Highlights and Condensed version

PURPOSE OF THE DOCUMENT

C++ in a Nutshell is a lean, focused reference that offers practical examples for the most important, most often used, aspects of C++. *C++ in a Nutshell* packs an enormous amount of information on C++ (and the many libraries used with it) in an indispensable quick reference. The author of the book is Ray Lischner and the pdf version is available in the link: https://docs.google.com/file/d/0B5_mAdKvdKTlUHZ4eF82LXdldEE/view. This document highlights and condenses the contents of the book and makes it easier for the programmer to grasp the concepts quickly and efficiently.

Table of Contents

CHAPTER 1: LANGUAGE BASICS

CV-QUALIFIERS

- const

Denotes an object that cannot be modified. A const object cannot ordinarily be the target of an assignment. You cannot call a non-const member function of a const object.

- volatile

Denotes an object whose value might change unexpectedly. The compiler is prevented from performing optimizations that depend on the value not changing. For example, a variable that is tied to a hardware register should be volatile .

ARRAYS

An array is declared with a constant size specified in square brackets. The array size is fixed for the lifetime of the object and cannot change. (For an array-like container whose size can change at runtime, see <vector> in Chapter 13.) To declare a multidimensional array, use a separate set of square brackets for each dimension:

```
int point[2];           double matrix[3][4]; // A 3 × 4 matrix
```

You can omit the array size if there is an initializer; the number of initial values determines the size. In a multidimensional array, you can omit only the first (left-most) size:

```
int data[] = { 42, 10, 3, 4 }; // data[4]

int identity[][3] = { { 1,0,0 }, {0,1,0}, {0,0,1} }; // identity[3][3]

char str[] = "hello";                                 // str[6], with trailing \0
```

POINTERS

A pointer object stores the address of another object. A pointer is declared with a leading asterisk (*), optionally followed by cv-qualifiers, then the object name, and finally an optional initializer.

```
int i, j;                       constant int (left)                int i, j;

int const *p = &i;                    vs                       int * const p = &i;

p = &j; // OK                   constant pointer (right)           p = &j; //
Error

*p = 42; // Error                                                *p = 42; // OK
```

FUNCTION POINTERS

A function pointer is declared with an asterisk (*) and the function signature (parameter types and optional names). The declaration’s specifiers form the function’s return type. The name and asterisk must be enclosed in parentheses, so the asterisk is not interpreted as part of the return type. An optional exception specification can follow the signature.

```
void (*fp)(int);     // fp is pointer to function taking int and returning void
void print(int);
fp = print;
```

hard to read --> better use typedef (note the additional step of "instantiation")

```
typedef void (*Function)(int);
Function fp;
fp = print;
int* (*fp[10])(int*(*)(int*), int);      // Array of 10 function pointers
                                    // return type of functions: pointer to int
                                                                    (int*)
                                    // function argument 1:
                                    // function pointer, again with return type
                                          int* and argument int*
                                    // function argument 2: int (how
                                          boring ;-) )
```

REFERENCES

A reference is a synonym for an object or function. A reference is declared just like a pointer, but with an ampersand (&) instead of an asterisk (*). A local or global reference declaration must have an initializer that specifies the target of the reference. Data members and function parameters, however, do not have initializers. You cannot declare a reference of a reference, a reference to a class member, a pointer to a reference, an array of references, or a cv-qualified reference. A reference, unlike a pointer, cannot be made to refer to a different

object at runtime. Assignments to a reference are just like assignments to the referenced object. References are often used to bind names to temporary objects, implement call-by-reference for function parameters, and optimize call-by-value for large function parameters.

// some custom examples here .. malloc .. normal call by reference ... etc
Even call-by-value can be simulated with a const reference as parameter, containing the performance gain (zero copy) of the argument.A reference must be initialized so it refers to an object. If a data member is a refer-ence, it must be initialized in the constructor’s initializer list

NAMESPACES

A namespace is a named scope. By grouping related declarations in a namespace, you can avoid name collisions with declarations in other namespaces. Multiple functions with the same syntax (return type, name and parameters) may exist consider popular names like

```
init()   or   initialize()   reset()   clear()   etc.
```

Define a namespace with the namespace keyword followed by an optional identifier (the namespace name) and zero or more declarations in curly braces. Namespace declarations can be discontinuous, even in separate source files or headers. The namespace scope is the accumulation of all definitions of the same namespace that the compiler has seen at the time it looks up a given name in the namespace. Namespaces can be nested.

```
using Declaration          vs          using directives

using std::cout;                       using namespace std;

imports one function                   adds a complete second
in the current scope                 namespace (here std) to the list
                                           of namespaces to search

                                       use with care
```

Unlike a using declaration, no names are added to the current namespace. Instead, the used namespace is added to the list of namespaces to search right after the innermost namespace that contains both the current and used namespaces. (Usually, the containing namespace is the global namespace.) The using directive is transitive, so if namespace A uses namespace B, and namespace B uses namespace C, a name search in A also searches C.

Namespaces have no runtime cost. Don’t be afraid to use them, especially in large projects in which many people contribute code and might accidentally devise conflicting names. The following are some additional tips and suggestions for using namespaces:

1. When you define a class in a namespace, be sure to declare all associated operators and functions in the same namespace.

2. To make namespaces easier to use, keep namespace names short, or use aliases to craft short synonyms for longer names.

3. Never place a using directive in a header. It can create name collisions for any user of the header.

4. Keep using directives local to functions to save typing and enhance clarity.

5. Use using namespace std outside functions only for tiny programs or for back-ward compatibility in legacy projects.

CHAPTER 2: DECLARATION

LVALUE vs RVALUE

- LValue is an object reference , RValue is a value

```
int x = 42; // x is lvalue, 24 is rvalue ... 42 cannot be on the left of an expression
```

- Functions may be both

```
malloc(&a, size) // lvalue -- returns reference

int x = add(5, 3) // rvalue -- function returns value
```

- Operators reacquiring an lvalue:

```
Assignment  (=)   Referencing (&)   increment & decrement (++, --)
```

TYPE CONVERSIONS

In an arithmetic expression, binary operators require operands of the same type. If this is not the case, the type of one operand must be converted to

match that of the other operand. When calling a function, the argument type must match the parameter type; if it doesn't, the argument is converted so its type matches. C++ has cast operators, which let you define a type conversion explicitly, or you can let the compiler automatically convert the type for you.

```
// some examples

// adding int with double, double with float --> try sizeof or typeof to see result ?

// also failing example --- use with care, better cast or use 1.0f as float

// using true / false as 1 /0

// also some examples with function parameters and implicit numeric conversion
```

