C++ 筆記

Q: 如何通過伺服器使客戶端取得客戶資料卻不會知道其中運作原理以保護伺服器安全？

Ans: header file 裡只定義函數名稱與變數供客戶端查詢及使用，而函數詳細的細節與操作則寫在 class.cpp 裡隱藏起來不讓客戶端看到。

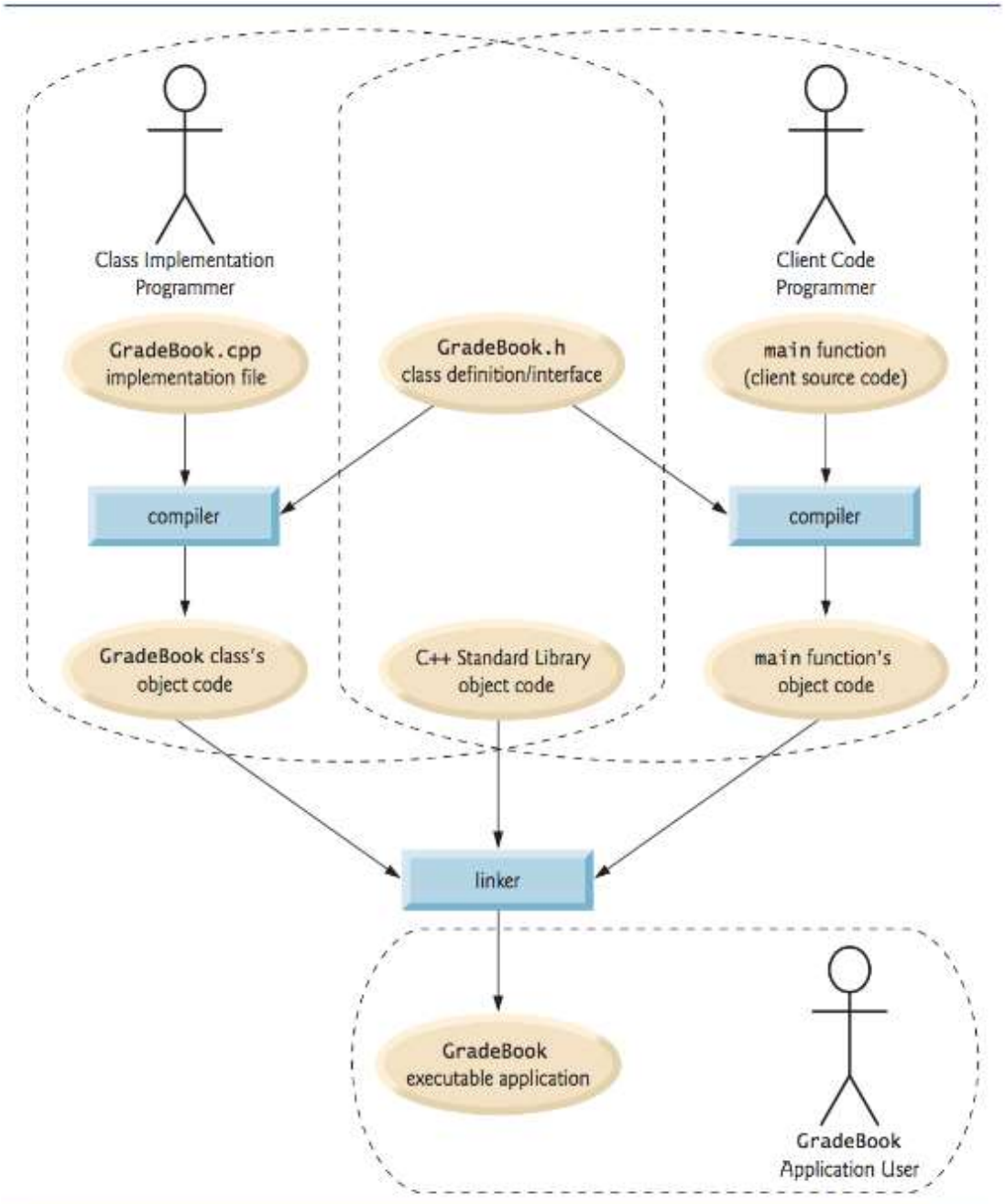


Fig. 3.14 | Compilation and linking process that produces an executable application.

C++ Standard Library

Standard Library header file	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and standard output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output. This header file replaces header file `<iostream.h>`.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header file is first used in Section 4.9 and is discussed in more detail in Chapter 15, Stream Input/Output. This header file replaces header file `<iomanip.h>`.
`<cmath>`	Contains function prototypes for math library functions (discussed in Section 6.3). This header file replaces header file `<math.h>`.
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header file are covered in Section 6.7; Chapter 11, Operator Overloading; Chapter 16, Exception Handling; Chapter 21, Bits, Characters, C Strings and `structs`; and Appendix F, C Legacy Code Topics. This header file replaces header file `<stdlib.h>`.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header file replaces header file `<time.h>`. This header file is used in Section 6.7.
`<vector>`, `<list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The `<vector>` header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these header files in Chapter 22, Standard Template Library (STL).
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file `<ctype.h>`. These topics are discussed in Chapter 21, Bits, Characters, C Strings and `structs`.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header file replaces header file `<string.h>`. This header file is used in Chapter 11, Operator Overloading.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header file is discussed in Section 13.8.
`<exception>`, `<stdexcept>`	These header files contain classes that are used for exception handling (discussed in Chapter 16, Exception Handling).
`<memory>`	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 16, Exception Handling.

Fig. 6.7 | C++ Standard Library header files. (Part 1 of 2.)

Standard Library header file	Explanation
<fstream>	Contains function prototypes for functions that perform input from files on disk and output to files on disk (discussed in Chapter 17, File Processing). This header file replaces header file <fstream.h>.
<string>	Contains the definition of class string from the C++ Standard Library (discussed in Chapter 18, Class string and String Stream Processing).
<sstream>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class string and String Stream Processing).
<functional>	Contains classes and functions used by C++ Standard Library algorithms. This header file is used in Chapter 22.
<iterator>	Contains classes for accessing data in the C++ Standard Library containers. This header file is used in Chapter 22.
<algorithm>	Contains functions for manipulating data in C++ Standard Library containers. This header file is used in Chapter 22.
<cassert>	Contains macros for adding diagnostics that aid program debugging. This replaces header file <assert.h> from pre-standard C++. This header file is used in Appendix E, Preprocessor.
<cfloat>	Contains the floating-point size limits of the system. This header file replaces header file <float.h>.
<climits>	Contains the integral size limits of the system. This header file replaces header file <limits.h>.
<cstdio>	Contains function prototypes for the C-style standard input/output library functions. This header file replaces header file <stdio.h>.
<locale>	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
<limits>	Contains classes for defining the numerical data type limits on each computer platform.
<utility>	Contains classes and functions that are used by many C++ Standard Library header files.

Fig. 6.7 | C++ Standard Library header files. (Part 2 of 2.)

Scope

The portion of a program where an identifier can be used is know as its scope.

Six scopes: function scope, global namespace scope, local scope, function-prototype scope, class scope, namespace scope.

Global namespace: Any identifier declared outside any function or class has global namespace. Its scope covers the entire file.

Function scope: **Labels** (i.e., identifiers followed by a colon such as "start:") are the only identifiers with function scope. Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body. (Appendix F)

Local scope: Identifiers declared inside a block have local scope, and ends at "}". Local variables declared static still have local scope, even though they exist from the time the program begins execution.

Function prototype scope: The only identifiers with function prototype scope are those used in the parameter list of a function prototype. Function prototypes do not require names in the parameter list—**only types are required.** Names appearing in the parameter list of a function prototype are **ignored** by the compiler.

PS. Functions with *empty* parameter list are functions with type void and empty parenthesis, indicating functions neither take arguments nor return values. E.g. void takeGrade (void);

Storage Class

An identifier's storage class determines the period during which that identifier exists in memory.

C++ provides 5 storage class specifiers, which can be split into two categories:

automatic storage class: **auto, register** (default for local variables, so 'auto' is rarely used.)

static storage class: **extern, static, mutable.**

'register' suggest that the compiler maintain the variable in one of the computer's high- speed hardware registers rather than in memory. E.g. if variables 'counter' or 'total' are often used, keeping them in hardware register will help reduce the tedious work for computer: loading the variables from memory into the registers and storing the results back into memory. (The compiler might ignore register declarations, since there might not be a sufficient number of registers available for the compiler to use. So the command line is a suggestion, but an order.)

