# Use *const* whenever possible

*Item 3 – Effective C++ 3rd Edition (Scott Meyers)*

The wonderful thing about const is that it allows you to specify a semantic constraint — a particular object should not be modified — and compilers will enforce that constraint. This **helps the compilers detect usage errors**. In addition, const can be applied to objects at any scope, to function parameters and return types, and to member functions as a whole.

**For pointers:**

char greeting[] = "Hello";

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

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

**For STL iterators:**

STL iterators are modeled on pointers, so an iterator acts much like a T\* pointer.

* Declaring an iterator const is like declaring a pointer const (a T\* const pointer) – the iterator isn’t allowed to point to something different, but the thing it points to may be modified.
* If you want an iterator that points to something that can’t be modified (a const T\* pointer), you want a const\_iterator:

std::vector<int> vec;

...

const std::vector<int>::iterator iter = vec.begin( );

// iter acts like a T\* const pointer

\*iter = 10; // OK, changes what iter points to

++iter; // Error, iter is const

std::vector<int>::const\_iterator cIter = vec.begin( );

// cIter acts like a const T\* pointer

\*cIter = 10; // Error!

++cIter; // OK

**For member functions:**

Following is a simple example of const function.

...

class Test {

int value;

public:

...

// We’ll get COMPILER ERROR if we add a line like "value = 100;"

// in this function.

int getValue() const {return value;}

};

# Direct initialization vs Copy initialization

In C++, there are 3 ways to initialize a variable:

int value1 = 1; // copy initialization

double value2(2.2); // direct initialization

char value3{'c'}; // uniform initialization (only C++11)

* A **direct initialization** is nothing but *default constructor* being called and this takes place as soon as an object for a class is created.
* By contrast, a **copy initialization** means initialization of object using another object of the same class using a *copy constructor*.
* **Uniform initialization** is equivalent to direct initialization, but it’s only available in C++11. Check its separate page [here](#_30j0zll).

To understand their differences, let’s take a look at this example:

#include <iostream>

using namespace std;

class Foo {

public:

Foo() {

cout << "defualt constructor" << endl;

}

Foo(const Foo&) {

cout << "copy constructor" << endl;

}

const Foo& operator= (const Foo&) {

cout << "copy assignment operator" << endl;

}

};

int main() {

Foo a; // call default constructor -> direct init

Foo b(a); // call copy constructor -> ???

Foo c = a; // call copy constructor -> copy init

c = b; // call copy assignment operator

return 0;

}

Output:

default constructor

copy constructor

copy constructor

copy assignment

For variables of primitive data types, **copy** and **direct** initializations are generally similar. However, for objects of class types, they differ quite significantly in sematic. Check [here](https://stackoverflow.com/questions/1051379/is-there-a-difference-between-copy-initialization-and-direct-initialization) and [here](http://cpppearls.blogspot.com/2007/06/direct-initialization-and-copy.html).

Copy constructor gets called internally if we haven’t defined one. In most cases, it works well. But in some cases, it is not preferable if objects consist of pointers or allocation of memory during runtime. In those cases, we **need to define our own copy constructor** (as shown in code above). And with user-defined copy constructor we could perform "Deep Copy" .

Copy initialization and direct initialization are almost everywhere:

Foo\* f = new Foo(2,3); // "=" is copy init, but the RHS is direct init

int x = static\_cast<int>(d); // "=" is copy init, but the RHS is direct init

try {

throw "error"; // copy init ("error" is the copy of string object)

} catch(string s) { } // copy init ("s" is passed by value)

int foo(int&p, int q) { // copy init ("q" is passed by value)

return p+q; // copy init (temporary holder for return)

}

const int& foo(int&p, const int& q) { // direct init ("q" is passed by ref)

int res(p+q); // direct init

return res; // direct init (no temp holder, but ref)

}

After all, you only need to remember this rule of thumb:

Foo b(a); // should

Foo b{a}; // should (C++11 only)

Foo c = a; // avoid

# Uniform initialization

C++ has at least four different initialization notations:

Parenthesized initialization looks like this:

std::string s("hello");

int m = int();

You can use the = notation for the same purpose in certain cases:

std::string s = "hello";

int x = 5;

For POD aggregates, you use braces:

int arr[4] = {0,1,2,3};

struct tm today = {0};

Finally, constructors use member initializers:

struct S {

int x;

S(): x(0) {}

};

