C++ Note

Yinghua Zhang

September 17, 2013

Contents

Chapter 1

Types and Declarations

1.1 Booleans

If integer is converted to bool, zero is converted to false, and a nonzero value is converted to true.

1.2 Characters

A plain char is chosen as signed or unsigned is implementation-defined.

1.3 Integer

- A plain integer is always signed.
- A literal starting with zero followed by a 'x' is a hexadecimal; a literal starting by zero but not followed by 'x' is a octal number.
- The suffix 'U' can be used to write explicitly unsigned literals. Similarly, the suffix 'L' can be used to write explicitly long literals. For example, '3U', '3L'.

1.4 Float Point

- By default, a float-point literal is of type 'double'.
- Floating point literal can be defined as 'float' by using the suffix 'f' or 'F', such as '2.0f', '2.9e-3f'. Floating point literal can be defined as 'long double' by using the suffix 'l' or 'L', such as '3.1415926L'.

1.5 Size

- $sizeof(N) \equiv sizeof(signedN) \equiv sizeof(unsignedN)$.
- A char has at least 8 bits, an int at least 16 bits, and a long at least 32 bits. An integer is typically a 4-bites word.

- Implementation defined aspects of fundamental types can be found by functions in <*limits*>.

1.6 void

'Void' is either used to specify that a function does not return a value, or as a base type of pointers to objects of unknown type.

1.7 Enumerations

$enum\ keyword\{ASM,\ AUTO,\ BREAk\};$	$ASM == 0;\ AUTO == 1;$ $BREAK == 2;$
$enum\ e1\{dark,\ light\};$	$range\ 0:1$
$enum\ e2\{a=3,\ b=9\};$	$range\ 0:15$
$enum\ e3\{min=10,\ max=1000\};$	$range\ -1024:1023$
$enum\ flag\{x=1,\ y=2,\ z=4,\ e=8\};$	$range\ 0:15$
$flag\ f1=5;$	*error: 5 is not of type flag*
$flag\ f2=flag(5);$	*ok: flag(5) is of type flag and within the flag range*
$flag\ f3=flag(z\|e);$	*ok: flag(12) is of type flag and within the flag range*
$flag\ f4=flag(99);$	*undefined, 99 is not within the range of flag*

1.8 Declarations

- There must be exactly one definition for each named entity in a C++ program. However, there can be many declarations. All declarations of an entity must agree on the type of the entity referred to.
- Names starting with a underscore are reserved for special facilities in implementation and the run-time environment, so such names should not be used in application programs.
- Except for function and name space declaration, a declaration is terminated by a semicolon.

1.8.1 Scope

- For a name declared in a function, that scope extends from the point of declaration to the end of the block in which its declaration occurs. A block is a section of code delimited by a '{ }' pair.
- A name is called global if it is defined outside any function, class, or name space. The scope of a global name extends from the point of declaration to the end of the file in which its declaration occurs. Object declared in global or name space scope and statics declared in functions or classes are created and initialized once and 'live' until the program terminates. Such objects are called static objects. Array elements and non-static structure

or class members have their lifetime determined by the object of which they are part.

- A declaration of a name in a block can hide a declaration in an enclosing block or a global name. That is, a name can be redefined to refer to a different entity within a block.
- A hidden global name can be referred to using the scope resolution operator '::'.

1.8.2 Initialization

- If an initializer is specified for an object, that initializer determines the initial value of an object.
- If no initializer is specified, a global, name space, or local static object is initialized to zero of the appropriate type.
- Local variables (or automatic objects) and objects created on the free store (or dynamic objects or heap objects) are not initialized by default.
- Members of arrays and structures are default initialized or not depending on whether the array or structure is static.

Chapter 2

Pointers, Arrays, and Structures

2.1 arrays

- The number of elements of the array, the array bound, must be a constant expression.
- When an array is declared without specific size, but with an initializer list, the size is determined by counting the number of the elements of the inilializer list.
- If a size is explicitly specified, it is an error to give surplus elements in an initializer list.
- If the initializer supplies too few elements, '0' is assumed for the remaining array elements.
- There is no array assignment: array cannot be initialized after declaration.
- An array of characters can be conveniently initialized by a string literal.
- A string literal is a character sequence enclosed within double quotes. It contains one more character than it appears to have: 0.
- string literal is a constant: a string literal can be assigned to a $char*$, but this pointer can not be used to change an element of the string literal.
- A string literal is statically allocated, so that it is safe to return one from a function. The memory holding the string literal will not go away after the function return.
- Long strings can be broken by white space to make the program text neater.
- A string with the prefix L, such as $L"angst"$ is a string of wide characters, its type is $const\ wchar\_t[\ ]$.

$char*\ p = "Plato"; p[4] = 'e';$ *error*;
$char\ p[\ ] = "zero"; p[0] = 'R';$ *OK, copy constant to an array*

2.2 Pointer to Arrays

An array name can be used to assign a pointer, but a pointer can not be used to assign an array. The implicit conversion of an array argument to a pointer means hat the size of the array is lost to the called function.

2.3 Constants

- Constants can not be assigned, so they must be initialized.
- Constants can be array bounds and case labels.
- The address of a constant cannot be assigned to a non-constant pointer.

$$
\begin{array}{ll}
const\ char * \ p = s; & pointer\ to\ constant\ char \\
char\ const * \ p = s; & pointer\ to\ constant\ char \\
char\ \ * const\ p = s; & constant\ pointer
\end{array}
$$

2.4 References

- Reference must be initialized. A reference cannot be changed after initialization, it always refers to the object it was initialized to denote.
- We cannot define an array of reference, they are not objects.
- References can be used to define functions that can be used on both the left-hand and right-hand sides of an assignment.
- The main use of references is for specifying arguments and return values for functions in general.

$$
\begin{array}{ll}
int\ i = 1;\ int\&\ r = i; & r\ and\ i\ now\ refer\ to\ the\ same\ int \\
extern\ int\&\ r2 & r2\ is\ initialized\ elsewhere
\end{array}
$$

2.5 Pointer to void

- A pointer to any type of object can be assigned to a variable of type $void *$.
- To use a $void *$, it must be explicitly converted to a pointer to a specific type.
- The primary use for $void *$ is for passing pointers to functions that are not allowed to make assumptions about the type of the object and for returning untyped objects from functions.

