Effective C++ 3rd edition

# Terminology

* Declaration: Tells compilers about the name and type of something, but it omits certain details

extern int x;

class Widget;

int numDigits(int number);

* Definition:Providers compilers with the details a declaration omits
* Initialization: The process of giving an object its first value.
* Default constructor: One that can be called without any arguments

class A {

public: A(int x = 0); // Also a default constructor

}

* Copy constructor: Used to initialise an object with a different object for the same type.

Copy assignment operator: Used to copy the value from one object to another of the same type

class A {

A(int x = 0); // Also a default constructor

A(const A& a); // Copy constructor

A& operator=(const A& a); // copy assignment operator

}

A a1; // invoke default constructor

A a2(a1); // invoke copy constructor

a1 = a2; // invoke copy assignment operator

A a3 = a2; // invoke copy constructor, if a new object is being defined

void some\_func(A a); // Pass by value, copying done by copy constructor

* Undefined behaviour: Cannot reliably predict what will happen at runtime. Normally bugs
* Interface: No interface in C++
* Client: Someone or something uses the code you write

# Accustoming yourself to C++

## Item 1: View C++ as a federation of languages

* View as 4 sub languages: C, Object-Oriented C++, Template C++, STL
* Rules for effective C++ programming vary, depending on the part of C++ you are using.

## Item 2: Prefer consts, enums, and inlines to #defines

* Prefer the compiler to the pre-processor

#define ASPECT\_RATIO 1.653

const double AspectRatio = 1.653; // uppercase names are usually for macros

const char\* const name = ""; // Need two const

class {

static const int NumTurns; // Class scope constant needs to be static

int scores[NumTurns]; // Some old compiler may reject this

enum { NumTurns = 5; } // Enum hack - Makes NumTurns a symbolic name for 5

int scores[NumTurns];

}

## Item 3: Use const whenever possible

char greeting[] = "Hello";

const char\* p = greeting; // non-const pointer, const data

char\* const p = greeting; // const pointer, non-const data

const char\* const p = greeting; // const pointer, const data

* If const appears to the left of \*, what's pointed to is constant.
* If const appears to the right of \*, the pointer itself is constant.

const char\* p is the same as char\* const p

* const member functions

class TextBlock {

const char& operator[] (int position) const { return text[position]; }

char& operator[] (int position) { return text[position]; }

// Return type is char& so it can be modified

}

TextBlock tb; tb[0]; // Calls non-const TextBlock::operator []

const TextBlock ctb; ctb[0]; // calls const TextBlock::operator []

const member functions cannot modify class members (bitwise/physical constness), but still possible to modify the object which the data member points to (logical constness).

mutable keyword allows modifying class member.

mutable int mData; // Can be modified inside a const function

* When const and non-const functions have essentially identical implementations, code duplication can be avoided by having the non-const version call the const version with cast.

## Item 4: Make sure that objects are initialized before they're used

* Manually initialize objects of built-in type, because C++ only sometimes initializes them itself
* In a constructor, prefer use of the member initialization list to assignment inside the body of the constructor. List data members in the initialization list in the same order they’re declared in the class.

class A::A(const std::string& name) {

mName = name; // Assignment. Initialized first and then assigned with value.

}

Class A::A(const std::string& name) : mName(name) {} // Initializations

* Avoid initialization order problems across translation units by replacing non-local static objects with local static objects: Singleton pattern to return object instead of referencing extern object

# Constructors, Destructors and Assignment Operators

## Item 5: Know what functions C++ silently writes and calls

* Compilers may implicitly generate a class’s default constructor, copy constructor, copy assignment operator, and destructor.

Compiler will reject default assignment operator on reference(&) or const data member.

## Item 6: Explicitly disallow the use of compiler-generated functions you do not want

* To disallow functionality automatically provided by compilers, declare the corresponding member functions private and give no implementations. Using a base class like Uncopyable is one way to do this.

Class Uncopyable {

private:

// Only declares the copy constructor and copy operator,

// but not define them: linker error

Uncopyable(const Uncopyable&);

Uncopyable& operator=(const Uncopyable&);

}

// Not defining copy constructor and copy operator.