Static identifier can be defined either locally or globally (external identifier): For external case, static specifiers has its own special meaning.

Static local variables are still known only in the function in which they are declared, but, unlike automatic variables, static local variables retain their values when the function returns to its caller. **The next time the function is called, the static local variables contain the values they had when the function last completed execution. (See Fig 6.12 p.233 for comparisons)**

For static global variables, see Appendix F, C Legacy Code Topics.

C++ covers basically four program design skill and thinking

1. **procedural-based paradigm: C-language style** 程序導向編成
2. **object-based paradigm:** 物件導向關於封裝的編程**(encapsulation), Class, Struct.**
3. **object-oriented paradigm:** 物件導向關於繼承與動態繫結編程**(dynamic binding)**
4. **generic paradigm:** 泛型編程**, template, function overloading.**

Programming skill

Many program can be divided logically into three phases:

Initialization phase: initializes the program variables.

Processing phase: inputs data values and adjusts program variables accordingly.

Termination phase: calculates and outputs the final results.

Files separating:

Separating files into .cpp and .hpp at the head start. Do not write all function in the main() and then try to separate later, you will find it hard to do so.

Top-down, stepwise refinement skill.

Top: is a statement representing a complete task of a program?

Then begin the refinement process to decompose top statement into pieces of simpler tasks.

Repetition skill:

When specifying loop-continuation condition, try to use '<=', '>=', rather than proper relational operators such as '<', '>'. This would be more intuitive to programmers and save time.

Some programmers use zero-based skill, which requires all variables be initialized to be 0 and start looping from 0.

The header of for, while, do…while, should only relates to one control variable. To make the codes easier be read, do not merge many variables and body statement into the header.

Assignment and logical operator skill:

When coding, try to use the following habits: x=1; 1==x; Thus, if you mistake == as =, you'll be protected by complier err.

Dynamically Allocated Memory

A copy constructor, a destructor and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory.

Function Call Stack

This data structure works behind the scenes and supports the function call/return mechanism and creation, maintenance, destruction of automatic variables of each called function. Easily said, when calling another function, the present local (auto) variable will be reserved and wait for the called function to return value. It's possible that this called function calls another function again, once again, the local variable is reserved and open a stack frame to wait for new function's value return. Once the value is returned, the stack frame and local variable in the last function are destroyed, then the middle function, and finally return values to the original function, like stack (Last-in, First-out).

PBV and PBR with Reference/Pointer Arguments

(C++ provides these three ways to pass arguments to a function)

Pass-by-value: function (int y); main() {…int x} 當呼叫其他函式時，會把原本的變數 copy 並拿到該函式裡使用(此時名稱為 y) ，故原始變數值不會改變，該函式的回傳值也不會改變原始值，除非你令原始變數 x =function(x);

Pass-by-reference with reference arguments: func (int& y){static int z}; 當呼叫其他函式時，直接將原本的變數拿到該函式裡使用(此時名稱為 y)，故原始變數值直接改變，即使該函式為 void 型態也一樣。

Pass-by-reference with pointer arguments: func (int* Ptr); main{…int arr[]; func(arr); }; Ptr 指向目標變數 arr 的記憶體地址(Ptr = &arr[0]) . 當要取值時, 透過 dereferenced operator '*'則可得到所指向位子得值 (*(Ptr+n) = arr[n]). 對*Ptr 做改變也會改變原始對應的 arr[0]值(pass-by-reference).

PS1. Generally speaking, pass-by-reference approach saves time from copying and returning values, but less secure since the original data is exposed to even client functions and can be accessed directly. One way to solve this problem is to use keyword const, e.g. void function (const int& y) {…int x}.

PS2. When returning a reference to a variable declared in the called function, the variable should be declared **static** in that function. Otherwise, it refers to an auto variable that is discarded when the function terminates; such a variable is said to be "undefined," and the program's behavior is unpredictable. References to undefined variables are called **dangling references**.

PS3. If the copy constructor simply copied the pointer in the source object to our target object's pointer, then both objects would point to the same dynamically allocated memory. When the first destructor executes, it deletes the dynamically allocated memory, and the other object's pointer would be undefined, a situation called a **dangling pointer**.

Function Overloading

C++ allows many functions be defined at the same name as long as they have **different signatures**, called function overloading, which is used to create several functions of the same name that perform similar tasks, but on **different data types**. A signature is a combination of a function's name and its parameter types (in order).

PS. Name mangling (name decoration): the compiler encodes each function identifier with the numbers & types of its parameters. For example, with GNU C++ compiler, if the function name is int **CooL2** (**c**har a, **i**nt b, **f**loat **&**c, **d**ouble **&**d), then its mangled name (signature) is: __Z5CooL2ciRfRd, which is of the form __Z + name_size + func_name + type_firstletter with or without R (depends on &).

Note: Constructor overloading is allowed; whereas destructor overloading is not. E.g., when declaring

a var of type vector, you can write: vector<int> vec(10, 5); OR, vector<int> vec(v.begin, v.end);

Function Templates:

Overloaded functions are normally used to perform similar operations that involve different program logic on different data types. If the program logic and operations **are identical** for each data type, we may use function template. Just write 1 single function template, and C++ will automatically generate separate **function template specializations** to each type of call appropriately:

template <class T, class U,…> (or template < typename T, typename U>)

U func(T value1, T value2, T value3, U value4, U value5)

template: keyword; < >: template parameter list;

class or typename: keyword; T, U: formal type parameters, placeholders for fundamental or user-defined types, used as in the above. (function's return type declaration, variables declaration)

Declare inline function

Function call usually spend compiler lots of time but return only one value. One way to solve it is to add "inline" in front of function's return type and put the whole line before the main() to replace function prototype. e.g. inline int getValue (int x) {…}. 經過 inline 宣告的函數會直接在程式碼內展開，省去呼叫所花的時間，但會使程式碼 size 增加很多。**Usually used for functions which has small size but often called.** But the compiler will determine whether to use inline or not based on the benefit-cost of time, so the command line is a suggestion, not an order.

Note: 如果在 class 內宣告並定義一個函式, i.e., class{ void function(…){…} }; 則其效果同 inline, compiler 會自動把 class 內的程式碼嘗試展開.

Default Arguments

You can specify the default value (default argument) for each parameter in a function prototype. When calling that function with two or more default arguments, if an omitted argument is not the rightmost argument in the argument list, then all arguments to the right of that argument also must be omitted.

E.g., int func (int =1, int =2, int =3, int =4); and call the function by typing: func(x, y), which means that 3rd, 4th parameters are default and 1st, 2nd parameters take the values of x and y, respectively.

Function Pointers

A function's name is actually the starting address in memory of the code that performs the function's task. Pointers to functions can be passed to functions, returned from functions, stored in arrays, assigned

to other function pointers and used to call the underlying function. E.g.,

```
void selectionSort (int [], const int, bool (*compare)( int, int ) ){    //function pointer
        if (! (*compare)( work[ smallestOrLargest ], work[ index ] ) )
                smallestOrLargest = index;
}
bool ascending (int, int);        bool descending (int, int);
int main () {
        const int arrSize = 5;      int arr[arrSize] = {2,4,8,6,10};      int order;      cin >> order;
        /**<function's name is its own function pointer>**/
        if (order == 1)  selectionSort (arr, arrSize, ascending);
        else selectionSort (arr, arrSize, descending);
}
```

To sum up this example,

Firstly, for **(*), parenthesis is required** to mean it's a pointer to function. If we write instead **bool *(int, int)**, it means a function with two int inputs **returns a pointer to a bool** data.

Secondly, just as a pointer to a variable is dereferenced to access the value of the variable, **a pointer to a function is dereferenced to execute the function.**

Thirdly, **(*compare), parenthesis is also required**. If not, i.e., ***compare (arg1, arg2)**, the compiler would try to **dereference the value returned from the function call**.

Fourthly, instead of (*compare) (arg1, arg2), you can also write **compare (arg1, arg2)**, which uses the pointer directly as the function name. **We prefer the first method** of calling a function through a pointer, because it explicitly illustrates that compare is a pointer to a function that is dereferenced to call the function.

Arrays & Vectors

type arrayName [(num>0)]: Two ways to initialize array values. Firstly, for loop; secondly, use the brackets, = {val_1, val_2, …}, or = {} to initialize all elements to 0.

PS1: int n [] = {1,2,3}, the complier would automatically create an array of three elements.