2.6 Structures

- A semicolon is needed after the curly brace of structure declaration.
- The notation used to initializing arrays can also be used for initializing variables of structure types.
- Structure can be assigned.
- == and ! = are not defined.
- The size of an object of a structure is not necessarily the sum of sizes of its members.
- The name of a type becomes available for use immediately after it has been encountered and not just after the complete declaration has been seen. used for linked list
- It is not possible to declare new objects of a structure type until the complete declaration has been seen. Because the compiler is not able to determine the size.
- To allow two or more structure types to refer ot each other, we can declare a name to be the name of a structure type.
- It is possible to declare a *struct*, *class*, *union* or *enum* by the same name of a non-structure, for example, a function. The structure must be referred to with the prefix *struct*.
- Two structures are different types even when they have the same members.
- A structure contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have varied data types. Each data object in a structure is a member or field. A union is an object similar to a structure except that all of its members start at the same location in memory. A union variable can represent the value of only one of its members at a time.

Chapter 3

Expressions and Statements

3.1 Expressions

comma (sequencing) $b = (a = 2, a + 1)$ assign 3 to b. !0 is he value *true*, ~0 is the bit pattern all-ones. $y = + + x$ is equivalent to $y = (x+ = 1)$, and $y = x + +$ is equivalent to $y = (t = x, x+ = 1, t)$. $while(*p + + = *q + +)$; copy a zero-terminated string.

3.2 Free Store

To deallocate space allocated by *new*, *delete* and *delete*[] must be able to determine the size of the object allocated. This implies that the object allocated by the standard implementation of *new* will occupy slightly more space than a static object. We can specify what new should do upon memory exhaustion. When *new* fails, it first calls a function specified by a call to *set_new_handler*() (Use this to set a callback function to deal with the case when *new* fails).

malloc is not typesafe in any meaningful way. In C++ you are required to cast the return from *void**. *malloc* returns NULL if allocation fails. *new* will throw *std::bad_alloc*

3.3 Casting

- *static_cast* convert from one pointer to another in the same class hierarchy, an integral type to enumeration, or a floating point to an integral type.
- *reinterpret_cast* handles conversions between unrelated types such as an integer to a pointer or a pointer to an unrelated pointer type.
- Some *static_case* are portable, but few *reinterpret_cast* are. hardly any guarantees are made for *reinterpret_cast*, but generally it produce a value of new type that has the same bits pattern as its argument.
- *dynamic_cast* is a form of run-time checked conversion.
- *const_cast* is used for removing *const* and *volatile* qualifiers.

- From C, C++ inherited the notation $(T)e$, which performs any conversion that can be expressed as a combination of *static_cast*, *reinterpret_cast*, and *const_cast* to make a value of type T from the expression e.

$int*\ p = static\_cast<int*>(malloc(100));$
$IO\_device*\ dl = reinterpret\_cast<IO\_device*>(0xff00);$

3.4 Constructors

- For a build in type T, $T(e)$ is equivalent to $(T)e$.
- Conversions to pointer types cannot be expressed directly using the $T(e)$ notation.
- The constructor notation $T()$ is used to express the default value of type T. $int\ j = int()$.

3.5 Statements

- Unless a variable is *static*, its initializer is executed whenever the thread of control passes through the declaration. There is rarely a reason to introduce a variable before there is a value for it to hold. The most common reason to declare a variable without an initializer is that it requires a statement to initialize it. Examples are input variables and arrays.
- We can declare a variable in conditions, like selection condition or iteration condition. $if(double\ d = prim(true))\{...\}$: d is declared and initialized and the value of d after initialization is tested as the value of the condition. The scope of d extends from its point of declaration to the end of the statement that the condition controls. For example, hand there been an else-branch to the if statement, d would be in scope on both branches. Declaring d before the condition opens the scope for the use of d before its initialization or after its intended useful life.
- User can explicitly exit a loop by a *break*, *return*, *goto*, *throw*.
- The scope of a *goto* label is the function it is in.
- A *break* statement exits its nearest enclosing switch-statement or loop.
- A *continue* statement is equivalent to going to the very end of a loop, in in this case.

Chapter 4

Functions

4.1 Function Declaration

- A function definition is a function declaration in which the function body of the function is presented.
- The argument names are not part of the argument types and need not to be identical.
- A local variable is initialized when the thread execution reaches its definition. A static variable will be initialized only the first time the thread of execution reaches its definition. A static variable provides a function with *a memory*.

4.2 Passing Arguments

- The absence of *const* in the declaration of a reference argument is taken as a statement of intent to modify the variable.
- Similarly, declaring a pointer argument *const* tells readers that the value of an object pointed to by that argument is not changed by the function.
- If an array is used as a function argument, a pointer to its initial element is passed.

4.3 Value Return

A pointer to a local variable or a reference to local variable should never be returned.

Like the semantics of argument passing, the semantics of function value return are identical to the semantics of initialization.

A return statement is considered to initialize an unnamed variable of the returned type.

4.4 Overload Function Names

- Using the same name for operations on different types is called *verloading*.
- Finding the right version to call from a set of overloaded functions is done by looking for a best match between the type of the argument expression and the parameters (formal arguments) of the functions.
- Overloading is not considered in overload resolution.
- Functions declared in different rednon-namespace scopes do not overload.
- Exact match: match using no or only trivial conversions, such as from T to $const\ T$.
- Match using promotions: such as from $bool$ to int, from $char$ to int, from $float$ to $double$.
- Match using standard conversion: such as from $double$ to int, from int to $double$, from $BASE*$ to $Derived*$, from $T*$ to $void*$, from int to $unsignedint$.
- Match using user defined conversions.
- Match using ellipsis ...