// Compilation error when try using them.

Class A : private Uncopyable {

}

## Item 7: Declare destructors virtual in polymorphic base classes

* Polymorphic base classes should declare virtual destructors. If a class has any virtual functions, it should have a virtual destructor.
* Classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.
* Virtual functions increases the object size: Object needs to carry extra vptr(virtual table pointer) to be used at runtime to determine which virtual functions should be invoked.

## Item 8: Prevent exceptions from leaving destructors

* Destructors should never emit exceptions. If functions called in a destructor may throw, the destructor should catch any exceptions, then swallow them or terminate the program.
* If class clients need to be able to react to exceptions thrown during an operation, the class should provide a regular (i.e., non-destructor) function that performs the operation.

## Item 9: Never call virtual functions during construction or destruction

* Don’t call virtual functions during construction or destruction, because such calls will never go to a more derived class than that of the currently executing constructor or destructor.

## Item 10: Have assignment operators return a reference to \*this

* Assignment is right-associative

x = y = 15; is the same as x = (y = 15); values assigned to y then result assigned to x

* Have assignment operators return a reference to \*this.

class A {

A& operator=(const A& a) { return \*this; }

A& operator +=(const A& a) { return \*this; }

}

## Item 11: Handle assignment to self in operator =

* Make sure operator= is well-behaved when an object is assigned to itself. Techniques include comparing addresses of source and target objects, careful statement ordering, and copy-and-swap.
* Make sure that any function operating on more than one object behaves correctly if two or more of the objects are the same.

Wrong way:

Widget& Widget::operator=(const Widget& rhs) {

delete pb; // Unsafe: May delete own data

pb = new Bitmap(\*rhs.pb); // May throw exception and make pb invalid data

return \*this;

}

Widget& Widget::operator=(const Widget& rhs) {

if (this == &rhs) return \*this; // Check whether is self

Bitmap \*pOrig = pb; // Remember original data

pb = new Bitmap(\*rhs.pb); // Point pb to a copy

delete pOrig; // delete original

return \*this;

}

## Item 12: Copy all parts of an object

* Copying functions should be sure to copy all of an object’s data members and all of its base class parts.

PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs):

Customer(rhs), priority(rhs.priority) {

}

PriorityCustomer& PriorityCustomer::operator=(const PriorityCustomer& rhs) {

Customer::operator=(rhs); // assign base class parts

priority = rhs.priority;

return \*this;

}

* Don’t try to implement one of the copying functions in terms of the other. Instead, put common functionality in a third function that both call.

# Resource Management

## Item 13: Use objects to manage resources

* Resources are acquired and immediately turned over to resource-managing objects
* Resource-managing objects use their destructors to ensure that resources are released
* Two common classes are std::tr1::shared\_ptr and auto\_ptr. tr1::shared\_ptr is usually the better choice, because its behavior when copied is intuitive. Copying an auto\_ptr sets it to null.

std::auto\_ptr<Investment> pInv1(createInvestment());

std::auto\_ptr<Investment> pInv2(pInv1); // pInv2 has the object; pInv1 is null

pInv1 = pInv2; // pInv1 points to the object; pInv2 is null

tr1::shared\_ptr<Investment> pInv1(createInvestment());

tr1::shared\_ptr<Investment> pInv2(pInv1); // Both point to the object

pInv1 = pInv2; // Nothing changes

## Item 14: Think carefully about copying behavior in resource-managing classes

Possible copying behaviors:

* Prohibit copying

class Lock : private Uncopyable

* Reference-count the underlying resource

class Lock {

// shared\_ptr accepts delete function pointer in the initialization

explicit Lock(Mutex \*pm) : mutexPtr(pm, unlock) {

lock(mutexPtr.get());

}

std::tr1:shared\_ptr<Mutex> mutexPtr;

}

* Copy the underlying resource: deep copy
* Transfer ownership of the underlying resource

## Item 15: Provide access to raw resources in resource-managing classes

* Explicit conversion and implicit conversion

class Font {

FontHandle get() const { return handle; } // explicit conversion function

Operator FontHandle() const {return handle; } // implicit conversion

}

* Explicit conversion is safer, but implicit conversion is more convenient for clients

Font f1; FontHandle f2 = f1;

// Meant to copy a font object, but instead implicitly converted f1 into its underlying FontHandle, then copied that.

