <<)

II • formatting control :

① manipulator → inserting special values.

e.g., change base : std::cout << std::hex / oct / dec ;
std::cout << x << "\n" ;
to output an int in hex / oct / decimal base.

* It's good practice to put a stream back into decimal mode afterwards.

e.g., flush the buffer : std::cout << std::flush ;
e.g., insert null / ~~EOF~~ \n : std::cout << std::ends / endl.

② via methods in the stream class

• width (streamsize wide)

e.g., std::cout.width(10) << 100 << '\n' ;

```
_______100
```

std::cout.fill('x').width(10) << 100 << '\n' ;

```
xxxxxxx100
```

std::cout << std::left << 100 << '\n' ;

```
100xxxxxxx
```

• precision (streamsize prec)

std::cout.precision(5) ; ~~≠~~

std::cout << 3.14159 << '\n' ;

```
3.1416
```

std::cout.precision(10) << 3.14159 << '\n' ;

```
3.14159
```

std::cout.setf(std::ios::fixed, std::ios::floatfield);

std::cout << 3.14159 << '\n' ;

```
3.1415900000
```

- setf / unsetf

fmtflags setf (fmtfl, mask) :

format flag value	field bitmask
left, right or internal	adjustfield
dec, oct or hex	basefield
scientific or fixed	floatfield

fmtflags setf (fmtfl) :

boolalpha, showbase, showpoint, showpos, unibuf, uppercase

unsetf (...)

corresponding to setf. Clear the format flag.

~~4~~. III. Overloading <<

① Written outside of the class :

left-hand operand ⇑ right-hand operand ⇑

```
std:: ostream & operator << ( std:: ostream s , const MyClass & rhs ){
```

(std:: ostream s) omitted if written in the class, as "this" is implicitly passed

```
        for( ){
            s << sth;
        }
        return s;
}
```

② Prototype declared as friend in the class, to have access to the private fields.

```
class MyClass {
        friend std:: ostream & operator << ( std:: ostreams , const
                                MyClass & rhs );
```

2. Input

`std::cin >> ... ;`

$\underset{\sim}{}$ ↳ stream extraction operator.

I. Read input.

- formatted.   e.g.,

```
int x;                | string s;
std::cin >> x;        | std::cin >> s;
```

↓ Stop at white space

- unformatted:

```
.get ( char & c );
.get ( char * s , streamsize n, (char delim));
.getline ( char * s , streamsize n, (char delim));
```

e.g.,

```
char str[256];
std::cout << "Enter the file name";
std::cin.get(str, 256);
```

II. Errors

- When an operation produces an error, it sets the stream's error state.
- The error state persists and causes future operation to fail, unless explicitly cleared.
- Methods:

good ( ) — check if any errors happened.

clear ( ) — clear the error state.

failures: eof ( )

fail ( ): unable to perform the proper conversion.

bad ( ): errors in I/O operation.

3. Files

- constructor $\begin{cases} \text{default: } & \texttt{fstream ( );} \\ \text{initialization: } & \texttt{fstream (const char * fname);} \end{cases}$

- methods :

  open (const char * fname) ;

  close ( ) ;

  is_open ( ) ; // check if it's opened successfully

  get (...) / getline ( ) / ...

- work well with ostream operators.

  ofsteam ~~is~~ ∈ ostream .

  fstream ⇔ ostream / istream .

4. string streams

↓

temporarily "store" ~~the~~ / "buffer" the input / output for futher manipulation on the string / formatted input / output.

① built up a string by using << s.

then transfer to a string using .str() method .

② pull apart a string as formatted input , the extract the pieces by using >> s.

e.g., std::stringstream ss ;

ss << 100 << ' ' << 200 ;   std::string S= ss.str( ) ;

int foo, bar ;

ss >> $\underset{\underset{100}{\downarrow}}{\text{foo}}$ >> $\underset{\underset{200}{\downarrow}}{\text{bar}}$;

Chapter 17 Templates

template is an instance of parametric polymorphism. an ability of the same code to operate on different types.

Templated Functions

* entirely written in header files.

```
              template parameters        type parameters,
template < typename T, typename S, int N >
              (= class)
T * someFunction ( T & t, const S s ){
        /* normal implementation. Just use T/S as type name */
}
```

- Template parameters have a scope of the entire declarations.
- An actual function is created after the template is instantiated. i.e., giving it the "values" for the parameters.

```
                             must be compile-time constants.
someFunction < int*, double, 3 > ( arrPtr, 3.14 );
                      ↑
            instantiation + function call.
```

- The resulting function is called template specialization.
- It is type-checked ~~the first time~~ at the time of instantiation.
- Template parameters may be inferred (not explicitly given), but not recommended.