The effect of default argument can alternatively be achieved by overloading: They are equivalent.

$$void\ print(int\ value,\ int\ base = 10);$$
$$inline\ void\ print(int,\ value)\ \{print(value,\ 10)\}$$

Default arguments may be provided for trailing arguments only.

4.5 Unspecific Number of Arguments

For example:

$$int\ printf(const\ char * ...);$$

when use, either

$$printf("Hello, world!\backslash n");$$

or

$$printf("My\ name\ is\ \%s\ \%s\backslash n",\ first\_name,\ second\_name);$$

the number of argument can not be determined.

4.6 Pointer to a Function

A example can be

$$void\ error(string\ s)\ \{/ * ... * /\};$$
$$void(* efct)(string); efct = \&error; efct("HelloWorld!");$$

The last sentence shows that dereferencing a pointer to a function is not necessarily to use $*$. Similarly, using & to get the address of a function is optional, that is

$$efct = error;$$

is also OK.

In pointer assignment, the complete function type must match exactly, including both the argument list and return value. There is no implicit conversion of argument or return types when pointers to functions are assigned or initialized.

Pointers to functions can be used to provide a simple form of polymorphic routines.

4.7 Macros

- Macro names cannot be overloaded, and the macro preprocessor cannot handle recursive calls.
- The directive $undef\ X$ ensures that no macro called X is defined - whether or not one was before the directive.
- Using Macros for conditional compilation is almost impossible to avoid. $\#ifdef...\#endif$.
- Common and unavoidable headers contain many dangerous and unnecessary macros.

Chapter 5

Namespaces and Exceptions

5.1 Namespaces

- A namespace is a mechanism to expressing logical grouping.
- A namespace is a scope.
- Ordinary local scopes, global scopes, and classes are namespaces.
- A name form another namespace can be used when qualified by the name of its namespace, using *namespace_name* :: *element_name*, the element name is called qualified name.
- When a name is frequently used outside its namespace, $using-declaration$ can be used to introduce a local synonym in current scope: *using nn* :: *en*;. After $using - declaration$, *en* can be used without *nn::* any more. Place $using - declaration$ in the declaration of a namespace enables all the element functions in this namespace freely use the declared elements from other namespeces. Place *using namespace nn* (*using_directives*)in the declaration of a namespce enables all the element functions in this namespaces use all the elements of namespace *nn*.
- Use namespace to avoid name clash of functions in different header files.
- Use a unnamed namespace to preserve locality of code only, rather than to present an interface to users. There is no dangerous for a unnamed namespace to have name clash in a global scope. unnamed namespaces in different translation units are different.
- If a function isn't found in the context of its use, we look in the namespaces of its arguments. In this way, explicit qualification can be avoid.
- We can define a short alias to represent a very long name of namespace: $namespace\ ATT = American\_Telephone\_and\_Telegraph$.
- $using-declaration$ avoids polluting the global namespace, nonlocal *using-directives* primarily a transition tool.
- Overloading works across namespaces.

- A namespace is open; that is, you can add names to it from several namespace declarations.
- We can $re-open$ a namespace to define new functions.

5.2 Exceptions

exception mechanism:

- $try-block$: $try\,\{...\}$
- $throw$ sentence in the block: $throw$ any data type containing error information
- $exceptionhandler$: $catch(...)\,\{...\}$

Chapter 6

Source Files and Programs

6.1 external linkage and internal linkage

externallinkage: a name that can be used in translation units different from the one in which it is was defined. *internallinkage*: a name that can be referred to only in the translation unit in which it is defined.

- Use the key word *extern* to indicate that a declaration is just a declaration but not a definition.
- A variable defined without an initializer in the global or a namespace scope is initialized by default. This is not the case for local variables and objects created on the free store.
- An object must be defined exactly once in a program. It might be declared multiple times, but the types must agree exactly.
- An *inline* function must be defined - by identical definitions - in every translation unit in which it is used.
- By default, *const*s and *typedef*s have internal linkage. A *const* can be externally linked by explicit declaration.
- Global variables that are local to a single compilation unit are a common source of confusion and are best avoided. To ensure consistency, programmers should usually place global *const* and *inlines* in header files only.
- An unnamed namespace can be used to make names local to a compilation unit.
- Do NOT use *static* except inside functions and classes.

6.2 header files

Use angle brackets < and > to include the standard library headers. A header file should never contain:

- Ordinary function definitions
- Data definitions and Aggregate definitions
- Unnamed namespace
- Exported template definitions.

The idea of the multiple-header approach is to represent each logical module as a consistent, self-contained unit. To allow redundant *#inlcude*, use *includeguards* in headers.

$$
\begin{array}{l}
\#ifndef\ XXX\_H \\
\#define\ XXX\_H \\
... \\
\#endif\ //XXX\_H
\end{array}
$$

6.3 One Definition Rule

The keyword *export* means *accessible from another translation unit.*

6.4 Linkage convention

When linking with non-C++ code, on can specify a linkage convention to be used in an *extern* declaration.

- There is a mechanism to specify linkage to a group of declarations.

  $extern"C"\{...\}$ (a *linkage block*), this construct can be used to enclose a complete C header to make a header suitable for $C++$. This technique is commonly used to produce a $C++$ header from a C header.

- Conditional compilation can be used to create a common C and $C++$ header.

$$
\begin{array}{l}
\#ifdef\ \_\_cplusplus \\
extern\ "C"\{ \\
\#endif \\
... \\
\#ifdef\ \_\_cplusplus \\
\} \\
\#endif
\end{array}
$$

Different from a namespace, a variable in it is declared, defined and initialized by default, but not considered as a local variable. To declare but not define a variable, you must apply the keyword *extern* directly in the declaration.