TYPE CASTS

A type cast is an explicit type conversion. C++ offers several different ways to cast an expression to a different type. The different ways expose the evolution of the language. The six forms of type cast expressions are:

`(type) expr`	
`type (expr)`	
`const_cast<type>(expr)`	force expression to be const
`dynamic_cast<type>(expr)`	cast base class to child class - one virtual function must exist
`reinterpret_cast<type>(expr)`	e.g. convert int to enum also for void* casts (from and to)
`static_cast<type>(expr)`	converting one type of pointer to another

CHAPTER 3: EXPRESSION RULES

C++ has the usual unary operators such as logical negation (!a), binary operators such as addition (a+b), and even a ternary operator (a?b:c). Unlike many other languages, an array subscript is also an operator (a[b]), and a function call is an n-ary operator (e.g., a(b, c, d)). When reading C++ expressions, you must be aware of the precedence and asso-ciativity of the operators involved. For example, *ptr++ is read as *(ptr++) because the postfix ++ operator has higher precedence than the unary * operator.

The logical operators (&& and ||) perform short-circuit evaluation. The left operand is evaluated, and if the expression result can be known at that point, the right operand is not evaluated: (Please avoid overloading && and ||)

```
if (false && f( )) ... // f( ) is never called.
if (true || f( )) ... // f( ) is never called.
```

THIS

Refers to the target object in a nonstatic member function. Its type is a pointer to the class type; its value is an rvalue

```
class Person{
      virtual ~Person() {} // destructor
};
class Student : public Person { };
class IT_student : public Student { };
int main(){
      Person* s = new Student; // object created with new --> lies on heap
                               --> dont forget to call destructor
      dynamic_cast<IT_student>(s); // error ... true class of s is Student,
                               Student has not derived from IT_student
      IT_student* ss = new IT_student;
      s = ss;
      s = dynamic_cast<Person*> ss // works, but not necessary, same as above
      delete(ss);       // delete all objects to avoid memory leaks
      delete(s);
      return 0;
}
```

NEW

Allocates and initializes a dynamic object or array of objects. The new expression first calls an allocation function (operator new) to allocate memory. It then constructs the object in the allocated memory. A class can provide its own allocation function by overriding operator new as a member function. Otherwise, a global operator new is called. see also new[] - analogous: delete and delete[]

```
new (expr-list) type ( optional-expr-list )
```

SHIFT EXPRESSIONS

```
expr1 << expr2 // left shift  = multiplcation with 2

expr1 >> expr2 // right shift = division by 2

overloaded by IO Streams (std::cout << "hello << std::endl)
```

RELATIONAL EXPRESSIONS

A relational expression compares two values for relative order. Relational operators have higher precedence than equality operators, so the following two expressions are equivalent:

```
a < b == c > d          →          (a < b) == (c > d)
```

The result of a relational expression is an rvalue of type bool - care with pointers!!!

EQUALITY EXPRESSIONS

An equality expression compares two values to see if they are equal or different. The result of an equality expression is an rvalue of type bool.

!!! IMPORTANT !!! never do "float-equal-checks"

Note that comparing the results of floating-point values for equality rarely gives the result you want. Instead, you probably want a fuzzy comparison that allows for floating-point imprecision.

When comparing pointers, the pointers must have the same type (after the usual conversions). The pointers are equal if any of the following conditions hold, and are not equal if none of the conditions hold:

• Both pointers are null pointers.

• Both object pointers point to the same object.

• Both object pointers point to one element past the end of the same array.

• Both function pointers point to the same function.

• Both member pointers point to the same member of the same most-derived object.

• Both member pointers point to any data members of the same union.

The equality operators have the following syntax:

```
expr == expr                                  // Returns true if the operands are equal.

expr != expr      or    expr not_eq expr      // Returns false if the operands are equal.
```

BITWISE OPERATIONS

Bitwise and (&, bitand) - bitwise or (|, bitor) - bitwise xor (^, xor)

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Logical AND (&&, and) / logical OR (||, or)

CONDITIONAL EXPRESSIONS

```
condition ? true-expr : false-expr
```

either true-expr or false expr are evaluated - never both

ASSIGNMENT EXPRESSIONS

An assignment expression assigns its right operand to its left operand. In addition to plain assignment (x = y), there are several other assignment operators, each of which is a shorthand for an arithmetic operation and an assignment. The left operand of an assignment expression must be a modifiable lvalue. The result is an lvalue: the left operand after the assignment is complete.

There exist assignment operators for the basic calculus (+= ; -= ; *= ; /=), for the modulo operation (%=), for logical expressions (&= ; |= ; ^=) and for shift operations (>>= ; <<=). Throwing an exception also assigns an object to the thrown exception.

CHAPTER 4: STATEMENTS

A statement with no expression (only a semi-colon ;) is called a null statement. Null statements are most often used for loops when no code is needed in the loop body.

if-else	**return;**
while and for (including continue and break)	**return expr;**
do-while	**goto & labels**
switch	

Every engineer should be familiar with the basic programming concepts, thus this section will not be explained in detail here.

Please avoid using goto & labels, as it is usually an indication for bad programming style.

EXCEPTION HANDLING

```
try - catch

custom exception class

class bad_data : public std::out_of_range {

};
```

Usage:

```
throw bad_data();
```

Functions that does not throw any exceptions need empty parenthesis

```
int noproblem(int x, int y) throw( );
```

CHAPTER 5: FUNCTIONS

FUNCTION DECLARATION, FUNCTION DEFINITION AND FUNCTION CALL

A function declaration contains the return type, the name and the argument list (which can be empty). Moreover, CV-Specifieres can be added. It does not contain a function body.

```
    void doSomething(int param1, bool param2) const throws exception;
```

A function definition usually repeats the declaration and extends it with a function body

```
void doSomething(int param1, bool param2){
      if(param2)
            param1++;
}
```

A function call is invoked by providing the function name together with real parameters for the function arguments.

```
doSomething(5, true);
```

return type can be modified with additional type specifiers such as

FUNCTION SPECIFIERS: EXPLICIT, INLINE AND VIRTUAL

- Explicit

An explicit constructor cannot be used in an implicit type conversion. // Maybe a bit more here ?? // TODO

