C++

Ming-Hwa Wang, Ph.D.
Department of Computer Engineering
Santa Clara University

Program

- file name, file suffix (.cpp, .o, .a)
- main function and optional subroutine functions
- preprocessor directives
  - conditional directives: #ifdef, #ifndef, #if, #else, #endif
    - use #ifdef __cplusplus to mix C and C++ program code
  - #include
    - predefined/standard header files: using #include <header.h>
    - user-specified header files: using #include "header.h", default in local directory or use -I compiling options to modify the include path
    - can be nested, use conditional directives to guard against multiple processing of a header file
  - #define
    - using constant and inline instead of #define
- linkage directives only at file scope (prefer in header file): extern “C” followed by single line or compound syntactic form for C++ program to call C
- global/static variable definitions
- extern variable/function declarations
- comments
  - comment pair '/* */': no nest, usually for multiple-line comment
  - comment delimiter '//', usually for single-line comment

Compiler

- translate C++ into assembly language using CC/g++
- syntax checking (syntax errors, type errors)
- precompiled functions using archive library: "CC -c foo.c" then "ar cr foo.a foo.o", and used by: "CC main.c foo.a"
- link with library, e.g., -lm for math library

I/O Library:

- strong typed iostream library (defined in iostream.h)
  - low level I/O: a sequence, or stream, of bytes, no notion of a data type
  - high/user level I/O: a sequence of possibly intermixed data types
- standard input cin, standard output cout, standard error cerr for unbuffered output and clog for buffered output
- ostream output/insertion operator: <<
  - newline character '\n'
  - endl inserts '\n' into output stream and flushes the buffer
  - pointer types will be displayed as the objects' address in hex (or decimal if explicit cast by long), but the type char* is interpreted as a string (to print address of string, cast it with void*)
  - successive occurrences of output operator can be concatenated
- put( ), write( ):
- istream input/extraction operator: >> (suppress white spaces: blanks, tabs, and newlines, even embedded with double quotes)
  - a null character is appended to a string during input, so the size is length of string + 1
  - use setw( ) in iomanip.h to prevent overflow input
- get( ), getline( ), and read( ): no suppress white spaces
  - gcount( ) returns the number of characters actually extracted by the last getline( ) or read( )
  - use int stead of char in get( )
  - end-of-file EOF (-1) in iostream.h
  - putback( ), peek( ), ignore( )
- overloading operator<<: a class need overload friend ostream operator<<(ostream&, <class type>)
- overloading operator>>: a class need overload friend istream operator>>(istream&, <class type>)
- iostream error bit-vector states: rdstate( ) returns the vector, clear( ) reset entire state to 0, e.g., is.clear(ios::badbit) sets badbit and reset all others to 0, is.clear(ios::badbit|is.rdstate( )) retains previous error in badbit, while (is && (ch = is.get( )) != lbrace) to prevent infinite loop
- file I/O in fstream.h: ifstream, ofstream, fstream
  - ifstream with file name and mode input(ios::in)
  - ofstream with file name and mode either output(ios::out) or append(ios::app)
  - an ifstream/ofstream object can be defined without specifying a file and later explicitly open( ), close( )
  - repositioned by seekg( ) to move to an absolute address or seekp( ) to move a byte offset from a particular position, the first argument streampos can be positive or negative (move backward), the second argument seek_dir can be ios::beg (the default), ios::cur, or ios::end, the current position can be get from tellg( )
  - eof( ), bad( ), fail( ), good( )
  - to test if file is open: if (f.rdbuf( )->is_open == 0)
- string/incore I/O: istrstream, ostrstream
  - str( ) returns a pointer to the character array associated to the ostrstream, it will freeze the array, thus only invoked after insertion is complete and need explicit delete
  - << ends to inserts a null character
- format state
  - using setf( ) and unsetf( ), the first argument is format bit flag: ios::showbase, ios::showpoints, ios::dec, ios::oct, ios::hex, ios::fixed, ios::scientific, and the second argument is format bit field: ios:basefield, ios::floatfield
  - using integer manipulators: hex, oct, dec (the default)

- using float manipulators: setprecision( ), default is 6, and get current precision by precision( )

Exception Handling

- blindly trust without test
- explicit test
- assert( ) macro: #include <assert.h>
- throw <exception>: transfers control from the point of the program anomaly to an exception handler
- try { .. } catch (<exception>)
  - the appropriate catch handler is determined in the order (no priorities) of their appearance by matching on the type of the thrown exception, if no match, terminate( ) is called, or use catch (…) to catch all exceptions
  - an empty throw expression can only occur within a catch handler, and simply passes the exception up to the enclosing try block
- the function throw list (empty list means no exception will be thrown)