6.5 Programs

In principle, a variable defined outside any function (that is, global, namespace, and class static variables) is initialized before $main()$ is invoked. Such nonlocal

variables in one translation unit are initialized in their definition order. If such a variable has no explicit initializer, it is by default initialized in their definition order. If such a variable has no explicit initializer, it is by default initialized to the default for its type. the default initializer value for built-in types and enumerations is 0.

It is generally best to minimize the use of global variables and in particular to limit the use of global variables requiring complicated initialization. a function returning a reference is a good alternative to a global variable.

Terminate a program by

- returning from *main*()
- calling *exit*()
- calling *abort*()
- throwing an uncaught exception

When use them

- If a program is terminated by *exit*(), the destructors for constructed static objects are called. Calling *exit*() in a destructor may cause an infinite recursion. If by *abort*(), they are not.
- Calling *exit*() means that the local variables of the calling function and its callers will not have their destructors invoked. Throwing an exception and catching it ensures that local objects are properly destroyed.
- Calling *exit*() terminates the program without giving the caller a chance to deal with the problem. *atexit*() offers the possibility to have code executed at program terminated. The destructor of a constructed statically allocated object created before a call of *atexit*(f) will be invoked before f is invoked.

Chapter 7

Classes

Constructors obey the same overloading rules as do other functions.

7.1 Static Members

- We can get the convenience of default arguments without the encumbrance of a publicly accessible global variable. We can use a variable that is part of the class, yet NOT a part of an object of that class, a *static* member.

- Similarly, a function that needs access to members of a class, yet doesn't need to be invoked for a particular object, is called a *static* function.

- Static member can be referred to like any other member. In addition, a static member can be referred to without mentioning an object. Instead, its name is qualified by the name of its class.

- Static members - both function and data members - must be defined somewhere. The keyword *static* is not repeated in the definition of a static member.

7.2 Copy Object

A class object can be initialized with a copy of an object of its class; similarly, class objects can by default be copied by assignment. By default, the copy of a class is a copy of each member. Memberwise copy is usually the wrong semantics for copying objects containing resources managed by a constructor/destructor pair. Since memberwise copy may just assign an address to a pointer, but not actually allocate memory for it, but for automatic variables, the destructor will be implemented for each objects, which could lead to multiple delete for a same piece of memory.

Such anomalies can be avoided by defining *copy constructor* and *copy as-*

signment operator.

```
T :: T(const T& t){
        int size;
        int * p;
        T(const T&); //copy constructor
        T& operator = (const T&); //copy assignment
};
T :: T(constT&t){
        p = newint[size = t.size];
        for(int i = 0; i < size; i++) {p[i] = t.p[i]; }
}
T&T :: operator = (const T& t){
        if(this! = & t){
                delete [ ] p;
                p = new p[size = t.size];
                for(int i = 0; i < size; i++) {p[i] = t.p[i]; }
        }
        return this;
}
```

I use a reference argument for the copy constructor, because any call would involved an infinite recursion without the &: argument by value will also call copy constructor to create a temporary input

7.3 Constant Member Function

example: *int f() const {return d;};*

- The *const* after the argument list in the function declaration indicates that the fuction *f* does not modify the state of an object.
- When a *const* is defined outside its class, the *const* suffix is required. (this is different from the static member function).
- A *const* member can be invoked by both *const* and $non-const$ objects, whereas a $non-const$ member can only be invoked by $non-const$ objects.
- Use *const_cast* operator to obtain a pointer of type *T* from *this*: in a *const* member function of a non-*const* object, the type of *this* is *const T**. In a *const* object, *const_cast* may not work.

Some times the *const* member function is only $logical-constness$

- Declare a member variable as *mutable*, then this variable can be freely changed in a *const* member function. (Even in *const* objects, *mutable* member variable can be changed in *const* function.
- Another technique to change variable in *const* member function is: to place the changing data in a separate object and access it via a member pointer. In this way, we do not change the member pointer in *const* function, but we change the member of the separate object via the member pointer.

7.4 Self-Reference

example:

$$
\begin{aligned}
&T\&\ T :: fun(int\ n)\{\\
&\quad ...\\
&\quad return\ *this;\\
&\}
\end{aligned}
$$

It is often useful to return a reference to the updated object so that the operations can be chained. Use *this* pointer to get the reference. *This* is not a ordinary pointer, it is not possile to take the address of *this* or to assign to *this*. In a non-*const* member function of class T, the type of *this* is $T*$. In a *const* member function of class T, the type of *this* is *const* $T*$.

7.5 structure

By definition, a *struct* is a class in which members are *by default* public. However, the *private* access specifier can also be used to declare *private* members in a *struct*.

7.6 Helper Functions

A class has a number of unctions associated with it that need not be defined in the class itself because they don't need direct access to the representation. They are *helper functions*. Their declarations were imply placed in the same file as the declaration of class. In addition to using a specific header, we can make the association explicit by enclosing the class and its helper functions in a namespace.

7.7 Overloaded Operators

For example:

$$
\begin{aligned}
&bool\ operator == (Date,\ Date);\\
&Date\&\ operator + +(Date\&\ d);\\
&Date\ operator + (Date\ d,\ int\ n);\\
&ostream\&\ operator << (ostream\&, Date\ d);\\
&istream\&\ operator >> (istream\&, Date\&\ d);
\end{aligned}
$$

7.8 Destructors

Destructors clean up and release resources. Destructors are called *implicitly* when an automatic variable goes out of scope, an object on the free store is deleted. Destructor notation uses the complement symbol ~ to hint at the destructors' relation to the constructor.