- inline

Tells the compiler to expand the function body at the point of the function call. An inline function must be defined in every file in which it is used, and the definition must be identical in every file. If the function body contains a local static object, including string literals, every expansion of the function in every file refers to a common object. A function definition in a class definition is an inline function definition, even without the use of the inline specifier. The inline specifier is a hint to the compiler, and the compiler is free to ignore the hint. Most compilers impose a variety of restrictions on which functions can be expanded inline. The restrictions vary from one compiler to another. For example, most compilers cannot expand a recursive function. Inline functions are most often used for extremely simple functions. For example, all standard containers have a member function empty, which returns true if the container is empty. Some containers might implement the function as the following inline function:

```
inline bool empty( ) const { return size( ) != 0; }
```

- virtual

Applies only to nonstatic member functions. A virtual function's definition is bound to the function call at runtime instead of at compile time. See Chapter 6 for details. return type cant be array or reference

DEFAULT ARGUMENTS AND CV-QUALIFIERS

Default arguments have to be assigned from right to left. Lookup of default parameters takes place during compile time, not runtime (p. 102)

BE CAREFUL ! MY ADVISE: AVOID USING DEFAULT PARAMETERS

(→ do not work for polymorphism and inheritance either)

Also functions can (and should) be declared with const or volatile. These qualifiers are usually added after the parameter list but before the exception statement (if any).

In general, member functions that do not change *this should be declared const → getter are usually const.

FUNCTION OVERLOADING

A single function name can have multiple declarations. If those declarations specify different function signatures, the function name is overloaded. A function call to an overloaded function requires the compiler to resolve the overloaded name and decide which function to call. The compiler uses the argument types (but not the return type) to resolve calls to overloaded functions.

```
int add(const int a, const int b);      // adds Integers
int add(Person x, Person y);            // adds Persons - whatever that means
                                                    (maybe age)
```

Overloaded functions must differ in their parameter lists: they must have a different number of parameters, or the parameter types must be different. Otherwise the functions is overwritten! Be careful!

A derived class cannot overload a function that is declared in a base class. Instead, if the derived class has a function with the same name, the derived class hides the name that would be inherited from the base class (or the derived class might override a virtual function.

- Operator overloading

For user-defined types (classes and enumerations), you can define alternate behavior for the C++ operators. This is called overloading the operators. You cannot define new operators, and not all operators can be overloaded (e.g. :: , . , .* ,)

```
enum logical { no, maybe, yes };

logical operator !(logical x);

logical operator &&(logical a, logical b);
```

- Defining Commutative Operators

When overloading a binary operator, consider whether the operator should be commutative (a + b is the same as b + a). If that is the case, you might need to define two overloaded operators.

Please avoid using Operator overloading as it is a common source of mistakes, although the performance could benefit.

TYPE CONVERSION

A class can declare type conversion operators to convert class-type objects to other types. The operator functions must be nonstatic member functions. The name of each operator is the desired type, which can be a series of type specifiers with pointer, reference, and array operators, but cannot be a function or array type:

```
class bigint {
      public:
            operator long();              // Convert object to type long
            operator unsigned long();
            operator const char*();       // Return a string representation
};
```

THE MAIN FUNCTION

Every program must have a function in the global namespace called main, which is the main program. This function must return type int . The C++ environment calls main; your program must never call main. The main function

cannot be declared inline or static. It can be called with no arguments or with two arguments:

```
int main()                    or:          int main(int argc, char* argv[])
```

The argc parameter is the number of command-line arguments, and argv is an array of pointers to the command-line arguments, which are null-terminated character strings. By definition, argv[argc] is a null pointer. The first element of the array (argv[0]) is the program name or an empty string.

CHAPTER 6: CLASSES

C++ supports object-oriented programming with a traditional, statically-typed, class-based object model. That is, a class defines the behavior and the state of objects that are instances of the class. Classes can inherit behavior from ancestor classes. Virtual functions implement type polymorphism, that is, a variable can be declared as a pointer or reference to a base class, and at runtime it can take on the value of any class derived from that base class. C++ also supports C-style structures and unions using the same mechanism as classes.

A class definition starts with the class, struct or union keyword. The difference between a class and a struct is the default access level. A class definition can list any number of base classes, some or all of which might be virtual.

In the class definition are declarations for data members (called instance variables or fields in some other languages), member functions (sometimes called methods), and nested types. A class definition defines a scope, and the class members are declared in the class scope. The class name itself is added to the class scope, so a class cannot have any members, nested types, or enumerators that have the same name as the class.

You can declare a name as a class, struct or union without providing its full definition. This incomplete class declaration lets you use the class name in pointers and references but not in any context in which the full definition is needed. You need a complete class definition when declaring a nonpointer or nonreference object, when using members of the class, and so on. This can be used for forward declaration if two classes refer to each other. Please avoid using structs or unions.

STATIC AND MUTABLE

- static

storage class - every instance of the class with the static class member share this member. In other words, there is a single copy of each static data member regardless of the number of instances of the class. Derived classes also share the single static data member. For non-static members, every object has its own copy of that member.

- mutable

Non-static data members declared with mutable can be changed even if the object is declared as const.

Only an integral or enumerated static data member can have an initializer in the class definition. The initializer must be a constant integral expression. The value of the member can be used as a constant elsewhere in the class definition. The definition of the member must then omit the initializer. This feature is often used to define the maximum size of data member arrays

```
static const int max_length = 1024;      // initializing normally not allowed
char name_[max_length];
```

MEMBER FUNCTIONS

Member functions implement the behavior of a class. Member functions can be defined within the class definition or separately. You can use the inline function specifier and either the static or virtual (but not both) specifier.

Defining a member function within the class definition declares the function inline, even if you do not use the inline specifier. A nonstatic member function can have const,volatile or both function qualifiers. Qualifiers appear after the function parameters and before the exception specification.

THIS POINTER

A nonstatic member function can be called only for an object of its class or for a derived class. The object is implicitly passed as a hidden parameter to the function, and the function can refer to the object by using the **this** keyword, which represents an rvalue pointer to the object. That is, if the object has type T, **this** is an rvalue of static type T* . In a call to a virtual function, the objects

dynamic type might not match the static type of **this**. Static member functions do not have this pointers.

If the function is qualified with const or volatile, the same qualifiers apply to **this** within the member function. In other words, within a const member function of class T, **this** has type const T*. A const function, therefore, cannot modify its nonstatic data members (except those declared with the mutable specifier).