Data Types

- bit, byte, word
- meaningful interpretation of fixed-length bit sequences at a particular address
- determine the amount of storage allocated to the variable and the set of operations that can be performed on the variable
- strongly typed language: all initializations and assignments are checked at compile time
- type conversions: implicitly (promotion or demotion by compiler) and explicitly (cast by programmer for safe, efficiency, disambiguation)
  - unsafe type conversion: from a wider data type to a narrow data type
- predefined data types
  - char (one byte)
  - int (one word): short, long, signed, unsigned
  - float, double, long double
- user-defined data types
- literal constants (non-addressable)
  - integral type constant: decimal, octal (lead with 0), hexadecimal (lead with 0x), long (end with L or l), unsigned (end with U or u)
  - real type constant: double precision exponent (using E or e), single precision (using F or f), extended long precision (using L or l)
  - char constant: using single quotation mark
    - escape sequences: \n, \t, \v, \b, \r, \f, \a, \\, \?, \', \", \ooo
  - string constant: using double quotation mark and \ for continuation
- variables
  - symbolic variables (addressable) must be defined/declared before used
  - definition only once (memory allocated), declaration can be multiple
  - lvalue: location/address value (assign)
  - rvalue: data value (read)
  - variable name: case sensitive identifier [A-Za-z_][A-Za-z_0-9]*
    - reserved keywords can't be used as id, id normally lower case, id for mnemonic name, use _ or upper first char to compose multiword id
- enumeration types
  - an enumeration type (with an optional tag name) declares a set of symbolic integral constants
  - the use of an enumeration tag name provides a helpful form of program documentation
  - by default, the first enumerator is assigned the value 0, each subsequent enumerator is assigned a value on greater than the value of the enumerator that immediately precedes it. A unique value may also be explicitly assigned to an enumerator
- pointer types
  - a pointer variable holds values that are the addresses of objects in memory
  - through a pointer, an object can be referenced indirectly
  - every pointer has an associated type, which specify the type of the object the pointer will address
  - the address-of operator &
  - the dereference operator *
  - pointer arithmetic
  - string pointer: char *, use string library functions in string.h
  - null pointer: NULL or 0
  - generic pointer void* maybe assigned the address of an object of any data type, but may not be dereferenced directly
  - applications: create linked list, manage objects allocation during execution
- reference types
  - a reference type (&<id>) serves as an alternative name (or alias) for the object with which it has been initialized.
  - a reference must be initialized and, once initialized, a reference cannot be made to alias another object (but no need to declare it explicitly as const, it is implied)
  - only a reference to a constant object or pointer can be initialized either with an rvalue, or lvalue not of its exact type
- constant types
  - to avoid using hard-coded or magic numbers for readability, locality and maintainability
  - constant modifier transforms a symbolic variable into a symbolic constant (read only, not addressable, and thus has to be initialized during definition)
  - pointers to constant objects (const char *pc), constant pointers(char *constant pc), and constant pointers point to constant objects(const char *constant char)
- array types
  - an array type is a collection of objects (elements) of a single data type, the individual objects is accessed by it position in the array (indexing or subscripting)

- a dimension specifies the constant number of elements contained in the array using bracket pair. The elements of an array are numbered beginning with 0
- explicitly initialized by comma-separated values enclosed by braces, no need specify a dimension, or specify a dimension >= initialized values
- to copy one array into another, each element must be copied in turn
- no range checking is provided by the compiler, user should check himself
- multidimensional array: row major order
- the array id (equivalent to a constant pointer, thus can't be changed) evaluate to the address in memory of the first element, and the pointer and array notation are equivalent

- class types
  - a user-defined data type, an aggregate of named data elements (may different types), and a set of operations to manipulate that data
  - class head: the keyword class and a tag name (a new data type)
  - class body: contains member definitions enclosed by braces and terminated by semicolon
  - control access to the class member: public, protected, private
  - information hiding
  - this is a pointer through which the programmer can access the class object invoking the member function
  - member functions can access its own class members directly
    - constructor: use the class tag name as the function name
      - default constructor (no argument)
      - member initialization list
    - destructor: use tag name preceded by ~
  - the scope operator ::
  - the new operator for free store or dynamic (run-time) memory allocation
    - either a single object or an array of objects can be allocated (uninitialized), after the memory for a class object is allocated, a class constructor, if defined, is automatically invoked to initialize that memory
  - the delete operator deinitializes or returns the allocated memory
    - to delete an array of class objects, use delete [ ]
  - the member selector operators: dot operator . and arrow operator ->
  - inheritance to define subtype relationships, need define only those aspects of its implementation that are different or in addition to (constructors are class-specific initialization functions and hence not inherited)
    - the colon operator : defines to be derived from
    - the base class, is the class derived from, can be assigned objects of a derived class
    - virtual class member functions are inherited members whose implementation is dependent on its class type at run time (can't be inlined)
  - template class: by parameterizing the types (in angle brackets) which need to change with each class instance
- typedef provides a general facility for introducing mnemonic synonyms for existing predefined, derived, and user-defined data types, it doesn't create a new type
- volatile objects using an additional modifier volatile (default is int), and their values can possibly be changed in ways outside either the control or detection of the compiler, thus compiler will not aggressively optimize codes containing the object

Expressions

- an expression contains one or more operations (operators and operands)
- the evaluation of an expression performs operations and yields a rvalue
- operator: unary, binary, ternary, left, right
- precedence and associativity (use parenthesis to override)
- arithmetic operators: + - * / %
- equality operators: == !=
- relational operators: < <= > >=
- logical operators: && || !
- bitwise operators (on unsigned operands for portability): ~ << >> & ^ |
- increment ++ and decrement -- can be prefix or postfix
- sizeof operator returns the size in bytes, application the sizeof operator on a pointer type returns the size of the memory necessary to contain an address of that type, but application the sizeof operator on a reference type returns the size of the referenced object
- ternary (arithmetic if) operator
- comma expression returns the value of the right-most expression
- return statement