PS2: When declaring an array with variable like n [var], then var must >0, and use for loop to initialize the array. (= {} leads to compilation error.)

PS3: as from PS2, usually, array declaration with size must use const int variable.

PS4: int arr[]; is equivalent to int *arr;

(Equivalently, int arr[]; int *ptr; ptr = arr; OR, int arr[]; int *ptr; ptr = &arr[0];),

PS5: array name (without a subscript) is a (constant) pointer to the first element of the array. Thus, we can set ptr refer to the array's first element by writing ptr=arr. or ptr (= (ptr+0) = ptr[0]) = &arr[0]) (which implies *ptr = *(ptr+0) = *(ptr[0]) = *&arr[0] = arr[0]).

Default Pass-by-reference with pointer argument:

type fun_prototype (type [], type) /fun_call (arrayName, arraySize): When passing array into functions, C++ automatically pass it by reference. This makes sense since copying an array usually consumes considerable memory.

PS1: If you put a number in "[]" in the function prototype, compiler will ignore it.

PS2: Although the whole array is passed by reference, individual array elements are passed by value. HENCE, when passing an array to a function, place const to prevent original array from being modified, if necessary.

Multidimensional Arrays: arrayName[m][n] = {{0, 1, …, m-1}, {0, 1, …, n-1}} is an m-by-n matrix.

PS1: Actually, a [x, y] is treated as a[y], because C++ evaluates the expression x, y simply as y.

Arrays of Pointers

*const char * const arr[4]= {"Hearts" , "Diamonds", "Clubs", "Spades"}:* It's an array with size 4 and each entry is a pointer pointing to const char data. This is equivalent to the C-string form of a var declared by **string arr[4]**. It seems like the array has 4 strings as its elements. In fact, **only pointers are stored**, which point to the address of the 1st letter of strings.

PS0: If you declare as char* const arr[4]={…} OR char* arr[4]={…}, compiler warns you **ISO C++11 doesn't allow conversion from string literal to 'char *const',** so *const* char* is required to avoid such waning. 自己認為是因為元素是指標但指向長度不固定的 **string** 並轉成 **C-string** 的話,那指標內容物應宣告為 **const,**否則改變內容物長度或使指標長度有變, 間接影響陣列記憶體儲存空間。

PS1: A string can be transformed into a char array containing '\0', called C-string. **A string can also be regarded as a pointer pointing to the address of its 1st element, and the address value is equal to stringName**.

PS2: The size of each element = length of string + 1, for '\0'.

PS3: arr strings could be placed into a two-dimensional array, but since 2D array must have fixed number of columns for each row when declaring, so considerable memory is wasted.

PS4: Assume const char* arr[4], and string brr[4], both having the same values as original arr[4].

Command: cout << arr;	(same for brr)	Prints: 0x7fff5fbff640
Command: cout << *arr; (== arr[0])	(same for brr)	Prints: Clubs
Command: cout << **arr; (== *arr[0] == arr[0][0])		
(brr requires additional .c_str() to perform this task.)		Prints: C

Note that 'sizeof (*arr)' = 8, which is the size of a pointer that is irrelevant to data type.

But 'sizeof(*brr)' = 24, is exactly the sizeof data type string. And '(*brr).size()' = 5 is exactly the length of the 1st string in brr. In contrast, arr's 1st element has length=5+1 = 6 (because of '\0').

For more details, pls refer to TextBookExample.cpp, line 330.

Dynamic Arrays

Use *new* operator to make object or array created in the ***free store*** (also called the ***heap***)—a region of memory assigned to each program for storing dynamically allocated objects.

E.g., Time * timPtr = new Time(12,45,0); new operator will preserve proper size of memory for Time object, and call constructor to initialize the Time object with its initial value, and finally return a pointer which points to the type specified after new.

If you no longer need the space reserved for the object: delete timePtr;

E.g.2, int *gradesArray = new int[m]; originally, the size of an array created at compile time must be specified using a constant integral expression. With the new operator, you can use any non-negative integral expression that can be evaluated ***at execution time***.

If you no longer need the space reserved for the array: delete [] Array;

PS: Although an array name is a pointer to the array's first element, the following is **not** allowed for dynamically allocated memory: int gradesArray[] = new int[m];

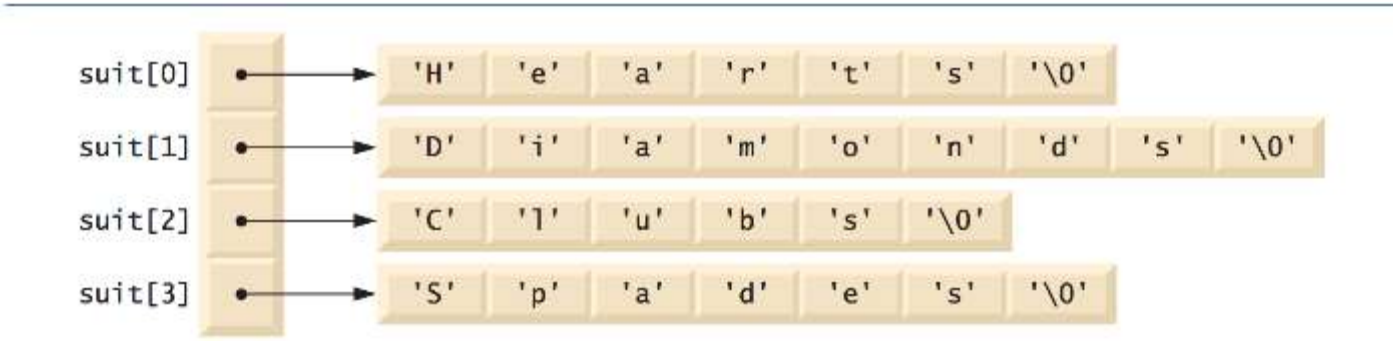


Fig. 8.19 | Graphical representation of the suit array.

Case Study: Array Class

Pointer-based arrays have many problems, including:

- A program can easily "walk off" either end of an array, because C++ does not check whether subscripts fall outside the range of an array (though you can still do this explicitly).
- Arrays of size n must number their elements 0… n – 1; alternate subscript ranges are not allowed.
- An entire array cannot be input or output at once; each array element must be read or written individually (unless the array is a null-terminated C string).
- Two arrays cannot be meaningfully compared with equality or relational operators (because the array names are simply pointers to where the arrays begin in memory and two arrays will always be at different memory locations).
- When an array is passed to a general-purpose function designed to handle arrays of any size, the array's

size must be passed as an additional argument.

- One array cannot be assigned to another with the assignment operator(s) (because array names are const pointers and a constant pointer cannot be used on the left side of an assignment operator).

C++ provides class and operators overloading to finish some of those problems (**c.f. Fig. 11.6 -11.8**, Section 11.9)

Vector

http://mropengate.blogspot.tw/2015/07/cc-vector-stl.html (For more vector operation details.)

Don't use bracket operator '[]'(不做邊界檢查, Segmentation Fault). Use vector.at() instead.

Constant Objects and Constant Member Function

"*[type] func(…)* ***const***": *add the word 'const' after the parameter list of a function declare the function as a constant.* C++ disallows member function calls for constant objects unless the member functions themselves are also declared const.

A **const member function** is a member function that guarantees it will not modify the object (no matter it is constant or not) or call any non-const member functions (as they may modify the object).

A constant member function could be overloaded with a non-constant version. If the object on which the function is invoked is a constant, the compiler uses the constant version. Similarly, if one is non-constant instead, compiler uses the non-constant version.

PS1. Using const with pointers: there are **four** different combinations:

1. Const pointer to const data: const int * const ptr;
2. Non-const pointer to const data: const int * ptr;
3. Const pointer to non-const data: int * const ptr;
4. Non-const pointer to non-const data: int * ptr;

PS2. int * const ptr vs int const * ptr: The first is a const pointer referring to data type int, and the latter is a pointer referring to const int data type.

PS3. **NULL POINTER**:

A pointer variable can only store address except the declaration: ptr = 0; (or ptr = NULL;)

A constant data member must be initialized using member initializer list (See below).

System implicit call to Constructor, Copy Constructor and Destructor

There are 3 functions that will be called implicitly even if the programmer didn't explicitly specify those functions: constructor, copy constructor and destructor.

There are 3 situations that will call a copy constructor of a class:

First, initialize an object with another object of the same class.

Second, pass-by-value: pass an object into a function.

Third, pass-by-value: return an object from a function.