This is confusing! Worse yet, in C++03 you can't initialize POD array members and POD arrays allocated using new[]. **C++11 cleans up this mess with a uniform brace notation**:

std::string s{"hello"};

class Foo {

public:

Foo(int i, int j);

};

Foo foo{0, 0}; // C++11 only. Equivalent to: Foo foo(0, 0);

int\* arr = new int[4] {0, 1, 2, 3}; // C++11 only

struct Bar {

int arr[4];

Bar() : arr{1, 2, 3, 4} {} // C++11 only

};

With respect to containers, you can say goodbye to a long list of push\_back() calls. In C++11, you can initialize containers intuitively:

std::vector<string> vec = {"first", "second", "third"};

std::map singers = { {"Lady Gaga", "+1(212)555-7890"},

{"Beyonce Knowles", "+1(212)555-0987"} };

# Postpone variable declarations as long as possible

*Item 26 – Effective C++ 3rd Edition (Scott Meyers)*

Consider the following code:

std::string encryptPassword(const std::string& password)

{

using namespace std;

string encrypted; // Default constructor of class string will be called

if (password.length() < MinimumPasswordLength) {

throw logic\_error("Password is too short");

}

// Set value for encrypted...

return encrypted;

}

The object encrypted isn’t completely unused in this function, but **it’s unused if an exception is thrown**. That is, you’ll pay for the construction and destruction of encrypted even if encryptPassword throws an exception. As a result, you’re better off postponing encrypted’s definition until you know you’ll need it:

std::string encryptPassword(const std::string& password)

{

using namespace std;

if (password.length() < MinimumPasswordLength) {

throw logic\_error("Password is too short");

}

string encrypted;

// Set value for encrypted...

return encrypted;

}

So you **should postpone a variable’s definition until right before you have to use the variable**. It increases program clarity and improves program efficiency.

# Always initialize objects before used

*Item 4 – Effective C++ 3rd Edition (Scott Meyers)*

**Reading uninitialized values yields undefined behavior**. On some platforms, the mere act of reading an uninitialized value can halt your program. More typically, **the result of the read will be semi-random bits**, which will then pollute the object you read the bits into, eventually leading to inscrutable program behavior and a lot of unpleasant debugging.

So the rule is to always initialize your objects before you use them.

**For non-member objects of built-in types:**

You can use *copy initialization* as follows:

int x = 0;

const char\* text = "";

double d; std::cin >> d;

**For non-member objects of class types:**

Instead of copy initialization, you should use *direct initialization* as follows:

std::string s("Hello World"); // should

std::string s = "Hello World"); // avoid

**For member objects:**

The responsibility for initialization falls on constructors:

class PhoneNumber { ... };

class ABEntry { // "Address Book Entry"

public:

ABEntry(const std::string& name, const std::string& address,

const std::list<PhoneNumber>& phones);

private:

std::string mName;

std::string mAddress;

std::list<PhoneNumber> mPhones;

int numTimesConsulted;

};

ABEntry::ABEntry(const std::string& name, const std::string& address,

const std::list<PhoneNumber>& phones)

{

mName = name; // AVOID!

mAddress = address; // These are all assignments,

mPhones = phones; // NOT initializations

numTimesConsulted = 0;

}

This yields objects with the values you expect, but it’s not the best approach. That’s because inside the ABEntry constructor, mName, mAddress, and mPhones aren’t being initialized, they’re being *assigned*. This is similar to:

{

std::string mName; // call defualt ctor

std::string mAddress;

std::list<PhoneNumber> mPhones;

int numTimesConsulted;

mName = name; // call operator =

mAddress = address;

mPhones = phones;

numTimesConsulted = 0;

}

The rules of C++ stipulate that **data members of an object are *initialized* before the body of a constructor is entered**. In other words, initialization took place earlier — when their default constructors were automatically called prior to entering the body of the ABEntry constructor.

A better way to write the ABEntry constructor is using the ***member initialization list***:

ABEntry::ABEntry(const std::string& name, const std::string& address,

const std::list<PhoneNumber>& phones)

: mName(name), // These are now all initializations

mAddress(address),

mPhones(phones),

numTimesConsulted(0)

{} // the ctor body is now empty

Using a member initialization list is almost identical to doing [direct initialization](#_gjdgxs) (or uniform initialization in C++11).