Statements

- default flow of statement execution is sequential
- the empty or null statement ;
- declaration statement can be specified outside function
  - extern <type> id [(<type>...)] [, id [(<type>...)]];
  - definition statement
  - [static|constant] <type> id [<initialization>] [,id [<initialization>]];
  - initialization: explicitly = <value>, implicitly (<value>)
- expression statement: expression followed by a semicolon
- assignment statement: assignment operator = can be concatenated, compound assignment operators op=
- compound statement (block)
- conditional statement: if if-else (match the else with the last occurring unmatched if to resolve dangling-else ambiguity, or override by blocks), good coding style always using compound statement braces to avoid possible confusion and error
- switch statement chooses among a set of mutually exclusive choices

    - switch case default break fall-through
- loop (iterative control) statement: while (test before execute), do (test after execute), for (step through a fixed-length data structure)
- jump statement: break (for loop and switch) continue (for loop) goto (only in same function, label must followed by a statement, can't jump forward over a variable definition with an explicit or implicit initializer

Functions

- benefits: reusability, readability, only change one localized instance, full type checking its arguments and return value
- function must be declared before used, use forward declaration
- function prototypes are best placed within header files
    - return type (returns exact one scalar value with default int or void for not return anything), function name, argument type list
- function public interfaces: return type and argument list (signature)
- parameter pass positionally: pass-by-value (copy the rvalues of the actual arguments into the storage of the formal arguments, unsuitable for pass a large class object or need to modify the argument, can pass pointers to solve the problem), pass-by-reference
    - the more arguments in a function the more complex it is (divide into subfunctions)
    - whenever a reference or pointer argument is not intended to be modified within the function, declare it as const
    - a pointer argument can also be declared as a reference, thus allow modify the pointer itself rather than the object addressed by the pointer
    - an array is passed by as a pointer to its zeroth element, the array's size is not relevant to the declaration of the formal argument, a multidimensional array formal argument must specify the size of all its dimensions beyond that of its first
- default initializers at end within signature: frees programmer to attend very small detail. If an argument is provided, it overrides the default value. An argument with default initializer can be specified only once in a file (better in header file.)
- variable number of parameters: using ellipses ... to suspend type checking
- function body contains statements
- use void and dummy instances to do incremental program building
- C++ scopes: file scope (the outermost scope), local scope (can be nested for function or block), class scope
- if multiple values need to be returned: global variables (simplicity but nonintuitive and may have side effect especially in recursion), return pointer to or reference of array/object, pointer/reference formal arguments
- global variables will be automatically initialized to 0 if there is no explicit initializer. local variables will be undefined if not initialized
- a name must be unique in one scope and may be used in distinct scopes, a variable is visible to the program only from within its scope, a local variable may reuse a global one and cause the global one hidden (or use scope operator to specify the scope), a global variable is defined in one file and other files need extern to use it (extern variable with initializer is definition instead of declaration), static global variables are intended to refer to different program entities
- static identifiers are internal linkage (inline functions and const definitions have internal linkage) and persist across invocations, and nonstatic global variable are external linkage
- register local variables for heavily used inner loop index/pointer, a formal argument can be declared as a register variable. The compiler will try as much as possible to put registered variables in register for speed
- function is invoked at run-time (managed by the run-time stack) except inline function (the body is expanded during compilation) for efficiency on small and frequently called functions
- the value returned by a function is also pass-by-value, thus do not return a reference to a local object (dangling reference), and any modification of the returned value changes the actual object being returned (use const return value to prevent modification)
- recursive function: a function that calls itself, either directly or indirectly. recursive function is slower and difficult to be inlined completely but smaller and less complex
- overloading function: the name and signature of a function uniquely identify it
    - function name overloading allows multiple function instances that provide a common operation on different argument types to share a common name, without overloading, each instance must be given its own unique name and separated implementation
    - the meaning remains invariant over a set of instances, each of which is implemented in a different way (transparent to the user)
    - when a function name is declared more than once, if both return type and signature match exactly, the second one is redeclaration of the first. If same signature but different return type, it is an error. If signatures are different, the function name is overloaded
    - only when making it more difficult to understand, or when you can compressed multiple instances of a function into a single one using default arguments, do not overloading
    - a call to an overloaded function is resolved through argument matching, it can be a match, no match, or ambiguous
    - matching can achieved in the following order of precedence:
        - an exact match, argument matching can distinguish between constant and nonconstant pointer and reference arguments, e.g., const char* and char* (const is not meaningful when applied to either an object, e.g., const int and int, or a constant pointer, e.g., int *const and int *), as well as between volatile and nonvolatile arguments, an exact match can be overridden by the use of an explicit cast, trivial conversions are given precedence over all other conversions
        - a match through promotion (char $\rightarrow$ int, unsigned char $\rightarrow$ int, short $\rightarrow$ int, unsigned short $\rightarrow$ int or unsigned int, float $\rightarrow$ double, enumeration $\rightarrow$ int) but not demotion