Member Initializer List

Constructor::constructor (type varA, type varB,…)**: data_member1(varA), data_member2(varB), superclass_constructor(arg1, arg2…){}**

From the above code, you can see the member initializer list is after the parameter list with a semi-colon and before the left bracket. Member initializer list is used to initialize data member of an (constant) object. **A constant data member can only be initialized by member initializer list (no other ways)**. You can't initialize it with an assignment in the constructor's body. Also, **calling another constructor of superclass** can only be done by initializer list.

Classes Composition

main(){ Class1 Obj1; Class2 Obj2(…, Class1 Obj1, …) …}: From the above code, the complier will first construct Obj1 of Class1, then enter into the member initializer list of class2 to construct obj1 again by calling "copy constructor"; finally, construct Obj2 of Class2.[1] Similarly, the destructor will first destruct Obj2, then Obj1 (from copy constructor) and finally Obj1 again.

Friend

class A {…; friend type CLASS/Function,…}; A class or a function with prefix **friend** in the definition of class A is a friend of class A. The friend function/class can access all data member and functions of class A no matter the target is public, protected or private. Note that the relationship friend is not symmetric and transitive.

Cascaded member function calls

Another use of the "**this"** pointer is to enable cascaded member-function calls—that is, invoking multiple functions in the same statement. As shown in Fig10.17-19, you can write " t.setTime(20, 20, 20).printStandard();" to execute setTime() and printStandard() in one statement. The reason is that setTime() returns Time&, so as long as t.setTime(20,20,20) finished executing, it returns a

[1] For more detail, see Chapter10.3 and Chapter11 of C++ How to Program

Time reference object and thus, can use the **selection operator(.)**.

Static data member & static member function

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized **outside** the class. A class's static members exist even when no objects of that class exist. To access a public static class member when no objects of the class exist, simply prefix the class name and the binary scope resolution operator (::) to the name of the data member. To access a private or protected static class member when no objects of the class exist, provide a public static member function.

Note: A static const data member of int or enum type can be initialized in its declaration in the class definition. However, all other static data members must be defined at global namespace scope (i.e., outside the body of the class definition) and can be initialized only in those definitions.

Note2: When static is applied to an item at global namespace scope, that item becomes known only in that file. The static class members need to be available to any client code that uses the class, so we declare them **static only in the .h file**.

A static member function can only access static data member, other static member functions and any other functions from outside the class. Static member functions have a class scope and they do not have access to the "**this"** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Note: Declaring a static member function const is a compilation error. The const qualifier indicates that a function cannot modify the contents of the object in which it operates, but static member functions exist and operate independently of any objects of the class.

Operator Overloading

When operators are overloaded as member functions, they must be **non-static**, because they must be called on an object of the class and operate on that object. To use an operator on class objects, that operator must be overloaded—with three exceptions, **the address (&), the comma (,) and the assignment (=) operator, which already have the definition of object version**.

Operators that must me overloaded as member functions if you wants to overload them: **=, (), [], ->.**

Operators that can/cannot be overloaded in C++.

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Fig. 11.1 | Operators that can be overloaded.

Operators that cannot be overloaded			
.	.*	::	?:

Fig. 11.2 | Operators that cannot be overloaded.

Restrictions on Operator Overloading

1. Overloading *cannot* change the **precedence**, the **associativity** and the **number of operands** of an operator.[2]
2. It is **not** possible to **create new operators**; only existing operators can be overloaded.
3. Operator overloading cannot change the meaning of how an operator works on objects of **fundamental types**. E.g., you cannot change the meaning of how + adds two integers.
4. Overloading = and + to allow statements like "object2 = object2 + object1;" doesn't imply that the += operator is also overloaded to allow statements like "object2 += object1". Operators can be overloaded only explicitly; there is **no implicit overloading**. (Unless case_static overloading or conversion operator is provided.)

Operator Functions as Class Members or Global Functions

An operator overloading must be defined explicitly as either a class' member function or a global function. When will an operator overloading be triggered?

Answer: 1. **The LHS operand must be an object of the operator's class**. (自家的object) if the operator function is implemented as an member function. Otherwise, the operator function must be defined as global function to help identify and deal with the LHS operand, which is not an object of the operator's own class.

[2] Page 469, Restriction on Operator Overloading, *C++ How to Program 7e*, Deitel.

E.g., cout << object1; from this statement, << operator **msut be overloaded as a global function** (despite the fact that both LHS operand and << belong to same class Ostream), since we cannot modify the default class Ostream. (If we could, we can overload by declaring: Ostream& operator<<(user-defined class); in the body of class Ostream, which is not allowed because we 're trying to modify the content of a default(fundamental) header file.)

Note: from the above you may figure out another situation that you will prefer to use global functions. That is, the commutative law (交換律). E.g., An operator + is overloaded to deal with the sum of a huge integer and a long int, or in a reverse order.

Overloading Unary Operators and Binary Operators

Case 1(class member function): bool operator!() const;
Then !s means s.operator!();
Case 2(global function): bool operator!(const String &); String s;
Then !s means operator!(s);
Case 3(class member function): bool operator<(const string &) const;
Then y<z means y.operator<(z);
Case 4(global function): bool operator<(const String &, const String &); String y, z;
Then y<z means operator<(y,z);

Static_cast Operator Overloading

1. **Declaration:**
   Assume class A is a user-defined class and 'a' is an class object of A.
   A::operator int() const; static_cast<int>(a); //same as: a.operator int()
   A::operator *otherClass()* const; static_cast< otherClass>(a); // same as: a.operator otherClass()
   Note: **An overloaded cast operator function does not specify a return type**
2. When necessary, the compiler can call cast operators and conversion constructors implicitly to create temporary objects.: for example, if you write the code: cout << a; without overloading <<, but you provide static cast overloading in the body of class A, then compiler will try to find those static_cast to transform object a into the type that can be used for <<. Such a conversion operator must be a **non-static** member function.

Conversion Constructor

Any **single-argument constructor** can be thought of as a conversion constructor. A conversion constructor has **two abilities**: first, it can be used as a constructor like normal case; second, to temporarily

convert some object into other types when necessary (system will implicitly do this action and do it only once.) Such a conversion operator must be a **non-static** member function.

Note: when you do not wish the compiler to automatically implicitly convert objects, use *keyword* ***explicit*** in the front of declaration of conversion constructor. (c.f. Fig. 11.13)

E.g., Array::Array(int); this conversion constructor can convert int into data type Array.

Other Important Operators Overloading

1. Prefix- vs. Postfix operators overloading:
   Date & operator++(); is a prefix increment operator. Take, for example, d1 as an object of the class Date, then ++d1 means d1.operator++();, which returns new d1 value as a reference directly.
   However, Date operator++(int); is a postfix increment operator. When coding as d1++, it means d1.operator+(0), which returns a copy of original object d1 but indeed changes the d1 value in the function body. Notice that 0 here is strictly a 'dummy value' that **enables the compiler to distinguish between the prefix and postfix increment operator functions.**
2. Overloading parenthesis **()** is very powerful:
   E.g., operator () (){…}; operator () (type){…}; operator () (type, type){…} and so on. This illustrates that an object, say s, can call parenthesis with as many arguments as you want like: s(); s(x); s(x,y) and so on.

Inheritance

Public, Protected and Private

Public: 任何人皆可使用(即使無任何關係)。
Protected: 介在 Private 與 Public 權限之間，這些財產不想給外人用，但又想給孫子繼承使用。
Private: 僅該 class 可使用，就算孫子繼承也無法直接呼叫，必須透過老爸留下來的 public or protected 成員來間接使用老爸的私有財產。

PS1. Main()裡面 只能使用 Object class's public members.
PS2. Friend 最大，可存取目標 class 的所有成員。通常是基於效率的考量，以直接存取私用成員而不透過函式呼叫的方式，來省去函式呼叫的負擔。
PS3. **繼承的 class(後代子孫)所新創立的 object，全部也都是 Base class 的 object.**
PS4. **任何 constructor, destructor, assignment operators, friend functions 都不能被繼承。**
PS5. 繼承之後， 若 Derived class 底下有一模一樣的 function，則依照前面是否有 keyword virtual 而分別為: **function redefining(without virtual)/overriding(with virtual)**，redefining 表示新的 function 和

原始 base class 的 function 是分開的，只是用 derived class object 呼叫時，會是呼叫新的；而 overriding 是直接覆蓋掉 base class' function 的內容，通常用在多型(詳見 polymorphism)。