Templated Class

- Specializations created by ~~tu~~ different instantiations are different classes.
- "friend" functions included in the ~~et~~ templated class declaration should be declared as templated function, e.g.,

```
template < typename T >
class MyClass {
    private :
        T * data ;
    public :
        MyClass( ) : data ( NULL ) { }
        My Class ( const MyClass & rhs ) : data ( new T ) {...} .
                  ⋮
        template < typename S >  // use a different typename to avoid conflict.
        friend std:: ostream & operator << (std::ostream & s , const MyClass<S>
                 & rhs ) ;
};

template < typename T >
std:: ostream & operator << ( std:: ostream & s , const MyClass < T > & rhs ) {
          ⋮
}.
```

- Type defined by template classes can be used ~~as~~ as type parameters in template parameters :

```
template < typename T ,  template < typename > class S  >
class  OtherClass {
        private :
            S < T > ;
              ⋮
}
```

- Specialization (and type-ckeck) for templated class occurs in parts :

  ① specialized at ~~a~~ the creation of an instance of the class :
    - fields
    - virtual methods

    i.e., the part that are represented in memory.

  ② specialized when the corresponding method is used :
    - non-virtual methods

    as they're not placed in the object.

  Therefore, methods in the template can only work on certain types but not on others.

- Template parameters for Classes can not be inferred. i.e., the < arguments > must always be explicitly specified.

■ Other Rules for Template

1. Dependent Name

   ↳ Whether $\boxed{T::x}$ is a type or a variable depends on what T is.

   When used as a type, typename keyword must be placed before T::x

   i.e.,

```
template <typename T>
Class MyClass {
    private :
        typename T::t field1;
        int field2;
            :
}
```

2. You can provide an explicit specialization for a template, which could behave completely different from the primary template. It may be partial or complete with respect to the parameters.

■ The Standard Templated Library (STL)

1. Vector

↳ std :: vector < T > , # include < vector >

constructor :

vector<T> () ; // empty

vector<T> (n, val) ; // $[\underbrace{val, \dots, val}_{n}]$

vector<T> (another_vector & s) ;

capacity :

. size ( ) ; // return the size ;

. resize (n) ; // change the size + truncate / default initialized.

. empty ( ) ; // return if it's empty.

modify :

. push_back (ele) ; // append the ele to the end.

. pop_back ( ) ; // delete last element.

. swap (another_vector) ;

. clear ( ) ; // destruct.

. insert (it pos , T ele) ;

. erase (it pos) ; // remove an arbitrary element.

operator :

= : assign new contents, ~~p~~ replace.

[ ] : access element by index ~~(type size~~

== : if two vectors are identical

< : order the two vectors lexicographically

} ==, < have to be defined for T.

2. Iterator :

↳ a class encapsulating the state of the internal traversal.

typically internal to a data structure.

e.g., std :: vector<T> :: iterator class.

```
. begin( ) ;  // return a (const) iterator pointing the first element
. end ( ) ;   //  - -   - -   - -  - - - - - - last - - - -
* it ;        // dereference the element
```

note: ① if the container is empty . begin( ) shall not be dereferenced.

② it may be invalidated after modification to that element.

Example of iteration :

```
void printElement ( const T & container) {
        typename T:: const_iterator it = container. begin( );
        while ( it != container.end() ) {
                std:: cout << *it << "\n";
                ++ it ;  // prefix increment operator, more efficient that "it++".
        }
}
```

3. Std :: pair

↳ # include < utility>

to pair two objects together into a single object, e.g., return two values.

```
std:: pair< T1, T2 > myPair ( ) ; // default
                     myPair ( var1 , var2 ) ; // initialization.
                     myPair ( anotherPair) ; // copy.
std:: cout << myPair.first << "," << myPair. second << '\n' ;
```

4. Algorithms in STL

↳ # include < algorithm >

1. Make use of iterators

e.g.,
```
template < typename It >
void sort ( It first , It last ) ; // sort [ first, last ) in ascending order

std::vector< std::string >  str1 ( ) ;
for ( int i = 0 ; i < 3 ; i++ )
        str1 . push_back ( "No" + i ) ;
sort ( str1 . begin() , str1 . end() ) ;
```

2. Make use of user-defined function object

↳ an object with " function call " behavior .

e.g., an overloaded function call operator ( )

( ) can take arbitrary number of parameters .