    - a match through of a standard conversion, no one standard conversion is given precedence over another, ambiguity can be resolved by an explicit cast
    - a match through application of a user-defined conversions
  - a call with multiple arguments is resolved by applying the matching rules to each argument in turn, the intersection rule: the function chosen is the one for which the resolution of each argument is the same or better than for all other functions in the overloaded set, and it is strict better than all other functions for at least one argument
  - the overloaded set of function instances associated with a particular name must all be declared within the same scope
  - operator new can be overloaded by the programmer with the prototype: void *operator new(size_t size);
- template function: provide an algorithm for the automatic generation of particular instances of that function varying by type
  - the implementation remains invariant over a set of instances, each of which handles a unique data type, strong-typing requires implementing an instance for each type
  - the macro expansion facility of preprocessor is dangerous, simple text substitution mechanism behaves unexpected under complex calls even with extra parenthesis (e.g., auto increment/decrement may be evaluated twice in simple max/min macro)
  - template instantiation or type substitution, the determination of the actual type to which to bind is made by an evaluation of the actual argument, but not the return type, when a specialized instance of a template function exists, it overrides the template in that particular type
  - it is not always possible to determine the legality of a template function until it is instantiated
  - template functions have NO nontrivial type conversions or promotions performed
  - a template function can be declared extern, inline, or static, the specifier is placed following the formal, a template function can be overloaded, a dummy argument indicating the type of return value:

    ```
    template <class T1, class T2, class RT>
    RT sum(T1, T2, RT dummy);
    ```
- pointers to functions can be passed as arguments to gain flexibility (e.g., sort or search function pointers)
  - a function name, when not modified by the call operator, or applying the address-of operator to the function name, evaluates as a pointer to a function of its type, an initialization or assignment is legal only if the argument list and return type matches
  - pointers can address instances of an overloaded function, the compiler resolves the instance by finding an exact match, pointer can address instances of a function template
  - the deference operator is not required in order to invoke a function through a pointer
  - array of pointer to functions and pointer to an array of pointers to functions
  - the use of typedef name can make the use of pointer to a function easier to read, e.g.,

    ```
    typedef int (*PFI) (int*, int);
    PFI ff(int);
    ```
  - when new fails, it test _new_handler (default 0) to see whether it points to a function, if no, new returns 0, otherwise, the function is invoked, _new_handler can be assigned directly or through the set_new_handler library function

Class

- struct A { … } is the same as class A { public : … }
- the C++ class mechanism allows users to define their own data types, a class with a private representation and a public set of operations is an abstract data type
- a C++ class has 4 associated attributes:
  - data members: the representation of the class, no explicit initializer is allowed, declare the data members in order of increasing size for optimal alignment, a forward declaration permits pointers and references to objects
  - member functions: the class interfaces, or operations, member functions defined in class body are automatically handled as inline functions, member functions defined outside of the class body must explicitly declared inline in order to be inlined, member functions can overloaded only other member functions of its class
  - levels of program access for information hiding and encapsulation: private (the default, only member functions or friends of its class can access), protected (only accessible through inheritance), public (accessible from everywhere)
  - class tag name or type specifier
- class declaration and class definition (space allocated)
- class member functions:
  - manager functions manage class objects and handling activities (initialization, assignment, memory management, type conversion), usually invoked implicitly by the compiler, e.g., constructor
  - implementor functions: provide the capabilities associated with the class abstraction
  - helping functions: provide support for the other class member functions, generally declared as private
  - access functions: support user access to otherwise private data
- member functions can be specified as const/volatile, only the destructor, constructors, and const/volatile member functions of a class can be invoked for a const/volatile class object
- constructor: a mechanism for the automatic initialization of class object, a user-supplied initialization function without a return type that is named with the tag name of its class, a default constructor is a constructor taking no argument, it may be overloaded, the execution of a constructor consists of initialization and assignment, arguments passed to member class constructors through the member initialization list can be used to initialize const and reference class data member, storage

allocation can be done locally on the run-time stack (disappear when the block it is defined in terminates) or dynamically from the free store (it be deleted before the class object goes out of scope), pointer assignment across scope is potentially dangerous, the member class constructors are always executed before the constructor for the containing class, the constructors are invoked in the order of 1) each base class constructor in the order of base class are declared within the derivation list, 2) each member class object constructor in the order of member class declarations, and 3) the derived class constructor