STATIC MEMBER FUNCTIONS

A static member function is like a function declared at namespace scope; the class name assumes the role of the namespace name. Within the scope of the class, you can call the function using an unqualified name. Outside the class scope, you must qualify the function name with the class name (e.g., cls::member) or by calling the function as a named member of an object (e.g., obj.member). In the latter case, the object is not passed as an implicit parameter (as it would be to a nonstatic member function), but serves only to identify the class scope in which the function name is looked up

Static member functions have the following restrictions:

- They do not have this pointers.
- They cannot be virtual .
- They cannot have const or volatile qualifiers.
- They cannot refer to nonstatic members, except as members of a specific object (using the . or -> operator).
- A class cannot have a static and nonstatic member function with the same name and parameters.

CONSTRUCTORS

Constructors and destructors are special forms of member functions. A constructor is used to initialize an object, and a destructor is used to finalize an object. A constructor cannot have a return type, and you cannot return a value. That is, you must use a plain return; or return an expression of type void . A

constructor cannot have const or volatile qualifiers, and it cannot be virtual or static .

Initializing constructor:

```
class-name(parameters) : member(expr), base-class(expr-list), ...

      {   compound-statement   }
```

A constructor can be declared with the explicit specifier. An explicit constructor is never called for an implicit type conversion, but only for function-like initialization in a declaration and for explicit type casts. See the next section for more information.

SPECIAL CONSTRUCTORS

Two kinds of constructors are special, so special they have their own names:

- default constructors and copy constructors.
- A default constructor can be called with no arguments.

```
Point p;   //default constr.      vs          Point p();   //func. Definition
```

Remember not to use an empty initializer in a declaration. For example, Point p( ); declares a function named p , not a default-initialized Point object. A copy constructor can take a single argument whose type is a reference to the class type. (Additional parameters, if any, must have default arguments.) The reference is usually const , but it does not have to be. The copy constructor is called to copy objects of the class type when passing objects to functions and when returning objects from functions.

```
Point(const Point& pt);                                    // Copy constructor
```

Default and copy constructors are so important, the compiler might generate them for you. Different situations where a (copy- or default or user defined) constructor is called: With automatic variables and constants in a function body:

```
void demo( ){
point p1(1, 2);
const point origin;
```

With static objects that are local or global:

```
point p2(3, 4);
void demo( ){
static point p3(5, 6);
```

Dynamically, with new expressions:

```
point *p = new point(7, 8);
```

With temporary objects in expressions:

```
set_bounds(point(left, top), point(right, bottom));
```

With function parameters and return values:

```
point offset(point p) { p.x(p.x( ) + 2); return p; }
point x(1,2);
x = offset(x);                          // Calls point(const point&) twice
```

With implicit type conversions:

```
point p = 2; // Invokes point(int=0, int=0), then point(const point&)
```

With constructor initializers:

```
derived::derived(int x, int y) : base(x, y) {}
```

DESTRUCTORS

A destructor finalizes an object when the object is destroyed. Typically, finalization frees memory, releases resources (such as open files), and so on. A destructor is a special member function. It has no return type, no arguments, and cannot return a value. A destructor can be virtual or inline, but it cannot be static, const or volatile . The name of a destructor is a tilde (~) followed by the class name. A class has only one destructor (although it may have several constructors).

Member and base-class destructors are called in reverse order of their constructors. If you do not write a destructor, the compiler does so for you. Any class that has virtual functions should have a virtual destructor. Dynamically allocated objects (using new) are destroyed by delete expressions.

IMPLICIT MEMBER FUNCTIONS

The compiler implicitly declares and defines default and copy constructors, the copy assignment operator, and the destructor in certain circumstances. Classes

using pointers usually need custom constructors, copy constructors, assignment operators and destructors, as the pointer cannot be copied straight forward (results in a pointer, pointing to the same adress). A new pointer has to be created and the content of the original pointer provided. A useful guideline is that if you write one of the three special functions (copy constructor, copy assignment operator, or destructor),you will probably need to write all three. Default and copy constructor are implicit defined if there are no user defined ones. Moreover, the compiler implicitly declares a copy assignment operator (operator=) if a class does not have one. The compiler declares an implicit destructor if the programmer does not provide one. If a base class has a virtual destructor, the implicit destructor is also virtual. The implicit destructor is like a programmer-supplied destructor with an empty function body.

INHERITANCE

A class can inherit from zero or more base classes. A class with at least one base class is said to be a derived class. A derived class inherits all the data members and member functions of all of its base classes and all of their base classes, and so on. A derived class can access the members that it inherits from an ancestor class, provided the members are not private. An object with a derived-class type can usually be converted to a base class, in which case the object is sliced. The members of the derived class are removed, and only the base class members remain. This effect change when using pointers!

```
struct file {
     std::string name;
};
struct directory : file {
     std::vector<file*> entries;
};
directory d;
file f;
f = d;                 // Only d.name is copied to f; entries are lost.
```

vs

```
directory* dp = new directory;
file* fp;
fp = dp;               // Keeps entries and identity as a directory object
```

As you can see in the previous examples, the compiler implicitly converts a derived class to a base class. You can also use an explicit cast operator, but if the base class is virtual, you must use dynamic_cast<> , not static_cast<> .

VIRTUAL FUNCTIONS

A nonstatic member function can be declared with the virtual function specifier, and is then known as a virtual function. A virtual function can be overridden in a derived class. To override a virtual function, declare it in a derived class with the same name and parameter types. The return type is usually the same but does not have to be identical. The virtual specifier is optional in the derived class but is recommended as a hint to the human reader. A constructor cannot be virtual, but a destructor can be. A virtual function cannot be static. A class that has at least one virtual function is polymorphic.

DECLARED TYPE VS DYNAMIC TYPE

```
file* f = new directory;      // static type: file <--> dynamic type: directory
```

An objects dynamic type can differ from its static type only if the object is accessed via a pointer or reference .A class that has at least one virtual function should also have a virtual destructor. If a delete expression (see Chapter 3) deletes a polymorphic pointer (for which the dynamic type does not match the static type), the static class must have a virtual destructor. Otherwise, the behavior is undefined.

COVARIANT RETURN TYPES

```
struct shape {
  virtual shape* clone( ) = 0;
};
struct circle : shape {
  virtual circle* clone( ) {
     return new circle(*this);
}
double radius( ) const {
 return radius_;
}
void radius(double r) { radius_ = r; }
```

```
struct square : shape {
      virtual square* clone( ) {
                  return new square(*this);
            }
      private:
            double size_;
            point corners_[4]; };
```

```
private:
  double radius_;
  point center_;
};
```

Note how the overwriting of the clone() function differs also in the return type - every derived class use its own type as return type.