7.9 Default Constructors

- A default constructor is a constructor that can be called without supplying an argument.
- If a user has declared a default constructor, that one will be used; otherwise, the complier will try to generate one if needed and if the user hasn't declared other constructors. A compiler generated default constructor implicitly calls the default constructors for a class' members of class type and bases. Non-class types (build in types) are not initialized (although build-in types also have default constructors).
- Because *const*s and references must be initialized, a class containing *const* or reference members cannot be default-constructed unless the programmer explicitly supplies a constructor.

7.10 Construction and Destruction

7.10.1 Local Variables

The constructor for a local variable is executed each time the thread of control passes through the declaration of the local variable. The destructor for a local variable is executed each time the local variable's block is exited.

7.10.2 Copy Objects

- Copying objects of a class by default means a memberwise copy.
- Having assignment interpreted this way can cause a surprising effect when used on objects of a class with pointer members.
- Copy constuctor initialize un-initialized memory, whereas the copy assignment operator must correctly deal with a well-constructed object.
- Assignment operator: protect against self-assignment, delete old elements, initialize, and copy n new elements.
- Every nonstatic member must be copied.

7.10.3 Free Store

An object created on the free store has its constructor invoked by the *new* operator and exists until the *delete* operator is applied to a pointer to it.

7.10.4 Class Objects as Members

- Arguments for a member's constructor are specified in a member initializer list in the definition of the constructor of the containing class. The member initializers are preceded by a colon and separated by commas.
- If a member's constructor needs not arguments, the member needs not to be mentioned in the member initializer list.

- A constructor assembles the execution environment for the member functions for a class from the bottom up(members first). The destructor disasssembles it from the top down(members last).
- There are two ways to initialize members: by initializer and by assignment. For objects of a class without default constructors, for *const* members, and for references, initializer must be used. (A default assignment cannot be generated for them)
- It is also possible to initialize a static integral constant member by adding a $constant - expression$ initializer to its member declaration.
- If and only if you use an initialized member in a way that requires it to be stored as an object in memory, the member must be defined somewhere, but the initializer may not be repeated.
- We can use an enumerator as a symbolic constant within a class declaration.
- Default copy constructor leaves a reference member referring to the same object in both the original and the copied object.

7.10.5 Arrays

If an object of a class can be constructed without supplying an explicit initializer, then arrays of that class can be defined. The destructor for each constructed element of an array is invoked when that array is destroyed.

7.10.6 Local Static Store

The constructor for a local static object is called the first time the thread of control passes through the object's definition.

7.10.7 Placement of Objects

We can place objects anywhere by providing an allocator function with extra arguments and then supplying such extra arguments when using *new*.

$$
\begin{aligned}
&void* \ operator \ new(size\_t, void* \ p)\{return \ p;\} \\
&void* \ buf = reinterpret\_cast<void*>(0xF00F); \\
&X* \ p2 = new(buf)X;
\end{aligned}
$$

Explicit call of a destructors, like the use of special-purpose global allocators, should be avoided wherever possible.

7.10.8 Unions

- A named union is defined as a *struct*, where every member has the same address.
- The type of the object stored in a union is unknown. Consequently, a union may not have members with constructors or destructors.

- Unions are best used in low-level code, or as part of the implmentation of classes that keep track of what is stored in the union.

Chapter 8

Operator Overloading

8.1 Operator Function

$c = a + b$ is a shorthand for an explicit call of the operator function: $c = a.operator + (b)$. Thus, the first operand is itself, the second operand is its argument, and the return value is assigned to variable c.

A binary operator can be defined either a nonstatic member function taking one argument

$$
\begin{array}{l}
class\ X\{ \\
\quad void\ operator + (int); \\
... \\
\};
\end{array}
$$

, or a nonmember function taking two arguments: $void\ operator + (X, X)$.

A unary operator can be defined by either a nonstatic member function taking no arguments or a nonmember function taking one argument.

The operators = (assignment), &(address-of), and ,(sequencing) have predefined meanings when applied to class objects. These predefined meanings can be made inaccessible to general users by making them private.

enumerations are user-defined types so that we can define operators for them:

$$
\begin{array}{l}
enum\ Day\{sun, mon, tue, wed, thu, fri, sat\}; \\
Day\&\ operator + +(Day\&\ d)\{ \\
\quad returnd = (sat == d)\ ?\ sun\ :\ Day(d + 1); \\
\}
\end{array}
$$

An operator is either a member of a class or defined in some namespace. Operators defined in namespaces can be found based on their operand types just like functions can be found based on their argument types.

typedef is just a synonym and not a user-defined type.

Note that in operator look-up no preference is given to members over nonmembers. This differs from look-up of named functions. The lack of hiding of operators ensures that built-in operators are never inaccessible and that users can supply new meanings for an operator without modifying existing class declarations.

To minimize the number of functions that directly manipulate the representation of an object. This can be achieved by defining only operators that inherently modify the value of their first argument, such as $+=$. Operators that imply produce a new value based on the values of its arguments, such as $+$, are then defined outside the class and use the essential operators in their implementation.

Composite assignment operators such as $+=$ and $*=$ tend to be simpler to define than their simple counterparts $+$ and $=$. It follows from the fact that three objects are involved in a $+$ operator, but one two objects are involved in a $+=$ operator.

For mix-mode arithmetic between different types, we just add more overloading operator functions.

A constructor requiring a single argument need not be called explicitly.

when a constructor is explicitly declared for a type, it is not possible to use an initializer list : $struct\ T\ x = \{3,\ 4\}$. If we have already defined a constructor for the type T, we can no longer use initializer list to initialize the object x.

For types where the default copy constructor has the right semantics, I prefer to rely on that default.