- memberwise initialization is a special constructor X::X(const X&), it copies each build-in or derived data member from one class object to another, but not pointer members (the class designer needs provide an explicit memberwise initialization constructor instance)
- destructor: deinitializes the class object prior to the normal deallocation of storage that occurs when an object goes out of scope or delete an object pointer, it is the tag name of its class prefixed by a tilde, it has no return type, takes no argument and may not be overloaded, the member class destructors are always executed after the destructor for the containing class, the destructors are invoked in the reverse order of the invocation of the constructors
- class array: an array of class object, use the subscript operator [ ]
- the reserved keyword “this” represents a pointer holding the address of the invoking class object, it supports the concatenation of member calls
- the friend mechanism gives nonmembers of the class access to the nonpublic member of a class, a class must specify each instance of an overloaded function it wish to make a friend to the class, can declare an entire class as friend to another class
- operator overloading
  - only the predefined set of C++ operators can be overloaded, the precedence of the operators cannot be overridden (has to be overridden by parenthesis), the predefined arity of the operator must be preserved, default arguments for an operator function are illegal
  - a member function requires that the left operand be an object of its class, if the left operand is another type, it must be made a nonmember function, if it requires access to nonpublic members, it must be a friend, symmetric binary operators are best defined as nonmembers
  - the subscript operator [ ] must be able to appear on both rhs and lhs, so its return value is a reference
  - the iterator operator ( ) returns a next element, or 0 if there is no next element
  - operator new and delete as static members in a class to overridden the default new and delete
  - the member selection operator -> is often referred to as that of a smart pointer
  - increment ++ and decrement -- operators
  - user-defined conversions: (only used judiciously)
    - a constructor that takes a single argument serves as a conversion operator between that argument type and the class, the invocation of conversion operator applied only if no other conversion is possible
    - a conversion operator: operator <type> ( ), must be a member function without return type and no argument, if required type does not match the conversion operator but only can be reached through a standard conversion, then the conversion operator is invoked
  - ambiguity may arise in connection with the implicit invocation of conversion operators, if two conversion operators are possible, the exact match is chosen before requires a standard conversion, if both need conversion, it can be resolved by explicit conversion, if two class define conversions between themselves, using cast or class conversion operator explicitly
- a static class member provides one variable for all the objects of one class rather than having each object maintain its own copy, it acts as a global variable for its class (information hiding enforced), static members can be accessed even no class object are defined, only one initialization of a static member can occur in a file together with the definitions of the noninline member functions (but not the class header file), it can be access directly through class scope operator ::, a static member function can only access static members but no “this” pointer
- pointers to class members
  - a pointer to a function may not legally be assigned the address of a member function even when the return type and signature of the two match exactly, because a member function requires class type
  - a pointer to class member can be invoked only when bound to an object or pointer of that class or of a class derived from it, and there is no implicit conversion of a pointer to member to a pointer of type void
  - the declaration of a pointer to a static class member looks the same as that of a pointer to a nonclass member, dereferencing the pointer does not require a class object
  - us of a typedef can make the pointer to member syntax easier to read
- class scope: every class maintain its own associated scope, the names of class members are said to be local to the scope of their class, a member function occurring within the scope of its class maintains its own local scope (the hidden class member can be access through the class scope operator ::)
- nested class: a class occurring at class scope, the enclosing class has no special access privileges with regard to the classes nested within it, nor does the nested class have any special access privileges to the nonstatic members of its enclosing class (use friend to do it), use extended class scope operator for defining nested class member functions or static data members outside of the class body

- local class: a class occurring at local scope of a function, a local class is not permitted to declare static data members, the enclosing function has no special access privileges to the nonpublic members of the local class
- unions: a space-saving class, the amount of storage allocated for a union is the amount necessary to contain its largest data member, only one member at a time may be assigned a value, a union cannot declare a static data member, nor can it declare a member as an object of a class that defines either a constructor or destructor, use a discriminant to keep track of the type of the value currently stored in the union, the data members of an anonymous union can be accessed directly, an anonymous union cannot have private or protected members, nor can it define member functions, an anonymous union defined at file scope must be declared static
- bit field: a space-saving member, the address-of operator (&) cannot be applied to a bit field, there can be no pointers to class bit field, nor can a bit field be declared to be static
- class template
  - template class declaration
    template <class T1, int I2, ..> class C1;
  - template class instantiation
  - template class specialization, an explicit implementation for a particular type
  - template class static members, memory is allocated upon each instantiation of the class, access of a static member of a template class is always through a particular instantiation
  - a template class can be serve either as an explicit base class, or as a derived class with a nontemplate base class, or as both
- object-oriented programming shifts the burden of type resolution from the programmer to the compiler using dynamic binding (virtual member functions), not only reduce the complexity and size of code, but also make for extensible code
  - single inheritance
  - multiple inheritance

```
class A {                                A   A
};                                       |   |
class B : public A {                     B   C
};                                        \ /
class C : public A {                       D
};
class D : public B, public C {
};
```

  - Diamond-shaped inheritance: a virtual base class is needed if the base class cannot be replicated.

```
class A {                                  A
};                                        / \
class B : public virtual A {             B   C
};                                        \ /
class C : virtual public A {               D
};
class D : public B, public C {
};
```