PURE VIRTUAL FUNCTIONS

A virtual function can be declared with the pure specifier (=0) after the function header. Such a function is a pure virtual or abstract function.

ABSTRACT CLASSES

An abstract class declares at least one pure virtual function or inherits a pure virtual function without overriding it. A concrete class has no pure virtual functions (or all inherited pure functions are overridden). You cannot create an object whose type is an abstract class. Instead, you must create objects of concrete type. In other words, a concrete class that inherits from an abstract class must override every pure virtual function.

MULTIPLE INHERITANCE

```
struct base1 { int n; };

struct base2 { int n; };

struct derived : base1, base2 {

     int get_n( ) { return base1::n; } // Plain n is an error.

};
```

Please avoid multiple Inheritance - especially for concrete classes. May be okay for interfaces / abstract classes with only pure virtual functions (=0).

ACCESS SPECIFIERS

Access specifiers restrict who can access a member. You can use an access specifier before a base-class name in a class definition and have access specifier labels within a class definition. The access specifiers are:

```
Public       // Anyone can access a public member.

Protected // Only the class, derived classes, and friends can access protected members.

Private     //Only the class and friends can access private members.
```

In a class definition, the default access for members and base classes is private .The access level of a base class affects which members of the base class are accessible to users of a derived class (not the derived classes access to the base class). The access level caps the accessibility of inherited members. In other words, private inheritance makes all inherited members private in the derived class.

Protected inheritance reduces the accessibility of public members in the base class to protected in the derived class. Public inheritance leaves all accessibility as it is in the base class.

FRIEND

A friend is permitted full access to private and protected members. A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends. Use the friend specifier to declare a friend in the class granting friendship. Note that friendship is given, not taken. In other words, if class A contains the declaration friend class B; , class B can access the private members of A, but A has no special privileges to access B (unless B declares A as a friend). Friendship is not transitive, that is, the friend of my friend is not my friend (unless I declare so in a separate friend declaration) nor is a nested class a friend just because the outer class is a friend. Friendship is not inherited. If a base class is a friend, derived classes do not get any special privileges.

NESTED TYPES

You can declare and define types within a class definition. Accessing these nested types is similar to using types declared within a namespace; the class name serves as the namespace name.

```
class outer {
      public:
            friend inner; // Implicit in Java
      class inner {
            friend outer; // Implicit in Java
      public:
      // Inner member class keeps track of outer instance.
            inner(outer& out) : out_(out) {}
            int get( ) const { return out_.x( ); }
            private:
            outer& out_;
}
int x( );
int y( ) { inner i(*this); return i.get( ); }
};
```

not how the inner class definition is not terminated with a semi-colon (is that a typo ???). To access fields from the inner class, use both class names as scope operator, e.g.

```
 outer::inner::out_
```

CHAPTER 7: TEMPLATES

C++ is more than a mere object-oriented programming language. The full power of C++ is seen when programming with templates. Templates lie at the heart of the standard library: strings, I/O streams, containers, iterators, algorithms, and more. A template takes one or more parameters, and when you instantiate a class or function template, you must supply arguments for the parameters. The classes and functions can have different behaviors or implementations, depending on the arguments. This style of programming is often called generic programming.

Example:

```
vector<int> vec; // list with integer values

vector<boost::shared_ptr<boost::thread> > // vector with shared_ptr pointing on threads - not the empty space between the two closing parenthesis '>'. Otherwise the symbol >> is interpreted as operator (shift, iostream ... )
```

Programming with templates is unlike traditional object-oriented programming. Object-oriented programming centers around type polymorphism (requiring classes, objects, and virtual functions). Template-based programming centers around parametric polymorphism, in which a function or class is defined indepen- dently of its parameters (which can be values, types, or even other templates).

This chapter has been shortened extremely, as creating templates is not the purpose of the current projekt. The interested reader however, is referred to chapter 7 of this book. Please avoid creating your own template classes unless you have really good reasons to.

CHAPTER 8: STANDARD LIBRARY

The C++ Standard library consists of a number of headers, in which each header declares types, macros, and functions for use in C++ programs. The standard library has 51 headers, almost all the names declared in the standard library are in the std namespace. To use the standard library, you must #include the desired header or headers. Some implementations #include headers within other headers.

<algorithm>	**Standard algorithms for copying, searching, sorting, and otherwise operating on iterators and containers. See Chapter 10 for more information about the standard algorithms.**
<bitset>	**Class template to hold a fixed-sized sequence of bits.**
<cassert>	**Runtime assertion-checking; C header.**
<cctype>	**Character classification and case conversion; C header (see also <locale>).**
<cerrno>	**Error codes; C header.**
<cfloat>	**Limits of floating-point types; C header (see also <limits>).**

<ciso646>	**Empty header because C declarations are incorporated in the C++ language; C header.**
<climits>	**Limits of integer types; C header (see also <limits>).**
<clocale>	**Locale-specific information; C header (see also <locale>).**
<cmath>	**Mathematical functions; C header.**
<complex>	**Complex numbers.**
<csetjmp>.	**Nonlocal goto; C header**
<csignal>	**Asynchronous signals; C header.**
<cstdarg>	**Macros to help implement functions that take a variable number of argu-ments; C header.**
<cstddef>	**Miscellaneous standard definitions; C header.**
<cstdio>	**Standard input and output; C header (see also <iostream> and related headers).**
<cstdlib>	**Miscellaneous functions and related declarations; C header.**
<cstring>	**String-handling functions; C header (see also <string>).**
<ctime>	**Date and time functions and types; C header.**
<cwchar>	**Wide character functions, including I/O; C header (see also <locale> ,<iostream> , <string> , and other I/O-related headers).**
<cwctype>	**Wide character classification and case conversion; C header (see also<locale>).**
<deque>	**Deque (double-ended queue) standard container.**
<exception>	**Base exception class and functions related to exception-handling.**
<fstream>	**File-based stream I/O.**
<functional>	**Function objects; typically used with standard algorithms.**
<iomanip>	**I/O manipulators; used with standard I/O streams.**
<ios>	**Base class declarations for all I/O streams.**
<iosfwd>	**Forward declarations for I/O objects.**
<iostream>	**Declarations of standard I/O objects.**
<istream>.	**Input streams and input/output streams**
<iterator>	**Additional iterators for working with standard containers and algorithms. See Chapter 10 for more information.**
<limits>	**Limits of numerical types.**
<list>	**Standard linked list container.**
<locale>	**Locale-specific information for formatting and parsing numbers, dates, times,and currency values, plus character-related functions for classifying,converting, and comparing characters and strings.**
<map>	**Associative map (sometimes called a dictionary) standard container.**