Example

三代的繼承方式有三種： public, protected, private.
Grandfather 有 3 份財產 fa, fb, fc : 打算 fa 公共財產大家都可使用, fb 給 Father, fc private 留給自己.
Father 有 3 冀財產 sa, sb, sc: 打算 sa, sb 給 Son, sc private 留家自己.
Question: fa fb 由 Father 繼承，則 Son 如何繼承？

Answer: 與當初 父親 如何繼承祖父財產的方式有關：
如果當初老爸覺得孫子不錯可以去經營財產 fa, fb, 那老爸當初從祖父繼承財產會是 public.
如果當初老爸覺得孫子相當好不僅可以經營 fa, fb, 甚至杜絕其他人使用財產 fa, fb, (唯一繼承) 那老爸當初從祖父繼承財產會是 protected.
如果當初老爸覺得孫子不好，無法經營財產 fa, fb, 那老爸就會從祖父那兒以 private 方式繼承，僅留給老爸自己使用。

其說明如下:

A. 當初老爸以 public 繼承: Grandfather's protected & public members are also Father's protected & public members, respectively.

Case1➔ Son 如果以 public 繼承，則祖父 public →老爸 public → Son public; 且
祖父 protected → 老爸 protected →Son protected (祖父和老爸的 private 不繼承)

Case2➔ Son 如果以 protected 繼承，則祖父 public →老爸 public → Son protected; 且
祖父 protected → 老爸 protected →Son protected (祖父和老爸的 private 不繼承)

Case3➔ Son 如果以 private 繼承，則祖父 public →老爸 public → Son private; 且
祖父 protected → 老爸 protected →Son private (祖父和老爸的 private 不繼承)

B. 當初老爸以 protected 繼承: Grandfather's protected & public members all becomes Father's protected members.

Case1➔ Son 如果以 public 繼承，則祖父 public & protected →老爸 protected → Son protected;
(祖父和老爸的 private 不繼承)

Case2➔ Son 如果以 protected 繼承，則祖父 public & protected →老爸 protected →Son protected;
(祖父和老爸的 private 不繼承)

Case3➔ Son 如果以 private 繼承，則祖父 public & protected →老爸 protected →Son private;
(祖父和老爸的 private 不繼承)

C. 當初老爸以 private 繼承: Grandfather's all public & protected members all become Father's private members

Only Case➔ Son 無論如何都無法繼承，頂多只能透過老爸遺留下來的 public & protected 成員來間接操作老爸的私有財產，但仍然不是自己名下財產。

注意： 若沒有標示繼承模式， 其預設繼承模式為 private。

注意：friend class/functions 皆不會被繼承。

```
class Grandfather{
        public:       int gfa;
        protected:    int gfb;
        private:      int gfc;
};
 class Father: public Grandfather{
        public:       int fa;
        protected:    int fb;
        private:      int fc;
};
class Son: <any inheritance mode> CB{ ... };
```

說明：

如果 Father 以 public 繼承 Grandfather，則 Grandfather's fa & fb 被繼承下來，並分別成為 Father 的 public member & protected member。(Grandfather's gfc cannot be inherited by class Father)

Son 繼承 Father， 則視 Father 為

```
class Father{
        public:       int gfa, fa;
        protected:    int gfb, fb;
        private:      int fc;
};
```

如果 class Father 的繼承模式改為 protected。 則 Son 視 Father 為

```
class Father{
        public:       int fa;
        protected:    int gfa, gfb, fb;
        private:      int fc;
};
```

如果 Father 的繼承模式改為 private。 則 Son 視 Father 為

```
class Father{
        public:       int fa;
        protected:    int fb;
        private:      int gfa, gfb, fc;
};
```

最後圖示整理:

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Fig. 12.27 | Summary of base-class member accessibility in a derived class.

Inheritance vs Composition

參考 Exercise 12.3 from textbook.

1. 繼承很複雜，雖然程式碼很簡潔，但是樹狀圖堆疊到兩三層之後就很需要思考到底該怎麼繼承才不會破壞被繼承的類別的封裝性
2. 組合與繼承皆可達到 code reuse 的效果，雖說組合相對來講要多寫一點程式碼，但這無傷大雅
3. 可參考網路說法，現在大部分皆提倡 prefer composition over inheritance
4. 要注意多重繼承(circle or diamond problem)在 C++支援，但在 Java 並不支援，所以使用上有限制，在 Python 中亦支援多重繼承，但發生菱形問題時(Diamond problem)，以程式碼最左側被繼承的 Class 為最大優先蓋過任何其他有衝突的 member function or data member definition.

Polymorphism

Basic Concept

A derived-class object "*is a*" base-class object; Derived-class objects can call member functions according to inheritance. Thus, **base-class pointers can point to derived-class objects**, even though the derived-class objects are of different types. (c.f. Fig. 13.15)

P.S. Note both the reverse statements of the above don't hold, i.e., we cannot treat a base-class object as an object of any of its derived class.

Another key point is that **base-class pointer can only access to base-class functions** no matter what class of the pointed object is. This is because compiler only takes the handle's type (i.e., the *pointer* or *reference* type) to determine the invoked functionality. If you wish to make the invoked functionality depending on the type of the object to which handle points, put keyword **virtual** in preceding the prototype of member functions, which is known as ***down-casting skill***.

E.g., virtual void draw() const; # will access derived class' overridden function draw() rather than accessing the base class' draw().

Core Concept and Virtual Functions

The essential concept for polymorphism is demonstrated by the following example: consider a base class Shape has some derived class like Rectangle, Circle, Triangle, etc. We wish to command the function draw in the base class, and then each derived class calls its own draw function to implement the drawing. The operation can even be implemented ***dynamically*** (i.e., at runtime) since the type of object to which base-class pointer points to can be determined at runtime by users.

To enable this behavior, declare **virtual draw()** in base class. Secondly, **override** draw() in each of the derived class to draw appropriate shape.

PS1. Once a function is declared virtual, it *remains virtual all the way down* the inheritance hierarchy from that point, even if that function is not explicitly declared virtual when a derived class overrides it. Even though, you can *put virtual at every level* of the hierarchy to promote program *clarity*.

PS2. Choosing the appropriate function to call at execution time (rather than at compile Time, called **static binding**) is known as **dynamic binding** or **late binding**.

Following, we ***summarize*** what you can do and what you cannot do between base/derived class pointers and objects:

1. Base-pointer pointing to base-object is straightforward.
2. Derived-pointer pointing to derived-object is also straightforward.

3. Base-pointer pointing to derived-object is safe and allowed to invoke base functions. (since derived-object is also a base-object.) But it outputs an error if you try to invoke derived functions. Rules here: base-pointer always invoke base functions unless you put 'virtual' in front of that specific function (it will make the base-pointer to invoke derived function! (down-casting skill)). Note: alternative way to invoke derived function by base-pointer is to make base-pointer point to the derived one, which is DANGEROUS.
4. Derived-pointer pointing to base-object is wrong since derived-pointer will always invoke derived function (when no virtual) but base-object cannot access them.

Besides, following gives the difference between static binding and dynamic binding.

Basis for Comparison	Static Binding	Dynamic Binding
Event Occurrence	Events occur at compile time are "Static Binding".	Events occur at run time are "Dynamic Binding".
Information	All information needed to call a function is known at compile time.	All information need to call a function come to know at run time.
Advantage	Efficiency.	Flexibility.
Time	Fast execution.	Slow execution.
Alternate name	Early Binding.	Late Binding.
Example	Overloaded function call, overloaded operators.	Virtual function in C++, overridden methods in java.

Abstract Class and Pure Virtual Functions

When we think of a class as a type, we assume that programs will create objects of that type. However, there are cases in which it's useful to define classes from which you never intend to instantiate any objects. Such classes are called **abstract classes**. Abstract base classes are too generic to define real objects; we need to be more specific before we can think of instantiating objects. Following are some features of abstract class:

1. Usually, it provides a base class from which other classes can inherit.
2. No object is allowed to be generated in it. (In contrast, classes that can be used to instantiate objects are called **concrete classes**.)
3. <Def> A class is made abstract if it contains at least one **pure virtual function**.
4. <Def> A pure function is a virtual function specified by placing "=0" in its declaration. "=0" is called **pure specifier**. E.g., virtual void draw() const = 0;

*Difference between virtual function and pure virtual function:

- Virtual function has an implementation and gives the derived class the *option* of overriding it.
- Pure virtual function *cannot* have an implementation initially and *requires* the derived class to override it if that class wants to be concrete; otherwise, the class remains abstract.