  - the abstract superclass
  - derivation/inheritance hierarchy or graph
- abstract base class is an incomplete specification and can't be instantiated
- default is private, members intended to be inherited but not intended to be public are declared as protected members
- A base class can be declared private (the default), protected, or public. Public derivation makes the derived class a subtype of its base, and this is the most common form of derivation. Protected and private derivations are used to represent implementation details. Protected bases 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 base so that stronger guarantees can be provided. Normally, public bases for interfaces, and protected bases for implementations
- a derived class inherits all the member functions of each of its base classes except the constructors, destructors, and assignment operators
- reuse of an inherited member's name within the derived class hides the inherited member, use class scope operator to access base class members
- inheritance is an is-a relationship and composition is a has-a relationship (implemented by making one class a member of the other class)
- safe or standard conversions: a derived class object/reference/pointer will be implicitly converted into a public base class object/reference/pointer, a pointer to a class member of a base class will be implicitly converted into a pointer to a class member of a publicly derived class and a pointer to any class object will be implicitly converted into a pointer of type void*
- unsafe conversion need explicit casting
- overloaded functions with class argument, the exact matching of a class argument, matching achieved by application of a standard conversion, and matching achieved by invoking a user-defined conversion operator
- a class intended as a candidate for derivation should always, if it provides a delete operator, supply the optional second argument of type size_t, which is initialized with the size in bytes of the object to be deleted, before the designer of a derived class chooses to reuse the new and delete operators of the base class, these operators must be examined for type-dependent size assumptions
  - virtual functions
  - the base type virtual function serves as a placeholder for as-yet-undetermined derived class types and never been invoked
  - a virtual function is undefined (pure virtual function) for an abstract class by initializing its declaration to 0, or provide a definition to serve as a default instance of the function
  - a class with pure virtual functions can be only used as a base class for subsequent derivations

- the destructor of an abstract class should always be specified as virtual
- virtual base class
- ordinarily a derived class can explicitly initialize only its immediate base class, but a virtual base class is initialized by it most derived class
- in a nonvirtual derivation, each derived class object contains a contiguous base and a derived class part, but in a virtual derivation, each derived class object contains a derived part and a pointer to the virtual base class part, and the virtual base class is not contained within the derived class object
- in a virtual derivation, the most derived instance of the member (or nested type) is said to dominate the inheritance chain
- virtual base classes are constructed before nonvirtual base classes regardless of where they appear either in the base derivation list or the derived class hierarchy, if a class has multiple immediate virtual base classes, the associated constructors are invoked in the order the class appear within the base derivation list
- when a derivation contains both virtual and nonvirtual instances of a base class, one base class object is created for each nonvirtual instance and another base class object is created for all virtual instances
- run-time type information (RTTI)
  - downcast: casting from a base class to a derived class
  - upcast: casting from a derived class to a base class
  - crosscast: a cast that goes from a base to a sibling class
  - dynamic casting only restrict to polymorphic types. A dynamic_cast can cast from a polymorphic virtual base to a derived class or a sibling class. When a dynamic_cast is used for a pointer type, a 0 indicate failure. Dynamic casting has a small run-time cost associate it usage
  - static casting does not examine (and not check) the object it casts from, so can't cast from virtual base. Static casting does not have run-time cost
- object-oriented design
- identify the classes and identify the relationships (is-a or has-a)
- define the interface (or set of operations) to the class or class hierarchy

Standard Template Library

Containers	vector	one-dimensional array of T
	list	doubly-linked list of T
	deque	doubly-ended queue of T
	queue	queue of T
	stack	stack of T
	map	associative array of T
	set	set of T
	bitset	set of boolean
General utilities	utility	operators and pairs
	functional	function objects
	memory	allocators for containers
	ctime	C-style date and time
Iterators	iterator	iterators and iterator support
Algorithms	algorithm	general algorithms
	cstdlib	bsearch(), qsort( ), etc.
Diagnostics	stdexcept	standard exceptions
	cassert	assert macro
	cerrno	C-style error handling
Strings	string	sting of T
	cctype	character classification
	cwtype	wide-character classification
	cstring	C-style string functions
	cwchar	C-style wide-character string functions
	cstdlib	C-style standard library
Input/Output	iosfwd	forward declarations of I/O facilities
	iostream	standard iostream objects and operations
	ios	iostream bases
	streambuf	stream buffers
	istream	input stream template
	ostream	output stream template
	iomanip	manipulators
	sstream	streams to/from strings
	cstdlib	character type functions
	fstream	streams to/from files
	cstdio	C standard I/O library, e.g., printf( )
	cwchar	C-style I/O of wide characters
Localization	locale	represent cultural differences
	clocale	represent C-style cultural differences
Language Support	limits	numeric limits
	climits	C-style numeric scalar-limit macros
	cfloat	C-style numeric floating-point limit macros
	new	dynamic memory management
	typeinfo	run-time type identification support
	exception	exception-handling support
	cstddef	C library language support

	cstdarg	variable-length function argument list
	csetjmp	C-style stack unwinding
	cstdlib	program termination, e.g., exit( )
	ctime	system clock
	csignal	C-style signal handling
Numerics	complex	complex numbers and operators
	valarray	numeric vectors and operators
	numerics	generalized numeric operators
	cmath	standard mathematical functions
	cstdlib	C-style random numbers

- Standard Containers

	Container Member Access and Operations
front( )	first element
back( )	last element
[ ]	subscripting, unchecked access (not for list)
at( )	subscripting, checked access (not for list)
size( )	number of elements
empty( )	is the container empty?
max_size( )	size of the largest possible container
capacity( )	space allocated for vector (for vector only)
reserve( )	allocated space for vector (for vector only)
resize( )	sdd elements to (end of) vector (for vector only)
swap( )	swap elements of two containers
get_allocator( )	het a copy of the container's allocator
==	is the content of two containers the same?
!=	is the content of two containers different?
<	is one container lexicographically before another?