This constructor yields the same output as the one above, but it will be more efficient. The assignment-based version first called *default constructors* to initialize mName, mAddress, and mPhones, then assigned them with new values on top of default-constructed ones. All the work performed in those default constructions was wasted.

However, the member initialization list avoids that problem, because the arguments in the list are used as constructor arguments for data members. In this case, mName is copy-constructed from name, mAddress is copy-constructed from address, and mPhones is copy-constructed from phones.

For most types, **a single call to a copy constructor is more efficient — sometimes *much* more efficient — than a call to the default constructor followed by a call to the copy assignment operator**.

For objects of built-in type like numTimesConsulted, there is no difference in cost between initialization and assignment, but for consistency, it’s often best to initialize everything via member initialization list.

Similarly, you can use the member initialization list even when you want to default-construct a data member; just specify nothing as an initialization argument. For example, if ABEntry had a constructor taking no parameters, it could be implemented like this:

ABEntry::ABEntry( )

: mName(), // call mName’s default ctor

mAddress(), // do the same for mAddress and mPhones

mPhones(),

numTimesConsulted(0) // but explicitly initialize numTimesConsulted to zero

{}

Sometimes the initialization list must be used, even for built-in types. For example, **data members that are const or are references must be initialized; they can’t be assigned**. To avoid having to memorize when data members must be initialized in the member initialization list and when it’s optional, the easiest choice is to always use the initialization list. It’s sometimes required, and it’s often more efficient than assignments.

# Always declare destructors 'virtual' in polymorphic base classes

You will be wrong if you write as below:

class TimeKeeper {

public:

TimeKeeper();

~TimeKeeper();

...

};

class Clock: public TimeKeeper { ... };

...

TimeKeeper \*p = new Clock();

...

delete p;

The problem is that new Clock() returns a pointer to a derived class object, that object is being deleted via a base class pointer and the base class has a non-virtual destructor. What typically happens at runtime is that the derived part of the object is never destroyed simply because the destructor of the base class is called instead of the derived class.

Eliminating the problem is simple: give the base class a virtual destructor:

virtual ~TimeKeeper();

# Prefer pass-by-reference-to-const to pass-by-value

*Item 20 – Effective C++ 3rd Edition (Scott Meyers)*

**1. Reduce overheads and memory usage**

Consider the following code:

class Person {

public:

Person( );

virtual ~Person(); ...

private:

std::string name;

std::string address;

};

class Student: public Person {

public:

Student();

virtual ~Student( );

...

private:

std::string schoolName;

std::string schoolAddress;

};

bool validateStudent(Student s); // pass by value

void main() {

Student plato;

bool platoIsOK = validateStudent(plato);

}

What happens when validateStudent(plato) is called?

The Student copy constructor is called to initialize the parameter s from plato. Equally clearly, s is destroyed when validateStudent returns. So the parameter-passing cost of this function is one call to the Student copy constructor and one call to the Student destructor. But that’s not the whole story. A Student object has two string objects within it, so every time you construct a Student object you must also construct two string objects. A Student object also inherits from a Person object, so every time you construct a Student object you must also construct a Person object. A Person object has two additional string objects inside it, so each Person construction also entails two more string constructions. The end result is that passing a Student object by value leads to one call to the Student copy constructor, one call to the Person copy constructor, and four calls to the string copy constructor.

When the copy of the Student object is destroyed, each constructor call is matched by a destructor call, so the overall cost of passing a Student by value is six constructors and six destructors!

Now, this is correct and desirable behavior. After all, you want all your objects to be reliably initialized and destroyed. Still, it would be nice if there were a way to bypass all those constructions and destructions.

The solution here is using ***pass by reference-to-const***:

bool validateStudent(const Student& s);

This is much more efficient: **no constructors or destructors are called, because no new objects are being created**.

Note that when the Student is being passed by reference, it’s necessary to also declare it const, because otherwise callers would have to worry about validateStudent making changes to the Student they passed in.

**2. Avoid slicing problem**

When a derived class object is passed (by value) as a base class object, the base class copy constructor is called, and the specialized features that make the object behave like a derived class object are "sliced" off. This is almost never what you want.