<memory>	**Allocators, algorithms for uninitialized memory, and smart pointers (auto_ptr).**
<new>	**Global operator new and operator delete and other functions related to managing dynamic memory.**
<numeric>	**Numerical algorithms**
<ostream>	**Output streams.**
<queue>	**Queue and priority queue container adapters.**
<set>	**Associative set container.**
<sstream> .	**String-based I/O streams**
<stack>	**Stack container adapter.**
<stdexcept>	**Standard exception classes.**
<streambuf>	**Low-level stream buffers; used by high-level I/O streams.**
<string>	**Strings and wide-character strings.**
<strstream>	**String streams that work with character arrays (see also <sstream>).**
<typeinfo>	**Runtime type information.**
<utility>	**Miscellaneous templates, such as pair , most often used with standard containers and algorithms.**
<valarray>	**Numerical arrays.**
<vector>	**Vector (array-like) standard container.**

ALLOCATORS

An allocator is a policy class that defines an interface for managing dynamic memory. You already know about the new and delete expressions for allocating and freeing dynamic memory. They are simple, expressive, and useful, but the standard library does not necessarily use them internally. Instead, the standard library uses allocators, which let you provide alternative mechanisms for allocating and freeing memory. Please do not use custom allocators, unless you have a really good reason to.

NUMERICS

The C++ library has several headers that support numerical programming. The most basic header is <cmath> , which declares transcendental and other mathematical functions. This C header is expanded in C++ to declare overloaded versions of every function. For example, whereas C declares only exp(double) , C++ also declares exp(float) and exp(long double) . The <complex> header declares a class template for complex numbers.

```
// some examples on usage .. Math.PI --- abs() function --- anything else?
```

Many numerical programmers find the C++ standard library to be lacking. However, the Blitz++ project is a popular, high-performance numerical library. Boost also has some numerical headers, such as a full-featured rational class template.

CHAPTER 9: INPUT AND OUTPUT (I/O)

// picture here

The ios_base class declares types and constants that are used throughout the I/O library. Formatting flags, I/O state flags, open modes, and seek directions are all declared in ios_base . The basic_istream template declares input functions, and basic_ostream declares output functions. The basic_iostream template inherits input and output func-tions through multiple inheritance from basic_istream and basic_ostream . I/O to and from external files is handled by basic_fstream , basic_ifstream , and basic_ofstream , using basic_filebuf as the stream buffer. These class templates are declared in <fstream> .You can also treat a string as a stream using basic_istringstream , basic_ ostringstream , and basic_stringstream . The stream buffer template is basic_stringbuffer . These class templates are declared in <sstream> . For a complete list of C++ I/O headers (and a lot of more information on I/O), the interested reader is referred to page 234 of C++ in a Nutshell.

CHAPTER 10: CONTAINERS, ITERATORS, ALGORITHMS

Every major programming language has fundamental containers, such as arrays or lists. Modern programming languages usually have an assortment of more powerful containers, such as trees, for more specialized needs. The C++ library has a basic suite of containers (deques, lists, maps, sets, and vectors), but more important, it has generic algorithms, which are function templates

that implement common algorithms, such as searching and sorting. The algorithms operate on iterators, which are an abstraction of pointers that apply to any container or other sequence.

CONTAINERS

The fundamental purpose of a container is to store multiple objects in a single container object. Different kinds of containers have different characteristics, such as speed, size, and ease of use. The choice of container depends on the characteristics and behavior you require. In C++, the containers are implemented as class templates, so you can store anything in a container. (Well, almost anything. The type must have value semantics, which means it must behave as an ordinary value, such as an int . Values can be copied and assigned freely. An original and its copy must compare as equal. Some containers impose additional restrictions.)

STANDARD CONTAINERS

The standard containers fall into two categories: sequence and associative containers. A sequence container preserves the original order in which items were added to the container. An associative container keeps items in ascending order (you can define the order relation) to speed up searching. The standard containers are:

- deque

A deque (double-ended queue) is a sequence container that supports fast insertions and deletions at the beginning and end of the container. Inserting or deleting at any other position is slow, but indexing to any item is fast. Items are not stored contiguously. The header is <deque> .

- list

A list is a sequence container that supports rapid insertion or deletion at any position but does not support random access. Items are not stored contiguously. The header is <list> .

- Map/multimap

A map (or dictionary) is an associative container that stores pairs of keys and associated values. The keys determine the order of items in the container. Map

requires unique keys. multimap permits duplicate keys. The header for map and multimap is <map> .

- Set/multiset

A set is an associative container that stores keys in ascending order. Set requires unique keys. multiset permits duplicate keys. The header for set and multiset is <set> .

- Vector

A vector is a sequence container that is like an array, except that it can grow as needed. Items can be rapidly added or removed only at the end. At other positions, inserting and deleting items is slower. Items are stored contiguously. The header is <vector> .

- Container Adapters

In addition to the standard containers, the standard library has several container adapters. An adapter is a class template that uses a container for storage and provides a restricted interface when compared with the standard containers. The standard adapters are:

- priority_queue

A priority queue is organized so that the largest element is always the first. You can push an item onto the queue, examine the first element, or remove the first element. The header is <queue> .

- queue

A queue is a sequence of elements that lets you add elements at one end and remove them at the other end. This organization is commonly known as FIFO (first-in, first-out). The header is <queue> .

- stack

A stack is a sequence that lets you add and remove elements only at one end. This organization is commonly known as LIFO (last-in, first-out). The header is <stack> .

- Pseudo-Containers

The standard library has a few class templates that are similar to the standard containers but fail one or more of the requirements for a standard container:

- bitset

Represents a bitmask of arbitrary size. The size is fixed when the bitset is declared. There are no bitset iterators, so you cannot use a bitset with the standard algorithms. See <bitset> in Chapter 13 for details.