We need only to be able to read the real and imaginary parts; writing them is less often needed. If we must do a *partial update*, we can

$$
\begin{array}{l}
void\ f(complex\&\ z,\ double\ d)\{ \\
\qquad //... \\
\qquad z = complex(z.real(), d); \\
\}
\end{array}
$$

8.2 Conversion Operators

A constructor cannot specify

- an implicit conversion from a user-defined type to a built-in type.
- a conversion from a new class to a previously defined class. (without modify the declaration for the old class)
- a member function $X :: operator\ T()$, defines a conversion from X to T. Note that the type being converted to is part of the name of the operator and cannot be repeated as the return value of the conversion function. In this respect also a conversion operator resembles a constructor (in a different type).
- If both user-defined conversions and user-defined operators are defined, it is possible to get ambiguities between the user-defined operators and the built-in operators. It is therefore often best to rely on either user-defined conversions or user-defined operators for a given type.
- User-defined conversions are considered only if they are necessary to resolve a call.

Implicit conversions are not used for non-*const* reference arguments.

8.3 Friend Functions

An ordinary member function declaration specifies three logically distinct things:

- The function can acccess the private part of the class declaration
- the function is in the scope of the class
- the function must e invoked on an object (has this pointer)

By declaring a member function ***static***, we can give it the first two, by declaring a function ***friend***, we can give it the first property only.

A *friend* declaration can be placed in either the private or the public part of a class declaration; it does not matter where. A friend function is explicitly declared in the declaration of the class. It is therefore as much a part of that interface as is a member function.

A member function of one class can be friend function of another class. (This implies that a friend function may not belongs to any of the classes.)

- It is not unusual for all functions of one class to be friends of another. There is a shorthand for this: declaring a *friend* class.
- A friend class must be previously declared in an enclosing scope or defined in the non-class scope immediately enclosing the class that is declaring it a friend.
- A friend function can be explicitly declared just like friend classes, or it can be found through its arguments even if it was not declared in the immediately enclosing scope.

8.4 Essential Operators

If a class X has a destructor that performs a nontrivial task, such as free-store deallocation, the class is likely to need the full complement of functions that control construction, destruction and copying.

There are three more cases in which an object is copied: as a function argument, as a function return value, and as an exception.

Implicit conversion can be suppressed by declaring a constructor ***explicit***, Explicit constructor will be invoked only explicitly. By declaring the constructor *explicit*, we make sure that A to B happens only when we ask for it and that accidental assignments are caught at compile time.

8.5 Subscripting

$$
\begin{aligned}
&double\&\ Assoc :: operator[\,](string\&\ s)\{\\
&\quad for(vector < Pair >:: iterator\ p = vec.begin();\\
&\qquad p! = vec.end();\ ++p)\\
&\qquad \{if(s == p- > name) return\ p- > val;\}\\
&\quad vec.push\_back(Pair(s, 0));\\
&\quad return\ vec.bakc().val;\\
&\}
\end{aligned}
$$

```
int main(){
string but;
Assoc vec;
while(cin >> buf){vec[buf]++};
}
```

An operator[ ]() must be a member function.

8.6 Function Call

```
class Add{
        complexval;
public :
        Add(complex c){val = c;}
        Add(double r, double i){val = complex(r, 1);}
        voidoperator()(complex& c) const {c+ = val;}
};

for_each(aa.begin(); aa.end(); Add(2, 3));
```

The most obvious, and probably also the most important, use of () operator is to provide the usual function call syntax for objects that in some way behave like functions. An object that acts like a function is often called a *function-like object* or simply a *function object*.

8.7 Dereferencing

```
struct Rec{
        string name;
        //...
}
class Rec_ptr{
        const char * identifier;
        Rec * in_core_address;
        //...
public :
        Rec_ptr(const char * p) : identifier(p), in_core_address(0){}
        ~Rec_ptr(){write_to_disk(in_core_address, identifier);}
        Rec * operator- > ();
};
Rec * Rec_ptr :: operator- > (){
        if(in_core_address == 0){
                in_core_address = read_from_disk(identifier);
}
        returnin_core_address;
};
```

```
void update(const char * s){
    Rec_ptr p(s);
    p-> name = "Rose";
    //...
};
```

Operator $->$ must be a member function. If used, its return type must be a pointer or an object of a class to which you can apply $->$.

Overloading $->$ is primarily useful for creating *smart pointer*, that is, objects that act like pointer and in addition perform some action whenever an object is accessed through them.

8.8 Increment and Decrement

"Smarter" increment and Decrement, which could provide more functions.

```
class Ptr_to_T{
    T * p;
    T * array;
    int size;
public :
    Ptr_to_T(T * p,T * v,int s);
    Ptr_to_T(T * p);
    Ptr_to_T& operator ++(); //prefix
    Ptr_to_T operator ++(int); //postfix
    Ptr_to_T& operator --(); //prefix
    Ptr_to_T operator --(int); //postfix
};
```

The *int* argument is used to indicate that the function is to be invoked for postfix application of ++. This *int* is never used, the argument is simply a dummy used to distinguish between prefix and postfix application. We can provide more functionalists such as error-checking in the smarter increment and decrement.

Chapter 9

Derived Classes

9.1 Derived Classes

Using a class as a base is equivalent to declaring an (unnamed) object of that class. A derived class is constructed bottom-up, first the base class, then its members, then the derived class itself. They are destroyed in the opposite order.