↑

e.g.
```
class smaller ~~larger~~ {
    public :
        bool operator() ( const int & a , const int & b ) const {
            return a < b ;
        }
};

int main ( ) {
    smaller order ; // create an instance of the functor
    if ( order ( 3,4 ) ) // using the overloaded ( ) that looks like a function
        std::cout << " 3 is smaller than 4 \n" ;
    return 0;
}
```

The function object can be passed into function templates provided by < algorithm > . i.e.,

```
# include < algorithm >
# include < vector >
# include < iostream >

// define a function
bool myFunction ( int i, int j ) { return i<j ; }

// define a function object
class myClass {
    public:
        bool operator( ) ( int i , int j ) { return i<j ; }
}

int main ( ) {

        int myInts = [ 22, 30, 6, 9, 7 ] ;
        Std:: vector < int >  myVector ( myInts , myInts + 3 );

        // using default comparison
        Std:: sort ( myVector. begin( ) , myVector. end( ) );
        // using function as comp
        Std:: sort ( myVector. begin( ) , myVector. end( ) , myFunction ) ;
        // using function object as comp
        Std:: Sort ( myVector. begin( ), my Vector. end( ) , myClass ( ) ) ;

        return 0 ;
}
```

e.g.,
```
std::min < std::string, myClass > ( str1, str2, myClass() );
                        ↑                       ↑
                      type.              function object
```

Chapter 18 Inheritance

A child class / subclass inherits from or extends the parent class/superclass when the child class is a parent class.

The child class can have more fields & methods of its own added to it, and can override the behavior of the methods it inherited.

■ Inherited class

1. Declaration :

for (class B : public A) :
Although B cannot access A's private field directly, it may still have public method in A that calls that private field. So it is still included in B's layout.

(1) Access specifier

• public , private, protected :

	public member	private member becomes :
public	public	private
private	private	private
protected	protected	private

⇓

① can be accessed by members of the class, or by members of any of its child class. Code which is outside them are unable to access them.

② can only be accessed through a pointer / reference / variable of the child class's own type.

(2) inherited field:

• Do not redeclare the fields or methods that inherited.

• If do re-declared, then there are two different fields of the same name (but different fully qualified names)

Example:

```
class A {
    private:
        int x;
        double y;
    public:
        double sum();
};
```

```
class B : public A {
    private:
        int a;
        double b;
    public:
        :
};
class C : public B { };
```

2. Construction

(1) Constructors are running from parent → child.

i.e., the parent-most ansestor is first run to initialize that portion of the object; then the next closest constructor is run. … .

(2) The type of an object changes during the construction process.

- The allocat~~ored~~ed space is for C
- The type changes: $\boxed{A \xrightarrow[\text{finishes}]{\text{A's constructor}} B \xrightarrow[\text{finishes}]{\text{B's constructor}} C}$

(3) Overloaded constructor in parent's class:

- Explicitly call a ~~cons~~ parent's constructor:

```
class A {
    int x;
  public:
    A() : x(0) {}
    A(int _x) : x(_x) {}
}
```

```
class B : public A {
    int y;
  public:
    B() : y(0) {}
    B(int value) : A(value),
                   y(0) {}
}
```

By writting the call to the parent class's constructor as the first element of the initializer list.

- Not explicitly specify a call to the parent's constructor:

  [default constructor of the parent is implicitly called.
    if no default one or it's private, then compiler will produce an error.

3. Destruction

(1) The process is the reverse of construction:

$$C \xrightarrow[\text{finishes}]{\text{C's destructor}} B \xrightarrow[\text{finishes}]{\text{B's destructor}} A \rightarrow \text{destruct } A.$$

However, it stops if any parent class's destructor is trivial.

(2) ~~Overloaded de~~ No issue of specifying which destructor to call.

(3) Declaring destructors as virtual whenever using class polymorphism.

■ Subtype Polymorphism

↳ the same code can perform on one type and its subtypes.

1. for Inheritance

- A child class is a sub-type of its parent class.

  ⇓

  Anything legal to do to the parent class is also legal to do to the child.

- Access modifier restriction:

  public: polymorphism may be freely used anywhere
  private / protected: only ~~for~~ when fields with that access restriction could be used.

- Polymorphism is only applicable when taking pointer or references as ~~parameters~~ arguments. because parent / child may have different size, but their pointers have the same size

2. Static type v.s. Dynamic type

static : - type obtained by the type checking rules (what's written in the code)
- the compiler only works with the static types . i.e., uses the declared types of variables for compiling.

dynamic : the type of object that's actually pointed at (what's executed).

3. Advantages

(1) Compiler allows implicit conversion from a child class to a parent class .