- basic_string/string/wstring

Represent character strings. The string class templates meet almost all of the requirements of a sequence container, and you can use their iterators with the standard algorithms. Nonetheless, they fall short of meeting all the requirements of a container, such as lacking front and back member functions. The header is <string>

- valarray

Represents an array of numeric values optimized for computational efficiency. A valarray lacks iterators, and as part of the optimization, the compiler is free to make assumptions that prevent valarray from being used with the standard algorithms. See <valarray> in Chapter 13 for details.

- vector<bool>

A specialization of the vector template. Although vector<> usually meets the requirements of a standard container, the vector<bool> specialization does not because you cannot obtain a pointer to an element of a vector<bool> object.

- iterator begin( )

const_iterator begin( ) const Returns an iterator that points to the first item of the container. Complexity is constant.

- void clear( )

Erases all the items in the container. Complexity is linear.

- bool empty( ) const

Returns true if the container is empty (size( ) == 0). Complexity is constant.

- iterator end( )

const_iterator end( ) const Returns an iterator that points to one past the last item of the container. Complexity is constant. (See Iterators later in this chapter for a discussion of what “one past the last item” means.)

- erase(iterator p)

Erases the item that p points to. For a sequence container, erase returns an iterator that points to the item that comes immediately after the deleted item or end( ) . Complexity depends on the container. For an associative container, erase does not return a value. Complexity is constant (amortized over many calls).

- erase(iterator first, iterator last)

Erases all the items in the range [first , last). For a sequence container, erase returns an iterator that points to the item that comes immediately after the last deleted item or end( ) . Complexity depends on the container.

For an associative container, erase does not return a value. Complexity is logarithmic, plus last “ first

- size_type max_size( ) const

Returns the largest number of items the container can possibly hold. Although many containers are not constrained, except by available memory and the limits of size_type , other container types, such as an array type, might have a fixed maximum size. Complexity is usually constant.

- container& operator=(const container& that)

Erases all items in this container and copies all the items from that .Complexity is linear in size( ) + that.size( ) .

- size_type size( ) const

Returns the number of items in the container. Complexity is usually constant.

- void swap(const container& that)

Swaps the elements of this container with that . An associative container also swaps the comparison function or functions. Complexity is usually constant. A container that supports bidirectional iterators should define rbegin and rend member functions to return reverse iterators.

The following functions are optional. The standard containers provide only those functions that have constant complexity.

- reference at(size_type n) /const_reference at(size_type n) const

Returns the item at index n , or throws out_of_range if n >= size( ) .

- reference back( )/const_reference back( ) const

Returns the last item in the container. Behavior is undefined if the container is empty.

- reference front( )/const_reference front( ) const

Returns the first item in the container. Behavior is undefined if the container is empty.

- reference operator[](size_type n)/const_reference operator[](size_type n)

Returns the item at index n . Behavior is undefined if n >= size( ) .

- void pop_back( )

Erases the last item in the container. Behavior is undefined if the container is empty.

- void pop_front( )

Erases the first item in the container. Behavior is undefined if the container is empty.

- void push_back(const value_type& x)

Inserts x as the new last item in the container.

- void push_front(const value_type& x)

Inserts x as the new first item in the container.

A sequence container should define the following member functions. The complexity of each depends on the container type.

- iterator insert(iterator p, const value_type& x)

Inserts x immediately before p and returns an iterator that points to x .

- void insert(iterator p, size_type n, const value_type& x)

Inserts n copies of x before p .

- template<InIter>

void insert(iterator p, InIter first, InIter last)

Copies the values from [first , last) and inserts them before p .

An according list for associative containers in contrast to the sequence containers can be found on page 253 of the book.

ITERATORS

An iterator is an abstraction of a pointer used for pointing into containers and other sequences. An ordinary pointer can point to different elements in an array. The ++ operator advances the pointer to the next element, and the * operator dereferences the pointer to return a value from the array. Iterators generalize the concept so that the same operators have the same behavior for any container, even trees and lists.

// already some code here?

ITERATOR SAFETY

The most important point to remember about iterators is that they are inherently unsafe. Like pointers, an iterator can point to a container that has been destroyed or to an element that has been erased. You can advance an iterator past the end of the container the same way a pointer can point past the end of an array. With a little care and caution, however, iterators are safe to use. Even a valid iterator can become invalid and therefore unsafe to use, for example if the item to which the iterator points is erased. Iterators for the array-based containers (deque , vector) become invalid when the underlying array is reallocated, which might happen for any insertion and for some

erasures. Iterators can also be used with I/O streams, the interested reader is again referred to the book (page 262f).

ALGORITHMS

The so-called algorithms in the standard library distinguish C++ from other programming languages. Every major programming language has a set of containers, but in the traditional object-oriented approach, each container defines the operations that it permits, e.g., sorting, searching, and modifying. C++ turns object-oriented programming on its head and provides a set of function templates, called algorithms, that work with iterators, and therefore with almost any container.

The advantage of the C++ approach is that the library can contain a rich set of algorithms, and each algorithm can be written once and work with (almost) any kind of container. And when you define a custom container, it automatically works with the standard algorithms (assuming you implemented the containers iterators correctly). The set of algorithms is easily extensible without touching the container classes. Another benefit is that the algorithms work with iterators, not containers, so even non-container iterators (such as the stream iterators) can participate.

C++ algorithms have one disadvantage, however. Remember that iterators, like pointers, can be unsafe. Algorithms use iterators, and therefore are equally unsafe. Pass the wrong iterator to an algorithm, and the algorithm cannot detect the error and produces undefined behavior. Fortunately, most uses of algorithms make it easy to avoid programming errors. Most of the standard algorithms are declared in the <algorithm> header, with some numerical algorithms in <numeric> .

HOW ALGORITHMS WORK

The generic algorithms all work in a similar fashion. They are all function templates, and most have one or more iterators as template parameters. Because the algorithms are templates, you can instantiate the function with any template arguments that meet the basic requirements. For example, for_each is declared as follows:

```
template<typename InIter, typename Function>
```

```
Function for_each(InIter first, InIter last, Function func);
```

Whenever you need to process the contents of a container, you should think about how the standard algorithms can help you. For example, suppose you need to read a stream of numbers into a data array. Typically, you would set up a while loop to read the input stream and, for each number read, append the number to the array. Now rethink the problem in terms of an algorithmic solution. What you are actually doing is copying data from an input stream to an array, so you could use the copy algorithm.