Q: When to use pure virtual function?

A: They are used when it does not make sense for the base class to have an implementation of a function, but you want all concrete derived classes to implement the function.

Case Study

Please refer to Chapter 13.6 & 13.8 (including Fig. 13.13-13.23 & 13.25), which demonstrates a polymorphic program on different types of employees.

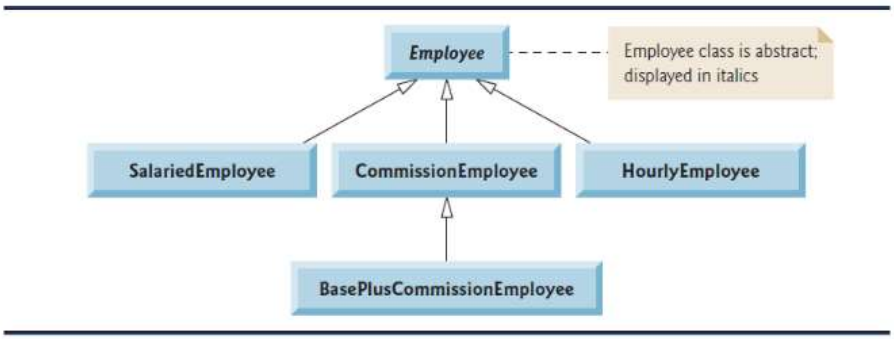


Fig. 13.11 | Employee hierarchy UML class diagram.

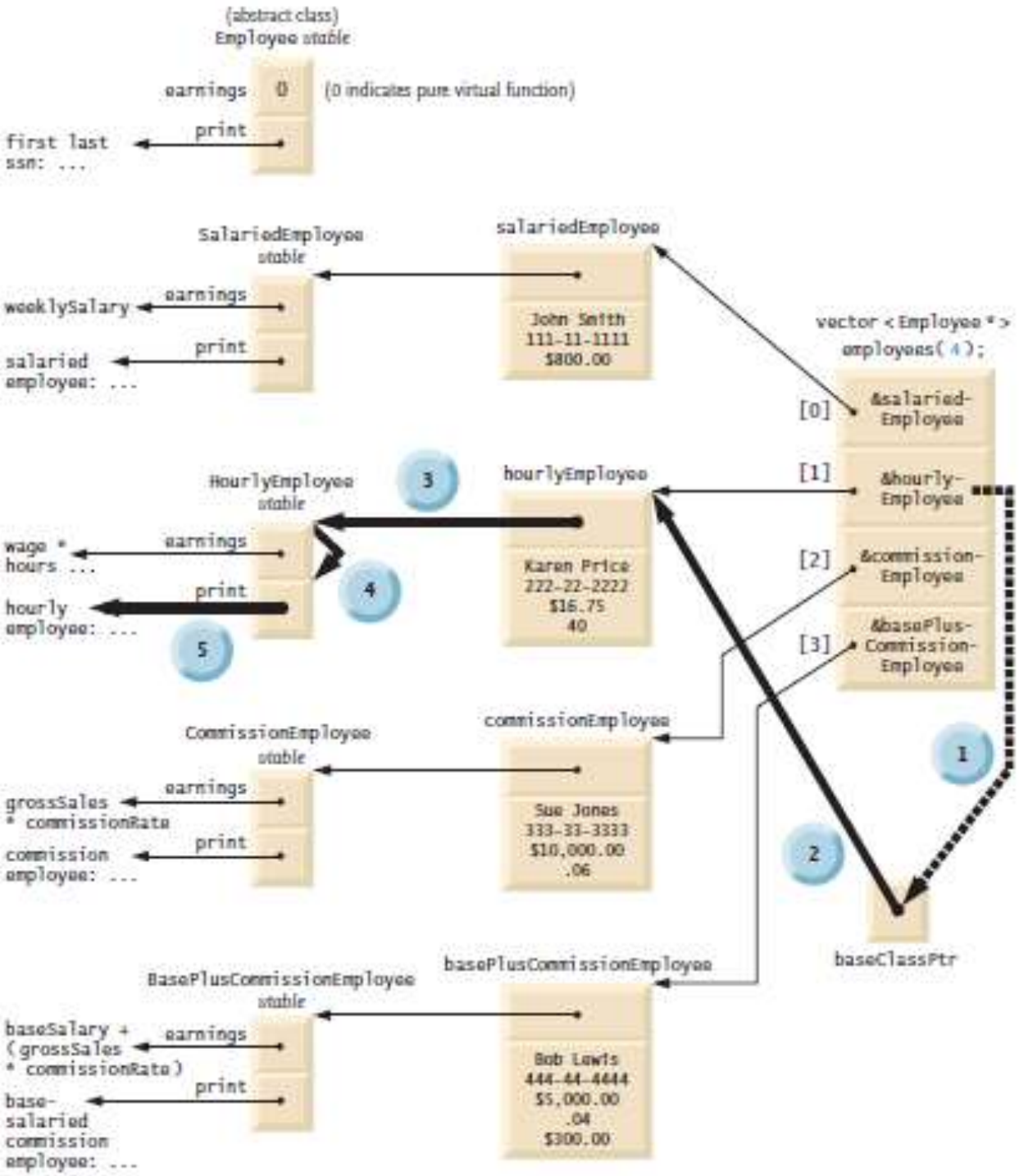
	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* (40 * wage) + ((hours - 40) * wage * 1.5)	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	baseSalary + (commissionRate * grossSales)	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 13.12 | Polymorphic interface for the Employee hierarchy classes.

When C++ compiles a class that has one or more virtual functions, it builds a virtual function table (**vtable**) for that class. An executing program uses the vtable to select the proper function implementation each time a virtual function of that class is called. Any class that has one or more null pointers in its vtable is an abstract class; otherwise, it's a concrete class.

Polymorphism is accomplished through an elegant data structure involving **three levels of pointers**.

1. We've discussed one level—the function pointers in the vtable.
2. Second level pointer: Whenever an object of a class with one or more virtual functions is instantiated, the compiler attaches to the object a pointer to the vtable for that class.
3. The third level of pointers simply contains the handles to the objects that receive the virtual function calls. The handles in this level may also be references. Fig. 13.24 depicts the vector employees that contains Employee pointers.



Flow of Virtual Function Call `baseClassPtr->print()`
When `baseClassPtr` Points to Object `hourlyEmployee`

1. pass `&hourlyEmployee` to `baseClassPtr`
2. get to `hourlyEmployee` object
3. get to `HourlyEmployee` vtable
4. get to `print` pointer in vtable
5. execute `print` for `HourlyEmployee`

Fig. 13.24 | How `virtual` function calls work.

Virtual Destructor

A problem might occur: If a derived-class object with a nonvirtual destructor is destroyed explicitly by applying the *delete* operator to a base-class pointer to the object, the C++ standard specifies that the behavior is undefined.

Sol: create a virtual destructor in the base class, which makes all derived class destructors virtual *even though they don't have the same name as the base class destructor.*

Tips: always provide a virtual destructor when a class has virtual function(s), even if sometimes it is not required for the class.

Note: constructors can never be virtual.

Compiler Error Records

GNU-GCC compiler:
In function `ZN18CommissionEmployeeD2Ev': multiple definition of
'CommissionEmployee::~CommissionEmployee()'
first defined here
原因: linker problem, 可能是 xxx.cpp file 不小心 #include "xyz.cpp": Never include .cpp files in .cpp file, only header files can.

Miscellaneous

A. Type: 宣告變數時，如果皆為 int 型，則相除後的結果仍為整數形式 int
B. **Conditional Operator (? :)**
   The conditional operator is C++'s only ternary operator – it takes three operands. For example, cout << (grade >= 60 ? "Passed" : "Failed"); Or, grade >= 60 ? cout << "Passed" : cout << "Failed";
   Both have the same effect that prints Passed if grade is larger than or equal to 60; and prints Failed if grade is smaller than 60.
C. Type: ***暫時***轉換變數型態可以用 static_cast<type>(expression),
   e.g. double avg=0; int Num=4, total=7;
   avg = static_cast<double>(total) / Num; 則 avg=1.75 (不會被 truncated 掉)
   直得注意的是，total 的型態仍為 int, 並非永久轉換。
D. Type: int vs char: 雖然一般來說字元會被存到 char 裡，但變數宣告 int 時，亦可儲存字元，因為 **int 的空間至少比 char 來得大**。故 int 可以被看作是整數或者字元，端看使用目的。事實上每個字元在電腦裡本身有個等價的 Unicode 數值，請參照 ASCII character set, which is a subset of Unicode. E.g.

cout << 'a' << endl; 會顯示 97, 因為這是 a 的 Unicode.