(2) Parent & child classes can be included in the same vector / set / other data structures, and be iterated with one iterator.

$\downarrow$ of parent's type.

(3) If a new child class type is ~~need~~ added, nothing needs to be changed if they uses the parent's type.

■ Method Overriding : Virtual method.

↳ child class specifies a new behavior for a method inherited from its parent.

1. Static v.s Dynamic dispatch

	how to override	which method to call ?
static dispatch	write another method with the same name.	have the static type to determine. (i.e., what's written in the code)
dynamic dispatch	declare the method as virtual in both the parent and the child	determined by the type of object the pointer actually points to .

2. Declaration of virtual method

e.g.,
```
class A {
    public:
        virtual void sayHi { std::cout << "Hello from class A \n" ; }
};
class B : public A {
    public:
        virtual void sayHi { std::cout << " Hello from class B \n" }
};
int main {
      A anA ; B aB ;
      A * ptr1 = & anA ;
      A * ptr2 = & aB ;  // static : A , dynamic : B
      ptr1 -> sayHi ;    // "Hello from class A"
      ptr2 -> sayHi ;    // "Hello from class B"
      return 0 ;
}
```

- The declaration of virtual must appear in the parent class.
- Once declared, it remains virtual in all child classes and their children, even if not explicitly declare so.
- Class that contains virtual method is not POD.
- Declare destructors as virtual whenever using subtype polymorphically.

3. Other notes :

- Use fully qualified name if want to call the parent class' version of a method.

- The access restriction of an overridden method may be more permissive, but cannot be more restricted.

  e.g., (✓) parent - private, child - public

  (X) parent - public, child - private.

- The return type of an overridden method in the subclass can be a subtype of the return type in the superclass. (covariant)

  But only for pointers and references.

4. Abstract Methods and Classes

abstract method / pure virtual member function : defined in the parent class and any child classes must override it ~~to~~ with a real impleme tion

(1) Declaration of an abstract method :

Example :

```
class Shape {
public:
        virtual bool containsPoints ( const Point & p ) const = 0 ;
}
```

(2) Rules about an abstract class

↳ when a class has an abstract method in it

- An abstract class cannot be instantiated.

  but a (A *) or (A &) can be used to polymorphically reference an instance of a concrete subclass of A.

- Any subclass of an abstract class is also abstract unless it defines concrete implementations for ALL abstract methods.

- Any object actually instantiated will (and must) have an implemetntation for all of methods declared in it.

- Abstract class can have constructor, but it cannot calls an abstract method.

■ Inheritance & Templates

1. Composable Aspects:

- parent & child may both be a templated class.

Example:

```
template < typename T , typename S >
class myVector : public std::vector<T> {
        S var;
          :
};
```

mixin: parameterize a class in terms of what its parent is:
(see also Chp 29)

Example:

```
template < typename T >
class myClass : public T {
              :
};
```

- Templated classes may have virtual methods.

2. Not composable Aspects

- A templated method cannot be virtual.

→ Alternative solution for this functionality is :

```
class myClass {
    protected :
        template < typename X >
        int myfunc_implementation ( const X & arg ) { ··· }
    public :
        virtual int myfunc ( const int & arg ) {
            return myfunc_implementation <int> (arg);
        }
        virtual int myfunc ( const double & arg ) {
            return myfunc_implementation <double> (arg);
        }
            ⋮
};
```

- A templated function cannot override an inherited method.

  instead. it creates a new method of the same name.

- Virtual methods are specialized when instantiating a class

  Other methods are specialized only when used

Inheritance Hierarchy Planning

- UML class diagram : used to show class hierarchies (relationship of classes) and describe ~~*~~ fields / methods / visibility / etc.

- MVC (Model, View, Controller) , UI Delegate (view, controller)

$$\begin{tikzcd}
\text{Model} \arrow[d] & \text{View} \arrow[d] & \text{Controller} \arrow[d] \\
\text{data+state} & \text{drawing the state} & \text{receiving input + updating model accordingly}
\end{tikzcd}$$

Chapter 19 Error Handling & Exceptions

- (X) silent failure : produce the wrong answer w/o informing the user
- (–) assert, abort
- (V) bullet proof code (code which can gracefully handle any possible problem)

C++-style error handling : have a function propagates an error to its caller.

■ Throw & Catch Exceptions.

1. Concepts :

- exception object : a temporary object in unspecified storage that is constructed by "throw" expression. (literally can be any type)
- throw exception : signals an erroneous condition and execute an error handler.
- try block : a compound ({} enclosed sequence of statement) statement associating one or more exception handlers
- catch block : exception handlers for matched exception type.