	Iterators
begin( )	points to first element
end( )	points to one-past-last element
rbegin( )	points to first element of reverse sequence
rend( )	points to one-past-last element of reverse sequence

	Stack and Queue Operations
push_back( )	add to end
pop_back( )	remove last element
push_front( )	add new first element (for list and deque only)
pop_front( )	remove first element (for list and deque only)

	List Operations
insert(p, x)	add x before p
insert(p, n, x)	add n copies of x before p
insert(p, first, last)	add elements from [first…last] before p
erase(p)	remove element at p
erase(first, last)	erase [first…last]
clear( )	Erase all elements

	Constructors
container( )	empty container
container(n)	n elements default value (not for associative containers)
container(n, x)	n copies of x (not for associative containers)
container(first, last)	initial element from [first…last]
container(x)	copy constructor, initial elements from container x
~container( )	destroy the container and all of its elements

	Assignments
operator=(x)	copy assignment, elements from container x
assign(n)	assign n elements default value (not for associative containers)
assign(n, x)	assign n copies of x (not for associative containers)
assign(first, last)	assign from [first…last]

	Associative Operations
operator[ ](k)	access the element with key k (for containers with unique keys)
find(k)	find the element with key k
lower_bound(k)	find the first element with key k
upper_bound(k)	find the first element with key greater than k
equal_range(k)	find the lower_bound and upper_bound of elements with key k
key_comp( )	copy of the key comparison object
value_comp( )	copy of the mapped_value comparison object

- Algorithms and Function Objects

	Nonmodifying Sequence Operations
for_each( )	do operation for each element in a sequence
find( )	find first occurrence of a value in a sequence
find_if( )	find first match of a predicate in a sequence
find_first_of( )	find a value from one sequence in another
adjacent_find( )	find an adjacent pair of values
count( )	count occurrences of a value in a sequence
count_if( )	count matches of a predicate in a sequence
mismatch( )	find the first element for which two sequences

	differ
equal( )	are the elements of two sequences pairwise equal
search( )	find the first occurrence of a sequence as a subsequence
find_end( )	find the last occurrence of a sequence as a subsequence
search_n( )	find the n^{th} occurrences of a value in a sequence

	Modifying Sequence Operations
transform( )	apply an operation to every element in a sequence
copy( )	copy a sequence starting with its first element
copy_backward( )	copy a sequence starting with its last element
swap( )	swap two elements
iter_swap( )	swap two elements pointed to by iterators
swap_ranges( )	swap elements of two sequences
replace( )	replace elements with a given value
replace_if( )	replace elements matching a predicate
replace_copy( )	copy sequence replacing elements with a given value
replace_copy_if( )	copy sequence replacing elements matching a predicate
fill( )	replace every element with a given value
fill_n( )	replace first n elements with a given value
generate( )	replace every element with the result of an operation
generate_n( )	replace first n elements with the result of an operation
remove( )	remove elements with a given value
remove_if( )	remove elements matching a predicate
remove_copy( )	copy a sequence removing elements with a give value
remove_copy_if( )	copy a sequence removing elements matching a predicate
unique( )	remove equal adjacent elements
unique_copy( )	copy a sequence removing equal adjacent elements
replace_copy( )	copy sequence replacing elements with a given value
reverse( )	reverse the order of elements
reverse_copy( )	copy a sequence into reverse order
rotate( )	rotate elements
rotate_copy( )	copy a sequence into a rotated sequence

random_shuffle( )	move elements into a uniform distribution

	Sorted Sequence
sort( )	sort with good average efficiency
stable_sort( )	sort maintaining order of equal elements
partial_sort( )	get the first part of sequence into order
partial_sort_copy( )	copy getting the first part of output into order
nth_element( )	put the n^{th} element in its proper place
lower_bound( )	find the first occurrence of a value
upper_bound( )	find the last occurrence of a value
equal_range( )	find a subsequence with a given value
binary_search( )	is a given value in a sorted sequence?
merge( )	merge two sorted sequences
inplace_merge( )	merge two consecutive sorted subsequences
partition( )	place elements matching a predicate first
stable_partition( )	place elements matching a predicate first and preserve relative order

	Set Algorithms
includes( )	is a sequence a subsequence of another?
set_union( )	construct a sorted union
set_intersection( )	construct a sorted intersection
set_difference( )	construct a sorted sequence of elements, in the first but not the second sequence
Set_symmetric_difference( )	construct a sorted sequence of elements, in one but not both sequence

	Heap Operations
make_heap( )	make sequence ready to be used in a heap
push_heap( )	add element to heap
pop_heap( )	Remove element from heap
sort_heap( )	sort the heap

	Minimum and Maximum
min( )	smaller of two values
max( )	larger of two values
min_element( )	smallest value in sequence
max_element( )	largest value in sequence
Lexicographical_compare( )	lexicographically first of two values