PS: from int to char: use sprintf(), and from char to int: use the code: int x = chary -'0';

E. **setprecision**: 精確度使用 setprecision(int), 採四捨五入。(#include<iomanip>)

不想用科學記號表示(預設)可以用 cout << setprecision(2) << fixed<< endl;

若想把小數點後的 0 印出來可以用 cout<< showpoint << fixed <<endl;

setw: **A.** 輸出螢幕時,若想靠左或靠右對齊可使用 setw(int)。(#include<iomanip>) 預設是靠右對齊,想靠左對齊就使用 cout << setw(5) << left <<endl; (or << right) **B.** Can restrict user to input limited characters. E.g., char arr[20], brr[30]; cin >> setw(20) >> arr >> brr; This ensures arr stores at most 19 characters from user, and the last space of arr stores '\0'. If user inputs extra chars, then brr stores those values. (**The setw stream manipulator applies only to the next value being input, i.e., only arr here.**)

setfill(char m): 輸出位數時, 固定補字元 n 在前面, 用 setw()控制固定補齊的位數. E.g., cout << setw (2) << setfill ('0') << var_int << endl; 用 left/right 控制補在我邊或右邊。

Note: setfill (…) 會一直持續生效, 可以再次使用 setfill(' ') 把它關掉.

<iostream> cin.getline(char* **var,** size**,** char **dlm='\n') OR**

<string> getline(cin, string **var,** char **dlm='\n')**: 前者存進 char array/後者僅接受存進 string, 讀取輸入直到碰到 delimiter(dlm, 預設為'\n') or EOF or 已經存入 size-1(最後一個位置給'\0'),**不會放入隊列等待**。

cin.get():可直接按順序讀取東西並搭配 while 迴圈,直到碰到 EOF,會放入隊列等待。E.g. cout << "Enter the letter grades." << endl << "Enter the EOF character to end input." << endl; while (grade = cin.get() != EOF) {…} 就可以一次輸入很多很多成績(不用一筆一筆輸入),最後再 EOF 結束。(int EOF; it's a keyword(end-of-file) in <iostream>)

On UNIX/Linux systems, EOF = *<Ctrl> d*, which is called the keystroke (EOF = *<Ctrl> z* for Windows.)

F. **cout, cerr and clog:**

cout 經過緩衝後輸出,預設情況下是顯示器。這是一個被緩衝的輸出,是標準輸出,並且可以重新定向;

cerr 不經過緩衝而直接輸出,一般用於迅速輸出出錯資訊,是標準錯誤,預設情況下被關聯到標準輸出流,但它不被緩衝,也就說錯誤消息可以直接發送到顯示器,而無需等到緩衝區或者新的分行符號時,才被顯示。一般情況下不被重定向。

G. **boolalpha**: 使 bool sxpression 在輸出時均顯示成 true 或 false。(#include<iostream>) (E.g. cout <<boolalpha << (0==1)<<endl;) (Note: 在 C++裡回傳任何非 0 值都代表 true,0 為 false)

H. break & continue statement: break 會直接跳離目前所在 braces { } (e.g. for, switch, while, do…while). Continue 則是直接忽略接下來的所有程式碼直到碰到"}"然後回到 for/while header 迴圈繼續執行下去。

I. **Std. Library.**

<string.h> vs <cstring> vs <string>: 前者是 C 的標準圖書館包含 strlen, strcpy 等等。中間 is basically a header containing a set of functions for dealing with C-style strings (char*), 用以取代前者。後者是 C++ 的包含 std::string, std::wstring 等等,與前兩者功能不同。

<math.h> vs <cmath>: 前者是 C 標準圖書館, 換到不同 compiler 皆可執行，相容性大。後者是 C++ 表準圖書館的檔案，主要定義在 namespace std 下。涵蓋: pow(double, double), swet()

Function	Description	Example
ceil(x)	rounds *x* to the smallest integer not less than *x*	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of *x* (*x* in radians)	cos(0.0) is 1.0
exp(x)	exponential function e^x	exp(1.0) is 2.718282 exp(2.0) is 7.389056
fabs(x)	absolute value of *x*	fabs(5.1) is 5.1 fabs(0.0) is 0.0 fabs(-8.76) is 8.76
floor(x)	rounds *x* to the largest integer not greater than *x*	floor(9.2) is 9.0 floor(-9.8) is -10.0
fmod(x, y)	remainder of *x*/*y* as a floating-point number	fmod(2.6, 1.2) is 0.2
log(x)	natural logarithm of *x* (base *e*)	log(2.718282) is 1.0 log(7.389056) is 2.0
log10(x)	logarithm of *x* (base 10)	log10(10.0) is 1.0 log10(100.0) is 2.0
pow(x, y)	*x* raised to power *y* (x^y)	pow(2, 7) is 128 pow(9, .5) is 3
sin(x)	trigonometric sine of *x* (*x* in radians)	sin(0.0) is 0
sqrt(x)	square root of *x* (where *x* is a nonnegative value)	sqrt(9.0) is 3.0
tan(x)	trigonometric tangent of *x* (*x* in radians)	tan(0.0) is 0

Fig. 6.2 | Math library functions.

J. **\n**：記得以後都放在字串之後寫而不是前面，不然常常會莫名其妙掛掉。

K. ++a vs a++ (prefix vs postfix): 僅當前那一行程式碼會受影響，prefix 先把 a 加一再存起來，再執行該行。Postfix 先執行完該行程式碼，再把 a 加上一才存起來。如下兩張圖。

Note: ++(a+1) 是不對的寫法。

```
11    c = 5; // assign 5 to c
12    cout << c << endl; // print 5
13    cout << c++ << endl; // print 5 then postincrement
14    cout << c << endl; // print 6
15
16    cout << endl; // skip a line
17
18    // demonstrate preincrement
19    c = 5; // assign 5 to c
20    cout << c << endl; // print 5
21    cout << ++c << endl; // preincrement then print 6
22    cout << c << endl; // print 6
23 } // end main
```

```
5
5
6

5
6
6
```

Operator	Called	Sample expression	Explanation
++	preincrement	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postincrement	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	predecrement	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	postdecrement	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Fig. 4.18 | Increment and decrement operators.

L. Operator precedence and Promotion hierarchy for data types:

Operators	Associativity	Type
::	left to right	scope resolution
() []	left to right	highest
++ -- static_cast< *type* >(*operand*)	left to right	unary (postfix)
++ -- + - !	right to left	unary (prefix)
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 7.2 | Operator precedence and associativity.

Data types	
long double	
double	
float	
unsigned long int	(synonymous with unsigned long)
long int	(synonymous with long)
unsigned int	(synonymous with unsigned)
int	
unsigned short int	(synonymous with unsigned short)
short int	(synonymous with short)
unsigned char	
char	
bool	

Fig. 6.6 | Promotion hierarchy for fundamental data types.

M. For: 如果參數在 for loop 就定義時(e.g. for(int x=1; x<=10; x++)) 則該變數在 for 迴圈外就沒有定義了，有效範圍僅在迴圈內。Notation: for (initialization; loopContinuationCondition; increment). 如果初始值在 for 之前已經定義則可以直接分號省略，如果 increment 已經在 for body 內有寫出來則 for header 的 increment expression 亦可直接省略，☐還是直接在 for header 寫就好，盡量不要在內部動到該控制變數。事實上，for header 的三個條件全部都可以打分號直接省略。

N. While: while(++counter<=10)… 這個寫法也可以，但在 debug, modify 時不容易找。

O. **sizeof():returns the size of the input target in bytes.**

It's regarded as an operator rather than a function. Reasons:

1. The input expression of sizeof isn't evaluated at runtime (sizeof a++ does not modify a).
2. sizeof() has very high precedence. E.g., sizeof a+b is not equal to sizeof (a+b).
3. The expression which is the operand of sizeof can have any type except void, or function types.
4. ***sizeof is a compile-time unary operator, not an execution-time operator, using sizeof does not negatively impact execution performance.***

PS1. If input is type name (int, string…) then the parenthesis cannot be ignored; otherwise, we can just write, for example, sizeof arrayName;

PS2. Some examples: sizeof (arr) = size of arr in bytes.

sizeof (*ptr) = size of the type of arr's 1st element, i.e. arr[0], where *ptr = arr;

sizeof (ptr) = 8, where *ptr = arr , returns the size of pointer in bytes (也就是指標本身配置記憶體的大小**, 跟系統有關與任何型態無關), not the size of arr in bytes.**

P. **enum**: enum Type_name {identifier1, identifier2, …};

The keyword enum declare a user-defined type Type_name, and variables under this type can only be assigned values in enumeration, which is a set of integer constants represented by identifiers, starting from 0 unless specified otherwise. E.g., enum Month {JAN=1, FAB, MAR}, where JAN represents 1, FAB represents value 2, and a variable of type Month can only be assigned as 1,2,3 or JAN, FAB, MAR.