Consider the following code:

class Window {

public:

...

std::string getName() const; // return name of window

virtual void display() const; // draw window and contents

};

class WindowWithScrollBars: public Window {

public:

...

virtual void display() const;

};

The fact that display is virtual tells you that the way in which the base class Window objects are displayed is apt to differ from the way in which the fancier WindowWithScrollBars objects are displayed

Now suppose you’d like to write a function to print out a window’s name and then display the window. Here’s the wrong way to write such a function:

void printNameAndDisplay(Window w) // incorrect parameter

{ // may be sliced!

std::cout << w.getName();

w.display( );

}

Consider what happens when you call this function with a WindowWithScrollBars object:

WindowWithScrollBars wwsb;

printNameAndDisplay(wwsb);

The parameter w will be constructed — it’s passed by value, remember? — as a Window object, and all the specialized information that made wwsb act like a WindowWithScrollBars object will be sliced off.

Inside printNameAndDisplay, w will always act like an object of class Window (because it is an object of class Window), regardless of the type of object passed to the function. In particular, the call to display inside printNameAndDisplay will always call Window::display, never WindowWithScrollBars::display.

The way around the slicing problem is to pass w by reference-to-const:

void printNameAndDisplay(const Window& w) // fine

{

std::cout << w.name();

w.display( );

}

**Note:**

If you peek under the hood of a C++ compiler, you’ll find that references are typically implemented as pointers, so passing something by reference usually means really passing a pointer. As a result, **if you have an object of a built-in type (e.g., an int), it’s often more efficient to pass it by value than by reference**. **This same advice applies to iterators and function objects in the STL**, because, by convention, they are designed to be passed by value. Implementers of iterators and function objects are responsible for seeing to it that they are efficient to copy and are not subject to the slicing problem.

# Rule of Five

The Rule of Three was a rule of thumb in C++. With the advent of C++11, it’s now broadened to the Rule of Five. These rules are related to following methods:

* *Destructor*
* *Copy constructor*
* *Copy assignment operator*
* *Move constructor (C++ 11)*
* *Move assignment operator (C++ 11)*

They are special member functions. If one of them is used without being defined by the programmer, it will be implicitly implemented by the compiler with the following default semantics:

* Destructor – Call the destructors of all the object's class-type members.
* Copy constructor – Construct all the object's members from the corresponding members of the copy constructor's argument, calling the copy constructors of the object's class-type members, and doing a plain assignment of all non-class type (e.g., int or pointer) data members.
* Copy assignment operator – Assign all the object's members from the corresponding members of the assignment operator's argument, calling the copy assignment operators of the object's class-type members, and doing a plain assignment of all non-class type (e.g. int or pointer) data members.

## What Does The Rule of Five Say?

**If we implement any of them, we should implement rest of them** to avoid possible (not always) exceptions.

So, the Rule of Five claims that **if one of these had to be defined by the programmer, it means that the compiler-generated version does not fit the needs of the class in one case and it will probably not fit in the other cases either**.

## Why Should Apply The Rule of Five?

**Avoid Possible Exceptions**

As said above.