## Item 16: Use the same form in corresponding uses of new and delete

* If use [] in a new expression, must use [] in the corresponding delete expression.
* If don’t use [] in a new expression, mustn’t use [] in the corresponding delete expression.

## Item 17: Store newed objects in smart pointers in standalone statements

void processWidget(std::tr1::shared\_ptr<Widget> pw, int priority);

processWidget(std::tr1::shared\_ptr<Widget>(new Widget), priority());

// May have resource leak(, compiler will generate 3 steps:

// new Widget(); Call shared\_ptr constructor; call priority();

// Execution order is not fixed!!!

// If runs in new, priority(), shared\_ptr(), and priority throws exception,

// may have memory leak.

# (ToDesigns and Declarations

## Item 18: Make interfaces easy to use correctly and hard to use incorrectly

* Ways to facilitate correct use include consistency in interfaces and behavioral compatibility with built-in types.
* Ways to prevent errors include creating new types, restricting operations on types, constraining object values, and eliminating client resource management responsibilities.
* tr1::shared\_ptr supports custom deleters. This prevents the cross-DLL problem, can be used to automatically unlock mutexes, etc.

## Item 19: Treat class design as type design

## Item 20: Prefer pass-by-reference-to-const to pass-by-value

* By default, C++ passes objects to and from functions by value (inherited from C)

bool validateStudent(const Student& s);

* Slicing problem: Only the base class will be created when use base class as type and pass by value
* The rule doesn’t apply to built-in types and STL iterator and function object types. For them, pass-by-value is usually appropriate.

## Item 21: Don’t try to return a reference when you must return an object

* Never return a pointer or reference to a local stack object, a reference to a heap-allocated object, or a pointer or reference to a local static object if there is a chance that more than one such object will be needed.

## Item 22: Declare data members private

* Declare data members private. It gives clients syntactically uniform access to data, affords fine-grained access control, allows invariants to be enforced, and offers class authors implementation flexibility.

## Item 23: Prefer non-member non-friend functions to member functions

* Prefer non-member non-friend functions to member functions (for utility functions).
* Put function which can be implemented entirely in terms of the public interface as non-member non-friend
* Increases encapsulation (reducing functions with access to private data members), packaging flexibility, and functional extensibility. Can use namespace (to group the functions.

## Item 24: Declare non-member functions when type conversions should apply to all parameters

* If you need type conversions on all parameters to a function the function must be a non-member.

const Rational operator\* (const Rational& lhs, const Rational& rhs);

## Item 25: Consider support for a non-throwing swap

(To read)

# Implementations

## Item 26: Postpone variable definitions as long as possible

* It increases program clarity and improves program efficiency.

## Item 27: Minimize casting

* C style: (T)expression T(expression)
* const\_cast: The only C++ style cast can cast away constness of objects
* dynamic\_cast: Perform “safe downcasting”, to determine whether an object is of a particular type in an inheritance hierarchy. Only C++ style that cannot be performed using old-c style syntax. Have significant runtime cost. Return null if not the type.
* reinterpret\_cast: Intended for low-level casts that yield implementation-dependent results, e.g cast a pointer to an int. Should be rare outside low-level code.
* static\_cast: For implicit conversions, eg. non-const to const, int to double, void\* to typed pointers, pointer-to-base to pointer-to-derived.
* A single object might have more than one address by pointers.

class Base; class Derived :

public Base; Derived d;

Base\* pBase = &d; Derived\* pDerived = &d;

// pBase and pDerived may have diferent addresses based on different compilers.

class Window {virtual void onResize {…}}

class SpecialWindow : public Window {

void onResize() {

// To run Window onResize first and do some extra work

// Wrong: Actually creates a copy and then call onResize on that copy

static\_cast<Window>(\*this).onResize();

// Use this

Window::onResize();  
 }

}

* Avoid casts whenever practical, especially dynamic\_casts in performance-sensitive code. If a design requires casting, try to develop a cast-free alternative.
* When casting is necessary, try to hide it inside a function. Clients can then call the function instead of putting casts in their own code.
* Prefer C++-style casts to old-style casts. They are easier to see, and they are more specific about what they do.