- In general, if a class *Derived* has a public base class *Base*, then a *Derived** can be assigned to a variable of type *Base** without the use of explicit type conversion. The opposite conversion, from *Base** to *Derived**, must be explicit.
- If base class has constructors, then one of them must be invoked in the constructor of the derived class. The base class acts exactly the same like a member of the derived class.
- Copying a derived class object to assign it to a base class object may lead to *slicing*: only a part of members are assigned. *Slicing* should be avoided.
- A function from a derived class with the same name and the same set of argument types as a virtual function in a base is said to ***override*** the base class version of the virtual function. Except where we explicitly say which version of a virtual function is called(as in the call BASE::fun()), the overriding function is chosen as the most appropriate for the object for which it is called. Without the scope operator ::, there will be infinite recursion.
- The keyword *virtual* need not to be repeated in the derived classes.
- Getting the right behavior from *base* functions independently of exactly what kind of *derived* class is actually used is called ***polymorphism***. A type with a virtual function is call a *polymorphism type*.

$$
\begin{aligned}
&Derived :: Derived(const\ string\&\ n,\ int\ d,\ int\ lvl)\\
&\quad : Base(n, d),\\
&\quad\ level(lvl)\{\\
&\quad\ //...\\
&\}
\end{aligned}
$$

9.2 Abstract Classes

- A vritual function can be made to a *pure virtual function* by initializer $= 0$. A class with one or more pure virtual functions are *abstract class*.
- An abstract class can be used only as an interface and as a base for other classes.
- A pure virtual function that is not defined in a derived class remains a pure virtual function, so the derived class is also an abstract class. This allows us to build implementations in stages.
- An important use of abstract classes is to provide an interface without exposing any implementation details.

When there is no data in abstract class, a constructor is not needed. Adding a virtual destructor can ensure proper cleanup of the data that will be defined in the derived classes. Since if you do not override the virtual destructor in derived class, this class is still an abstract class, and can not creates any objects

Deriving directly from more than one class is usually called *multiple inheritance*.

Chapter 10

Templates

10.1 Function Template

A version of a template for a particular template argument is called a *specialization.*

10.1.1 Function Template Argument

C++ can deduce the template argument for a call from the function arguments.

```
template < class T, int max > T& lookup(Buffer < T, max > & b,
                                        const char * p);
class Record{
      const char v[12];
      //...
}
Record& f(Buffer < Record, 128 > & buf, const char * p){
      return lookup(buf, p);
}
```

In this example, but the call of the template function *lookup*, the template argument T is deduced as *Record*, and *max* is deduced as 128, although neither *Record* or 128 are explicitly given in the function call.

```
template < class T, class U > T implicit_cast(U u){
      return u;
}
void g(int i){
      implicit_cast < double > (i);
}
```

In this example, the first type T can not be deduced from the function call, so we need to explicitly give type for T in a pair of angle brace.

10.1.2 Function Template Overloading

$template < class\ T > T\ sqrt(T);$
$template < class\ T > complex < T >\ \ sqrt(complex < T >);$
$double\ sqrt(double);$

We use the usual function overload resolution rules to these specializations and all ordinary functions:

- consider all possible template specifications, and choose the most specialized.
- If a template function argument has been determined by template argument deduction, that argument cannot also have promotions, standard conversions, or user-defined conversions applied. For $sqrt(2)$, $sqrt < int >(int)$ is an exact match, so it is preferred over $sqrt(double)$.
- If a function and a specialization are equally good matches, the function is preferred. Consequently, $sqrt < double > (double)$ is preferred over $sqrt < double > (double)$ for $sqrt(2.0)$.

We could resolve the two ambiguities either by ex;licit qualification or by adding suitable declaration.

$template < class\ T > T\ max(T,\ T);$
$void\ f(){$
    $max < double > (2.7,\ 4);$
$}$

We need explicitly add $< double >$ to clarify the ambiguity.

Passing the operations as a template parameter has significant benefit compared to alternatives such as passing pointers to functions: Several operations can be passed as a single argument with no run-time cost.

Each class generated from a class template gets a copy of each *static* member of the class template.

Type argument in Template definition can also pick a default.

10.2 Specification

A vector template:

$$template < class\ T > class\ Vector\{...\}$$

To obtain a contrainer of pointer, two step:

- define a specialization of *Vector* for pointers to *void*
- define a partial specialization

$template <> class\ Vector < void* > \{...\}$
$template < class\ T > class\ Vector < T* >:\ private\ Vector < void* >$
$\{...\}$

The general template must be declared before any specialization

$$template < class\ T > class\ List\{/ * ... * /\};$$
$$template < class\ T > class\ List < T* > \{/ * ... * /\};$$
$$List < int* > \ li;$$

Specialization is also useful for template functions.

$$template < class\ T > bool\ less(T\ a,\ T\ b)\{/ * ... * /\}$$
$$template <> bool\ less(const\ char * \ a,\ const\ char * \ b)\{/ * ... * /\}$$

template argument can be deduced from the function call.

A overloaded template functions example:

$$template < class\ T > void\ swap(T\&\ x,\ T\&\ y)\{/ * ... * /\}$$
$$template < class\ T > void\ swap(vector < T > \&\ a, Vector < T > \&\ b)$$
$$\{...\}$$

Specialization and overloading are useful when there is a more efficient alternative to a general algorithm for a set of template arguments. In addition, specialization comes in handy when an irregularity of an argument type causes the general algorithm to give an undesired result.

10.3 Derivation and Templates

Deriving a template class from a non-template class is a way of providing a common implementation for a set of templates.

It is also useful to derive one template class from another. If members of a base class depend on a template parameter of a derived class, the base itself must be parameterized.

$$template < class\ T > class\ vector\{/ * ... * /\}$$
$$template < class\ T > class\ Vec\ :\ public\ vector < T > \{/ * ... * /\}$$

- virtual functions provide *run-time polymorphism*
- templates offer *compile-time polymorphism* or *parametric polymorphism*

If no hierarchical relationship is required for objects, they are best used as template arguments. If the actual types of these objects cannot be known at compile-time, they are best represented as classes derived from a common abstract class.

- A class or a class template can have members that are themselves template.
- copy constructor and copy assignment can not be template member.
- template member can not be *virtual* (virtual function table can not be used any more.)