2. Typical structure :

```
try {
        // code that might throw an exception
        // e.g.,  throw any_exception_object();
}
catch ( any_exception_object & e ){   // or catch (...) to accept any type.
        // code that handles that type of error.
}
```

3. When an exception is thrown :

↓

the exception is potentially copied out into some location that will persist through handling.

↓

execution arrow jumps to try{...}⬆. Any variables that go out of scope are destructed.

← If not directly in try-block, then e propagates out of that function.
function frame and any objects in it will be destroyed as if func. returns.
(destructors is called)
↓
unwind~~ing~~ the stack

↓

search for catch( ){ } that matches the e type.

← if not found

↓ if found

catch block handle the exception :

① bind the exception to the variable (reference) declared in ( )

② if encounters throw;

← re-throw the exception ;

else

code is executed as normal;

③ when/if reaches ~~at~~ catch{}⇧, the exception object is deallocated.

Execution continues normally at the statement immediately after.

Note : If, during the unwinding of the stack, one of the destructors of the objects in the frame itself throws an exception, and it propagates out of the destructor, the program crashes.

3. exception type

Generally, what's being thrown should be subtypes of Std :: exception (#include < exception >)

① built-in subtypes : #include < stdexcept >

② user-defined class that inherits publicly from them.

```
e.g., class Divide_by_Zero : public std::exception {};
                                                  ↓
                                        override what() is needed
      int divide ( int a, int b ) {
              if ( b==0 )
                      throw Divide_by_Zero();
              return a/b;
      }
      int main () {
              int a = 3 , b=0 ;
              int c ;
              try {
                      c = divid ( a, b ) ;
              }
              catch ( Divide_by_Zero & e ) {
                      Std:: cerr << " Cannot divide 0 \n" ;
              }
              return 0;
      }
```

Subtype polymorphism takes effect.

4. try-block variation : function try block , used in a constructor (usually)

↳ catch exceptions thrown in the initializer list, or the body.

- the exception is automatically rethrown.
- Anything successfully constructed (including the parent's portion) is destroyed before entering the handler.

Example:

```
class myClass {
        int a;
        int b;
    public:
        myClass() try : a(1), b(0) {
                        ⋮
        }
        catch ( std::exception & e ) {
                        ⋮
        }
              ⋮
};
```

■ Exception specification : as Part of a Function's Interface

↳ a list of the types of exception that it may throw.

- `int f (int x) throw ( std::bad_alloc, std::invalid_argument );`

  ⇒ can only throw the specified exception types.

- `int g (int x) throw ();`

  ⇒ may $\boxed{\text{not}}$ throw any exceptions.

- `int h (int x);`

  ⇒ may throw any type of ~~sp~~ exceptions.

Overridden method may only be more restricted.

■ Exception Corner Cases

$$\begin{tikzcd}
 & \text{① function throws an exception not specified in the exception specification} \arrow[d] & \\
 & \text{call unexpected()} \arrow[d] & \\
 & \text{the specification allows std::bad\_exception ?} \arrow[dl, "\text{Yes}"'] \arrow[dr, "\text{no}"] & \\
\text{handling continues normally} & & \text{call terminate()} \arrow[d] \\
 & & \text{call abort()}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{default} \arrow[d, leftrightarrow] \\
\text{customized via set\_unexpected or set terminate}
\end{tikzcd}$$

② An exception is throw during object construction : see before.

■ Exception Safety

↳ which guarantees the code makes in exceptional circumstances.

↑ Stronger	No Throw	Will not throw any exception. → (for destructors) (It handles all of them inside)
	Strong	No side effect if an exception is thrown. (objects unmodified , no memory leaks)
	Basic	Objects remain in valid state. (no dangling pointer , invariants remain , no memory leaks)
	None	Does not provide any guarantee.

1. RAII : Resource Acquisition Is Initialization:

↳ It's better to have an object in the local frame that is constructed when the resource is allocated, and that whose destructor frees that resource.

Instead of directly call "new" to allocate memory.

Example:

```
X * myFunc ( A & anA ) {
    std:: auto_ptr<X> myX ( new X() ); // as oppose to X*myX = new X();
    anA. someMethod ( myX. get() ); // as oppose to anA. someMethod ( myX )
    return myX. release(); // remove ownership of the pointer.
},
```

So that if an exception is thrown, we don't need to free the allocated memory explicitly.

2. Temp-and-swap

↳ modify an object by creating a temp object, then swap the contents of the newly created object with the original object.