## Item 28: Avoid returning “handles” to object internals

* Avoid returning handles (references, pointers, or iterators) to object internals. Not returning handles increases encapsulation, helps const member functions act const, and minimizes the creation of dangling handles.

## Item 29: Strive for exception-safe code

* When an exception is thrown, exception-safe functions:
  + Leak no resources
  + Don’t allow data structures to become corrupted
* Exception-safe functions offer one of three guarantees:
  + Basic guarantee: If an exception is thrown, everything in the program remains in a valid state. No object or data corruption and in a consistent state (may be unpredictable).
  + Strong guarantee: if an exception is thrown, the state of the program is unchanged. Either succeed completely or state not changed as if not called.
  + Nothrow guarantee: never to throw exceptions, because they always do what they promise to do.

void func() throw(); // throw() is to note empty exception spec. But if throws inside the func, unexpected function should be called, crash

* Strong guarantee can often be implemented via copy-and-swap (put new object in a copy and swap with current), but the strong guarantee is not practical for all functions.
* A function can usually offer a guarantee no stronger than the weakest guarantee of the functions it calls.

## Item 30: Understand the ins and outs of inlining

* Implicit inline: Function is defined in a class definition

class Person {

int age() const { return mAge; }

}

* Explicit inline: inline keyword in the function declaration
* Limitations:
  + Increase code size
  + A request other than a command, not guaranteed to be inlined
  + May not be able to debug as the function not exist
* Limit most inlining to small, frequently called functions. This facilitates debugging and binary upgradability, minimizes potential code bloat, and maximizes the chances of greater program speed.
* Don’t declare function templates inline just because they appear in header files.

## Item 31: Minimize compilation dependencies between files

* Design strategy:
  + Avoid using objects when object references and pointers will do
  + Depend on class declarations instead of class definitions whenever you can

class Date; // Declaration

Date today(); void setDate(Date date); // No definition of Date needed

* + Provide separate header files for declarations and definitions
* The general idea behind minimizing compilation dependencies is to depend on declarations instead of definitions. Two approaches based on this idea are Handle classes and Interface classes.
* Handle class: Pointer to implementation (pimpl idiom).a Forward function call to Impl

class Person { private: PersonImpl\* pImpl; }

* Interface class:

class Person {

public:

virtual ~Person(); virtual std::string name() const = 0;

static std::tr1::shared\_ptr<Person> create(const std::string& name);

// Can use static factory method to return the instance.

}

class RealPerson { } // The implementation

* Library header files should exist in full and declaration-only forms. This applies regardless of whether templates are involved.

# Inheritance and Object-Oriented Design

## Item 32: Make sure public inheritance models “is-a.”

* Public inheritance means “is-a.” Everything that applies to base classes must also apply to derived classes, because every derived class object is a base class object.

## Item 33: Avoid hiding inherited names

* Names in derived classes hide names in base classes. Under public inheritance, this is never desirable.

class A { void func(); void func(int n); }

class B : public A { void func(); }

B b; b.func(0); // Compilation error! Func is hidden

* To make hidden names visible again, employ using declarations or forwarding functions.
  + Using keyword:

class B: public A { using A::func; }

* + Forwarding functions: Can selectively choose what function to forward

class B: private A { void func() { A::func(); } }

## Item 34: Differentiate between inheritance of interface and inheritance of implementation

* Inheritance of interface is different from inheritance of implementation. Under public inheritance, derived classes always inherit base class interfaces.
  + Pure virtual functions specify inheritance of interface only.
  + Simple (impure) virtual functions specify inheritance of interface  plus inheritance of a default implementation.
  + Non-virtual functions specify inheritance of interface plus inheritance of a mandatory implementation.