Q. **Compare rand() and srand():** (#include<cstdlib>)

Function rand actually generates **pseudorandom numbers.** Repeatedly calling rand produces a sequence of numbers that appear to be random. However, the sequence repeats itself each time the program executes. Thus, rand is used to check the program and debug.

srand (int seed) uses the parameter seed to seeds the rand() function to produce a different sequence of random numbers for each execution. To randomize without having to enter a seed each time, you may use "srand (time (0))." (#include<ctime>)

R. **Unary Scope Resolution Operator (::)**

When a local variable has the same name as some global variable, "::" can help **access the global** variable simply by adding it in front of the desired variable name. E.g., cout << ::number << endl;

S. ***square brackets ([]) operator can be used to catch the characters in a string (i.e., str[*num*])***

str.**at(num)**: catch the position num in str. (the 1st position in str corresponds to num=0)

str**.substr(**int A**,** int B**):** A substring with length B, starting from position A (the first char is at position 0), of a string str.

strlen(str**):** starting from the first non-null character and terminating at the last non-null character, it returns the length of string str. (#characters)

str**.length():** returns the length of string str in bytes. (#bytes)

str.**size():** equivalent to str.length!

vector.**size():** returns the number of elements in vector. (not the capacity)

strcpy(char1* , char2* **):** Copies the C string pointed by source(char2*) into the array pointed by destination (char1*).

strncpy(char1*, char2*, **size_t** num**):** same as strcpy(char1*, char2*) but with restricted copy of length =num.

str**.c_str():** Returns a pointer to an array that **contains a null-terminated sequence of characters** (i.e., a C-string) 也就是說把 str 轉成 C-string (= 存在 array 裡的一連串 char 且最後附上 null-terminated character '\0') 可搭配 strcpy 使用

T. **Predicate Functions:**

http://www.cplusplus.com/reference/cctype/ (<cctype> contains many useful type checking tool.)

isalnum(int n**):** returns true if a char n is either a decimal digit or an uppercase/lowercase letter; returns false otherwise. (notice that n is a char to be checked, casted to an int, or EOF)

isdigit(int n**):** returns true if a character n is a decimal digit; returns false otherwise.

isalpha(int n**):** returns true if a char n is an uppercase/lowercase letter; returns false otherwise.

stoi(const string& __str, **size_t*** __idx = 0, **int** __base = 10**):** string to int data type, with base=進位制.

atoi(const char * str**):** return C-string str into int type.

isEmpty/isFull/isAm/isPM

U. **goto statement**: 按照歷史發展 1960 年代大家都在用，但發現效率低落而且使城市更複雜更亂，所以盡量避免使用比較好(1970 年代大家都盡量不使用了)，改由以下三種操作方法取代。

C++只有 3 種 control structures: 作者稱之為"Control statements"分別是:

1. Sequence statement: 就是一般依序執行的程式碼。
2. Selection statement: 只有 3 種: if, if…else…, switch…case….
3. Repetition statement: 只有 3 種: while, for, do…while.

V.

C++ Keywords				
Keywords common to the C and C++ programming languages				
auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			
C++-only keywords				
and	and_eq	asm	bitand	bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	false
friend	inline	mutable	namespace	new
not	not_eq	operator	or	or_eq
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

Fig. 4.3 | C++ keywords.

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they're evaluated left to right.
*, /, %	Multiplication, Division, Modulus	Evaluated second. If there are several, they're evaluated left to right.
+ -	Addition Subtraction	Evaluated last. If there are several, they're evaluated left to right.

Fig. 2.10 | Precedence of arithmetic operators.

Standard algebraic equality or relational operator	C++ equality or relational operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Fig. 2.12 | Equality and relational operators.

Operators	Associativity	Type
`::`	left to right	scope resolution
`()` `[]`	left to right	highest
`++` `--` `static_cast< type >( operand )`	left to right	unary (postfix)
`++` `--` `+` `-` `!`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 7.2 | Operator precedence and associativity.

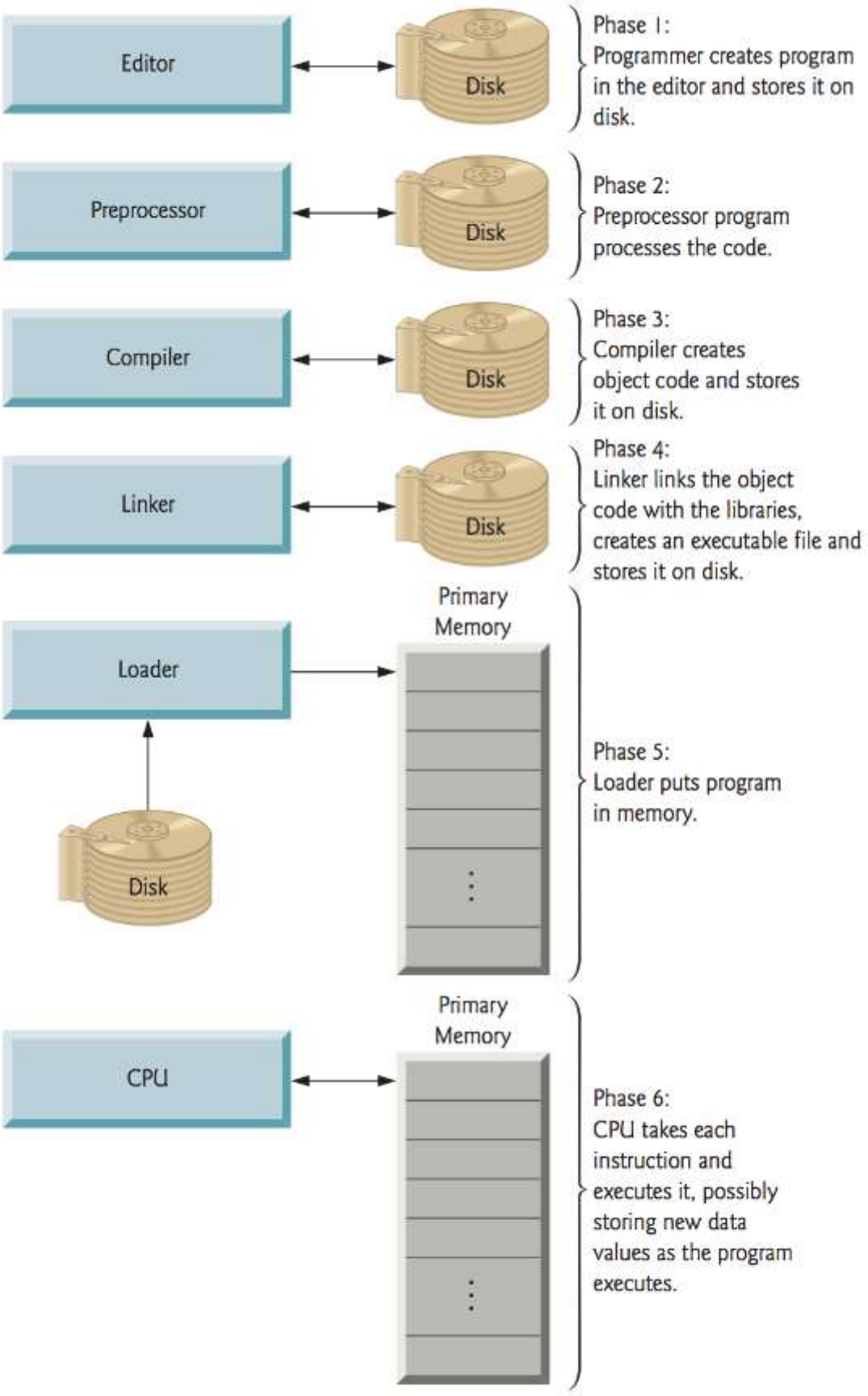


Fig. 1.1 | Typical C++ environment.