NON-MODIFYING OPERATIONS

The following algorithms examine every element of a sequence without modifying the order:

• count	Returns the number of items that match a given value
• count_if	Returns the number of items for which a predicate returns true for_each. Applies a function or functor to each item
• Comparison	The following algorithms compare objects or sequences (without modifying the elements):
• equal	Determines whether two ranges have equivalent contents lexicographical_compare. Determines whether one range is considered less than another range
• max	Returns the maximum of two values
• max_element	Finds the maximum value in a range
• min	Returns the minimum of two values
• min_element	Finds the minimum value in a range
• mismatch	Finds the first position where two ranges differ
• Searching	The following algorithms search for a value or a subsequence in a sequence (without modifying the elements):
• adjacent_find	Finds the first position where an item is equal to its

	neighbor
• find	Finds the first occurrence of a value in a range
• find_end	Finds the last occurrence of a subsequence in a range
• find_first_of	Finds the first position where a value matches any one item from a range of values
• find_if	Finds the first position where a predicate returns true
• Search /search_n	Finds a subsequence in a range
• Binary search	The following algorithms apply a binary search to a sorted sequence. The sequence typically comes from a sequence container in which you have already sorted the elements. You can use an associative containers, but they provide the last three functions as member functions, which might result in better performance.
• binary_search	Finds an item in a sorted range using a binary search
• equal_range	Finds the upper and lower bounds
• lower_bound	Finds the lower bound of where an item belongs in a sorted range
• upper_bound	Finds the upper bound of where an item belongs in a sorted range

Modifying sequence operations: The following algorithms modify a sequence:

• copy	Copies an input range to an output range
• copy_backward	Copies an input range to an output range, starting at the end of the output range
• Fill /fill_n	Fills a range with a value
• Generate/ generate_n	Fills a range with values returned from a function
• iter_swap	Swaps the values that two iterators point to

• random_shuffle	Shuffles a range into random order
• remove	Reorders a range to prepare to erase all elements equal to a given value
• remove_copy	Copies a range, removing all items equal to a given value
• remove_copy_if	Copies a range, removing all items for which a predicate returns true
• remove_if	Reorders a range to prepare to erase all items for which a predicate returns true
• replace	Replaces items of a given value with a new value
• replace_copy	Copies a range, replacing items of a given value with a new value
• replace_copy_if	Copies a range, replacing items for which a predicate returns true with a new value
• replace_if	Replaces items for which a predicate returns true with a new value
• reverse	Reverses a range in place
• reverse_copy	Copies a range in reverse order
• rotate	Rotates items from one end of a range to the other end
• rotate_copy	Copies a range, rotating items from one end to the other
• swap_ranges	Swaps values in two ranges
• transform	Modifies every value in a range by applying a transformation function
• Unique	Reorders a range to prepare to erase all adjacent, duplicate items
• unique_copy	Copies a range, removing adjacent, duplicate items

Sorting: The following algorithms are related to sorting and partitioning. You can supply a comparison function or functor or rely on the default, which uses the < operator.

• nth_element	Finds the item that belongs at the nth position (if the range were sorted) and reorders the range to partition it into items less than the nth item and items greater than or equal to the nth item.
• partial_sort	Reorders a range so the first part is sorted.
• partial_sort_copy	Copies a range so the first part is sorted.
• partition	Reorders a range so that all items for which a predicate is true come before all items for which the predicate is false.
• sort	Sorts items in ascending order.
• stable_partition	Reorders a range so that all items for which a predicate is true come before all items for which the predicate is false. The relative order of items within a partition is maintained.
• stable_sort	Sorts items in ascending order. The relative order of equal items is maintained.

Merging: The following algorithms merge two sorted sequences:

• inplace_merge	Merges two sorted, consecutive subranges in place, so the results replace the original ranges merge
TODO	Merges two sorted ranges, copying the results to a separate range

Set operations: The following algorithms apply standard set operations to sorted sequences:

• includes	Determines whether one sorted range is a subset of another
• set_difference	Copies the set difference of two sorted ranges to an

	output range
• set_intersection	Copies the intersection of two sorted ranges to an output range
• set_symmetric_difference	Copies the symmetric difference of two sorted ranges to an output range
• set_union	Copies the union of two sorted ranges to an output range

Heap operations: The following algorithms treat a sequence as a heap data structure:

• make_heap	Reorders a range into heap order
• pop_heap	Reorders a range to remove the first item in the heap
• push_heap	Reorders a range to add the last item to the heap
• sort_heap	Reorders a range that starts in heap order into fully sorted order

Permutations: The following reorder the elements of a sequence to generate permutations:

• next_permutation	Reorders a range to form the next permutation
• prev_permutation	Reorders a range to form the previous permutation

CHAPTER 11: PREPROCESSOR REFERENCE

The preprocessor handles preprocessing directives, which can define and undefine macros, establish regions of conditional compilation, include other source files, and control the compilation process somewhat. A macro is a name that represents other text, called the macro replacement text. When the macro name is seen in the source file, the preprocessor replaces the name with the replacement text. A macro can have formal parameters, and actual arguments are substituted in the expansion.

```
#define PI 3.1415

#define RAD 180/PI
```

we will use preprocessor macros primarily for include guards, e.g.

```
#ifndef BLABLA_H

#define BLABLA_H

// some code in between, e.g. class and member declarations

#endif

// we could try defining cudaMain instead of forward declarations ...
```

PREDEFINED MACROS

The following macros are predefined. Do not undefine or redefine any of the predefined macros.

```
_ _cplusplus
```

Has the value 199711L . Future versions of the C++ standard will use a larger value. Nonconforming compilers should use a different value.

```
_ _DATE_ _
```

Expands to the date of compilation, as a string literal, in the form "Mmm dd yyyy" , in which dd begins with a space for numbers less than 10. An implementation is free to substitute a different date, but the form is always the same, and the date is always valid.

```
_ _FILE_ _
```

Expands to the name, as a string literal, of the source file being compiled.

```
_ _LINE_ _
```

Expands to the line number, as a decimal constant, of the source file being compiled.

```
_ _STDC_ _
```

Is implementation-defined. C++ compilers might define this macro; if it is defined, the value is implementation-defined. Note that C compilers are required to define _ _STDC_ _ as 1 , and in some implementations, the same preprocessor might be used for C and C++.

```
_ _TIME_ _
```

Expands to the compilation time, as a string literal, in the form "hh:mm:ss" . An implementation is free to substitute a different time, but the form is always thesame, and the time is always valid.

```
#include "header.h"     // cwd
```

vs

```
#include <header>       // searches only in system libraries
```