## Item 35: Consider alternatives to virtual functions

* Non-virtual interface idiom: A form of the Template Method design pattern that wraps public non-virtual member functions around less accessible virtual functions

class GameCharacter {

public: int healthValue() const { return doHealthValue(); }

private: int doHealthValue() { … } }

* Strategy Pattern via function pointers: Function pointers as data member

typedef int (\*HealthCalcFunc)(const GameCharacter&);

* + Different character can have different health calculation functions
  + Functions can be changed at runtime
* Strategy Pattern via tr1::function: Function object as data member

typedef std::tr1::function<int (const GameCharacter&)> HealthCalcFunc;

* Allowing use of any callable entity which can be converted to method signature:

std::tr1::function<int (const GameCharacter&)>

short calHealth(const GameCharacter&); // Return type not int

struct HealthCalculator {

// Class for health calculation function objects

// HealthCalcFunc = HealthCalculator();

int operator()(const GameCharacter&) const { ...}

}

* A disadvantage of moving functionality from a member function to a function outside the class is that the non-member function lacks access to the class’s non-public members.

## Item 36: Never redefine an inherited non-virtual function

* Virtual functions are dynamically bound. Non-virtual functions are statically bound

class B { void func(); }

class D : class B { void func(); }

D d; B\* pB = &d; pB->func(); // calls func in B

D\* pD = &d; pD->func(); // calls func in D

## Item 37: Never redefine a function’s inherited default parameter value

* Default parameters are statically bound

class Shape { virtual void draw(Color=Red); }

class Rectangle : Shape { virtual void draw(Color=Green); }

Rectangle rec; rec.draw(); // Draw with Red!!

## Item 38: Model “has-a” or “is-implemented-in-terms-of” through composition

* Composition has meanings completely different from that of public inheritance.
* In the application domain, composition means has-a. In the implementation domain, it means is-implemented-in-terms-of.

## Item 39: Use private inheritance judiciously

* Private inheritance means is-implemented-in-terms-of
* Private inheritance means nothing during software design, only during software implementation
* Use composition whenever you can, and use private inheritance whenever you must: When a derived class needs access to protected base class members or needs to redefine inherited virtual functions.
* Unlike composition, private inheritance can enable the empty base optimization. This can be important for library developers who strive to minimize object sizes.

## Item 40: Use multiple inheritance judiciously

* Multiple inheritance is more complex than single inheritance. It can lead to new ambiguity issues and to the need for virtual inheritance.
* Virtual inheritance: No data replication

class File { string filename; }

class InputFile : virtual public File {}

class OuputFile : virtual public File {}

class IOFile : public InputFile, public OutputFile {}

// Using virtual, there will be only one filename in IOFile.

// Remove virtual, there are two: InputFile::filename, OutputFile::filename

* Avoid using virtual inheritance. If have to use, avoid putting data into them.
* Virtual inheritance imposes costs in size, speed, and complexity of initialization and assignment. It’s most practical when virtual base classes have no data.
* Multiple inheritance does have legitimate uses. One scenario involves combining public inheritance from an Interface class with private inheritance from a class that helps with implementation.

# Templates and Generic Programming

## Item 41: Understand implicit interfaces and compile-time polymorphism

* Both classes and templates support interfaces and polymorphism.
* For classes, interfaces are explicit and centered on function signatures. Polymorphism occurs at runtime through virtual functions.
* For template parameters, interfaces are implicit and based on valid expressions. Polymorphism occurs during compilation through template instantiation and function overloading resolution.

## Item 42: Understand the two meanings of typename

* When declaring template parameters, class and typename are interchangeable.

template<class T> class Widget;

template<typename T> class Widget; // same!

* Use typename to identify nested dependent type names, except in base class lists or as a base class identifier in a member initialization list.

typename C::const\_iteratoriter(container.begin());

* C::const\_iterator, a type that depends on the template parameter C.
* Names in a template that are dependent on a template parameter are called dependent names. When a dependent name is nested inside a class, call it a nested dependent name.
* Whether need typename may depend on the compiler

## Item 43: Know how to access names in templatized base classes

* In derived class templates, refer to names in base class templates via a “this->” prefix, via using declarations, or via an explicit base class qualification.

class CompanyA { public: void sendClear() const; }

class CompanyB { public: void sendEncrypted(); }

template<typename Company> class MessageSender {

void sendMessage(const Company& c) { c.sendClear(); } }

CompanyA ca; CompanyB cb;