There can not be any default relationship between classes generated from the same templates. Member templates allow us to specify many such relationships where desired.

$$
\begin{array}{l}
template < class\ T > \\
\quad template < class\ T2 > \\
\quad\quad Ptr < T >:: operator\ Ptr < T2 > (\ )\{ \\
\quad\quad\quad return\ Ptr < T2 > (p); \\
\quad\quad \}
\end{array}
$$

The template parameter lists of a template and its template member cannot be combined.

To be accessible from other compilation units, a template definition must be explicitly declared *export*. Otherwise, the definition must be in scope wherever the template is used.

Chapter 11

Exception Handling

Chapter 12

Class Hierarchies

12.1 Multiple Inheritance

When two classes have member functions with the same name, a member function in the derived class overrides both of them. If the overriding member function in derived class does not resolve the ambiguity, the compiler recursively looks in its base classes.

12.1.1 Ambiguity Resolution

- Overload resolution is not applied across different class scopes. In particular, ambiguities between functions from different vase classes are not resolved based on argument types.

- *Using-declaration* allow a programmer to compose a set of overloaded functions from base classes and the derived class. functions declared in the derived class hide functions that would otherwise be available from a base.

- *Using-declaration* may not be used for a member of a class from outside that class, its derived classed, and their member functions. A *using-directive* may not appear in a class definition.

12.1.2 Replicated Base Classes

- For replicated base classes, two separate objects are used to represent them in the derived class.

- When declaring the replicated base classes as *virtual*, one shared object will be used by the derived class.

- Different derived classes override the same function is allowed if and only if some overriding class is derived from every other class that overrides the function.

12.2 Access Control

- For a private member, its name can be used only by member functions and friends of the class in which it is declared. (functions implementing the class)
- For a protected member, its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class. (function implementing the derived class)
- For a public member, its name can be used by any function. (other functions)

In a *class*, a member is by default private; in a *struct*, a member is by default public.

Declaring data member protected is usually a design error. There have always been alternatives to placing significant amounts of information in a common base class for derived classes to use directly. Note that one of these objections are significant for protected member functions, protected is a fine way of specifying operations for use in derived classes.

12.3 Access to Base Classes

- Public derivation makes the derived class a subtype of its base; this is the most common form of derivation. The base class may only provide a interface.
- Protected and private derivation are used to represent implementation details. The user of derived class can not directly use facilities of the base class.
- Protected base are useful in class hierarchies in which further derivation is the norm
- Private bases are most useful when defining a class by restricting the interface to a vase so that stronger guarantees can be provided.

The access specifier for a base class can be left out. In that case, the base defaults to a private base for a class and a public base for a struct. For readability, it is always to use an explicit access specifier.

Consider D is derived class and B is base class:

- B is private: its public and protected member can be used only by member functions and friends of D. Only friends and member of D can convert $D*$ to $B*$.
- B is protected: its public and protected member can be used only by member functions and friends of D or derived class of D. Only friends and member of D or derived class of D can convert $D*$ to $B*$.
- B is public: its public member functions can be used by any function. its protected member can be used only by member functions and friends of D or derived class of D. Any function can convert $D*$ to $B*$.

If a name can be reached through multiple paths in a multiple inheritance lattice, it is accessible if it is accessible through any path. If a single entity is reachable through several paths, we can still refer to it *without ambiguity*. A *using-declaration* cannot be used to gain access to additional information.

12.4 Run-Time Type Information

$$dynamic\_cast < T* > (p)$$

The purpose of *dynamic_cast* is to deal with the case in which the correctness of the conversion cannot be determined by the compiler. A *dynamic_cast* requires a pointer or a reference to a polymorphic type in order to do a down-cast or a cross-cast.

A *dynamic_cast* to *void ** can be used to determine the address of the beginning of an object of polymorphic type.

- *dynamic_cast* $< T* >$ (p): Is the object pointed to by p is type T? if not, return 0.
- *dynamic_cast* $< T\& >$ (r): The object referred to by r is of type T? if the assertion is not correct, a *bad_cast* exception is thrown.

For replicated virtual base, $run-timeambiguity$ may happens when down-casting. *up-casting* are caught at compiling time.

- *static_cast* does not examine the object it casts from. It can not cast from virtual base. The *dynamic_cast* requires a polymorphic operand because there is no information stored in a non-polymorphic object that can be used to find the objects for which it represents a base.
- *dynamic_cast* cannot cast from a $void*$. In this case, *static_cast* is needed.
- Both *dynamic_cast* and *static_cast* can not violate base access, and can not cast away *const*.

Class construction is bottom-up, while class destruction is top-down. It is best to avoid calling virtual functions during construction and destruction.

$$const\ type_info\&typeid(type_name)throw();$$

Use *typeid* to know the exact type of an object. RTTI can be used to write thinly disguised *switch-statement*. Using *dynamic_cast* rather than *typeid* would improve the *switch* only marginally.

Many examples of proper use of RTTI arise when some service code is expressed in terms of one class and a user wants to add functionality through derivation.

12.5 Pointer to Members

$$void(T :: * pm)() = \&T :: operation1;$$

A static member is associated with a particular object, so a pointer to a static member should be simply an ordinary function pointer.

We can safely assign a pointer to a member of a base class to a pointer to a pointer of a derived class. (*contravariance*)

12.6 Free Store

The operators *operator new()* and *operator delete()* can be overloaded. They are implicitly ***static***.

A constructor can not be ***virtual***, but the effect can be abtained: creating a new object without knowing its exact type.

```
class expr{
public :
    Expr();
    Expr(const Expr&);
    virtual Expr * new_expr() const {return new Expr();}
    virtual Expr * clone() const {return new Expr(*this);}
    //...
};
class Cond : public Expr{
public :
    Cond();
    Cond(const Cond&);
    virtual Cond * new_expr() const {return new Cond();}
    virtual Cond * clone() const {return new Cond(*this);}
    //...
};
```

List of Figures

List of Tables