	Permutations
next_permutation( )	next permutation in lexicographical order

prev_permutation( )	previous permutation in lexicographical order

- Functional

Predicates		
equal_to	binary	arg1 == arg2
not_equal_to	binary	arg1 != arg2
greater	binary	arg1 > arg2
less	binary	arg1 < arg2
greater_equal	binary	arg1 >= arg2
less_equal	binary	arg1 <= arg2
logical_and	binary	arg1 && arg2
logical_or	binary	arg1 || arg2
logical_not	unary	! arg

Arithmetic Operations		
plus	binary	arg1 + arg2
minus	binary	arg1 - arg2
multiplies	binary	arg1 * arg2
divides	binary	arg1 / arg2
modulus	binary	arg1 % arg2
negate	unary	- arg

	Binders, Adapters, and Negaters
bind2nd(y)	call binary function with y as 2nd argument
bind1st(x)	call binary function with x as 1st argument
mem_func( )	call 0-argument or unary member through pointer
mem_func_ref( )	call 0-argument or unary member through reference
ptr_fun( )	call unary or binary pointer to function
not1( ), not2( )	negate unary/binary predicate

- Iterator Operations

Category: Abbreviation:	output Out	input In	forward For	bidirectional Bi	random-access Ran
read:		= *p	= *p	= *p	= *p
access:		->	->	->	-> [ ]
write:	*p =		*p =	*p =	*p =
iteration:	++	++	++	++ --	++ -- + - += -=
comparison:		== !=	== !=	== !=	== != < > <= >=

50 Specific Ways to Improve Your Programs & Designs

Shifting from C to C++

1. use const and inline instead of #define
2. prefer iostream.h to stdio.h
3. use new and delete instead of malloc and free
4. prefer C++ comments

Memory Management

5. use the same form in corresponding calls to new and delete
6. call delete on pointer members in destructors
7. check the return value of new
8. adhere to convention when writing new
9. avoid hiding the global new
10. write delete if you write new

Constructors, Destructors, and Assignment Operators

11. define a copy constructor and an assignment operator for class with dynamically allocated memory
12. prefer initialization to assignment in constructors
13. list members in an initialization list in the order in which they are declared
14. make destructors virtual in base classes
15. have operator= return a reference to *this
16. assign to all data members in operator=
17. check for assignment to self in operator=

Classes and Functions: Design and Declaration

18. strive for class interfaces that are complete and minimal
19. differentiate among member functions, global functions, and friend functions
20. avoid data members in the public interface
21. use const whenever possible
22. pass and return objects by reference instead of by value
23. don't try to return a reference when you must return an object
24. choose carefully between function overloading and parameter defaulting
25. avoid overloading on a pointer and a numerical type
26. guard against potential ambiguity
27. explicitly disallow use of implicitly generated member functions you don't want
28. use struct to partition the global namespace

Classes and Functions: Implementation

29. avoid returning "handles" to internal data from const member function
30. avoid member functions that return pointers or references to members less accessible than themselves
31. never return a reference to a local object or a dereferenced pointer initialized by new within the function
32. use enums for integral class constants
33. use inlining judiciously
34. minimize compilation dependencies between files

Inheritance and Object-Oriented Design

35. make sure public inheritance models "isa"

36. differentiate between inheritance of interface and inheritance of implementation
37. never redefine an inherited nonvirtual function
38. never redefine an inherited default parameter value
39. avoid casts down the inheritance hierarchy
40. model "has-a" or "is-implemented-in-terms-of" through layering
41. use private inheritance judiciously
42. differentiate between inheritance and templates
43. use multiple inheritance judiciously
44. say what you mean; understand what you're saying

Miscellany

45. know what functions C++ silently writes and calls
46. prefer compile-time and link-time errors to runtime errors
47. ensure that global objects are initialized before they're used
48. pay attention to compiler warnings
49. plan for coming language feature
50. read the ARM

35 New Ways to Improve Your Programs and Designs

Basics

1. distinguish between pointers and references
2. prefer C++ style casts
3. never treat arrays polymorphically
4. avoid gratuitous default constructors

Operators

5. be wary of user-defined conversion functions
6. distinguish between prefix and postfix forms of incremental and decremental operators
7. never overload &&, ||, or ,
8. understand the different meanings of new and delete

Exceptions

9. use destructors to prevent resource leaks
10. prevent resource leaks in constructors
11. prevent exceptions from leaving destructors
12. understand how throwing an exception differs from passing a parameter or calling a virtual function
13. catch exception by reference
14. use exception specifications judiciously
15. understand the costs of exception handling

Efficiency

16. remember the 80-20 rule
17. consider using lazy evaluation
18. amortize the cost of expected computations
19. understand the origin of temporary objects
20. facilitate the return value optimization
21. overload to avoid implicit type conversions
22. consider using op= instead of stand-alone op
23. consider alternative libraries
24. understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI

Techniques

25. virtualizing constructors and non-member functions
26. limiting the number of objects of a class
27. requiring or prohibiting heap-based objects
28. smart pointers
29. reference counting
30. proxy classes
31. making functions virtual with respect to more than one object

Miscellany

32. program in the future tense
33. make non-leaf classes abstract
34. understand how to combine C++ and C in the same program
35. familiarize yourself with the language standard