MessageSender<CompanyA> sa; sa.sendMessage(); // OK

MessageSender<CompanyB> sb; sb.sendMessage(); // Error: CompanyB does not have sendClear function

// Template specialization

template<> class MessageSender<CompanyB> {

void sendMessage(const CompanyB& c) { c.sendEncrypted(); }}

MessageSender<CompanyB> sb; sb.sendMessage(); // OK

template<typename Company>

class LoggingMessageSender : public MessageSender<Company> {

public:

// Solution: using Message<Company>::sendMessage;

void sendMessageWithLogging(const Company& c) {

sendMessage( c ); // Error:

// Solution:

// this->sendMessage(c);

// MessageSender<Company>::sendMessage(c);

}

}

## Item 44: Factor parameter-independent code out of templates

* Templates generate multiple classes and multiple functions, so any template code not dependent on a template parameter causes bloat.
* Bloat due to non-type template parameters can often be eliminated by replacing template parameters with function parameters or class data members.
* Bloat due to type parameters can be reduced by sharing implementations for instantiation types with identical binary representations (void\*).

Template<typename T, size\_t n>

class SquareMatrix { public: void invert(); }

SquareMatrix<double, 5> sm1; sm1.invert();

SquareMatrix<double, 10> sm2; sm2.invert(); // Will generate different replicated code

template<typename T> class SquareMatrixBase {

protected:

void invert(size\_t matrixSize); }

template<typename T, size\_t n>

class SquareMatrix : private SquareMatrixBase {

private: using SquareMatrixBase<T>::invert;

public: void invert() { invert(n); }}

## Item 45: Use member function templates to accept “all compatible types.”

* Use member function templates to generate functions that accept all compatible types.
* If you declare member templates for generalized copy construction or generalized assignment, you’ll still need to declare the normal copy constructor and copy assignment operator, too.

## Item 46: Define non-member functions inside templates when type conversions are desired.

* When writing a class template that offers functions related to the template that support implicit type conversions on all parameters, define those functions as friends inside the class template.

## Item 47: Use traits classes for information about types

* Traits classes make information about types available during compilation. They’re implemented using templates and template specializations
* In conjunction with overloading, traits classes make it possible to perform compile-time if...else tests on types.

## Item 48: Be aware of template metaprogramming

* Template metaprogramming can shift work from runtime to compile-time, thus enabling earlier error detection and higher runtime performance.
* TMP can be used to generate custom code based on combinations of policy choices, and it can also be used to avoid generating code in- appropriate for particular types.

# Customizing new and delete

## Item 49: Understand the behavior of the new-handler

* set\_new\_handler allows you to specify a function to be called when memory allocation requests cannot be satisfied.
* Nothrow new is of limited utility, because it applies only to memory allocation; associated constructor calls may still throw exceptions.

## Item 50: Understand when it makes sense to replace new and delete

* There are many valid reasons for writing custom versions of new and delete, including improving performance, debugging heap usage errors, and collecting heap usage information.

## Item 51: Adhere to convention when writing new and delete

* operator new should contain an infinite loop trying to allocate memory, should call the new-handler if it can’t satisfy a memory request, and should handle requests for zero bytes. Class-specific versions should handle requests for larger blocks than expected.
* operator delete should do nothing if passed a pointer that is null. Class-specific versions should handle blocks that are larger than expected.

## Item 52: Write placement delete if you write placement new

* When you write a placement version of operator new, be sure to write the corresponding placement version of operator delete. If you don’t, your program may experience subtle, intermittent memory leaks.
* When you declare placement versions of new and delete, be sure not to unintentionally hide the normal versions of those functions.

# Miscellany

## Item 53: Pay attention to compiler warnings

* Take compiler warnings seriously, and strive to compile warning-free at the maximum warning level supported by your compilers.
* Don’t become dependent on compiler warnings, because different compilers warn about different things. Porting to a new compiler may eliminate warning messages you’ve come to rely on.

## Item 54: Familiarize yourself with the standard library, including TR1

## Item 55: Familiarize yourself with Boost