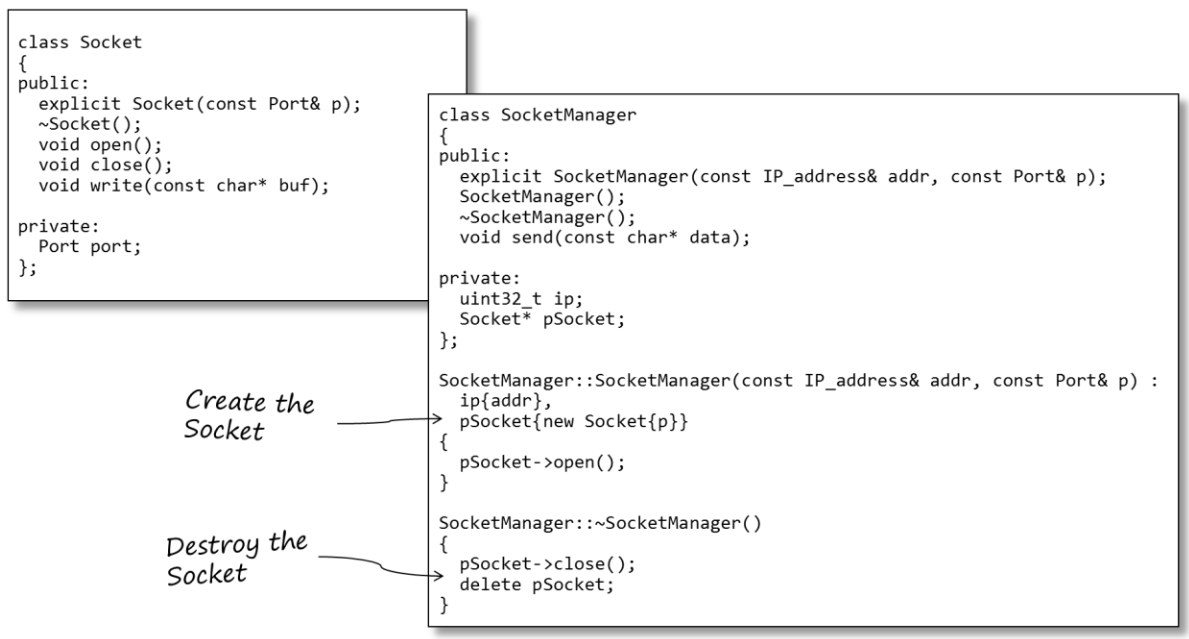
**Improve Performance**

Because compiler-generated version constructors and assignment operators simply copy all class data members by value ("shallow copy"), we should define explicit copy constructors and copy assignment operators for classes ("deep copy") that encapsulate complex data structures or have external references such as pointers.

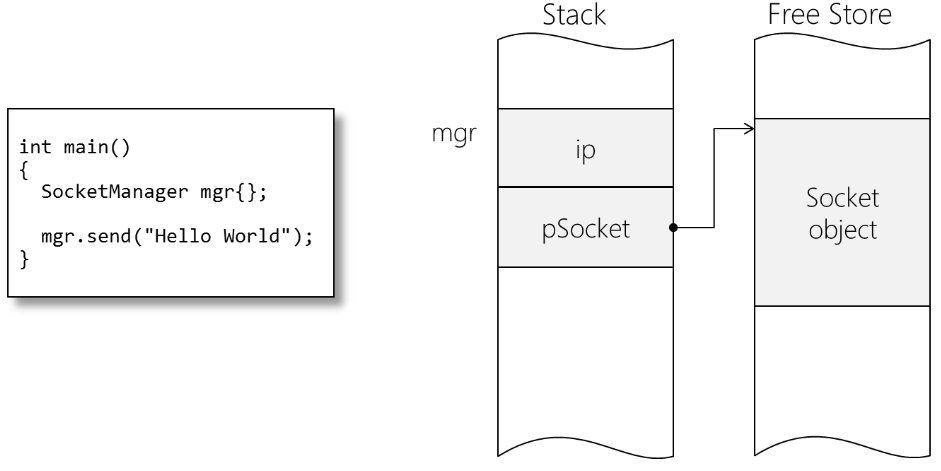
## Examples

### 1. Destructor

We use a SocketManager class to manage the lifetime of a Socket class. The Socket is allocated in the constructor and de-allocated in the destructor of SocketManager.



Here’s the memory layout for a SocketManager object:



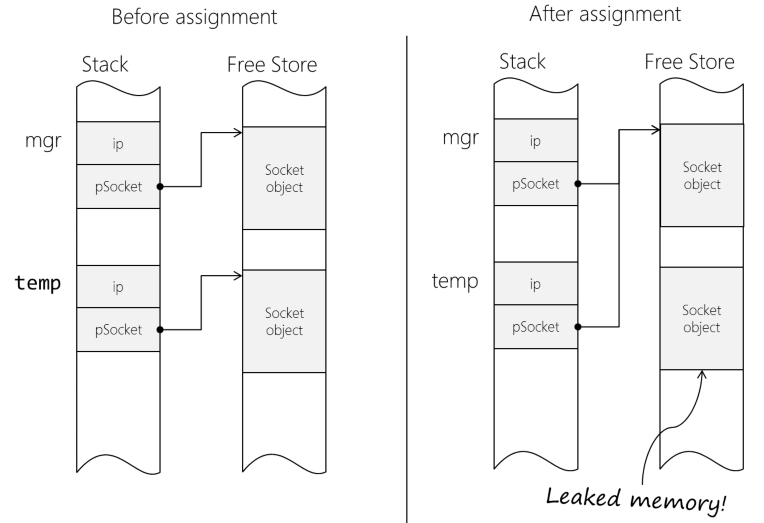
When mgr goes out of scope (in this case at the end of main), its destructor is called to help release the memory allocated for the Socket.

### 2. Copy Assignment Operator

A drawback of compiler-version default assignment operator is that it is a member-wise copy function, meaning each data member is copied in turn.

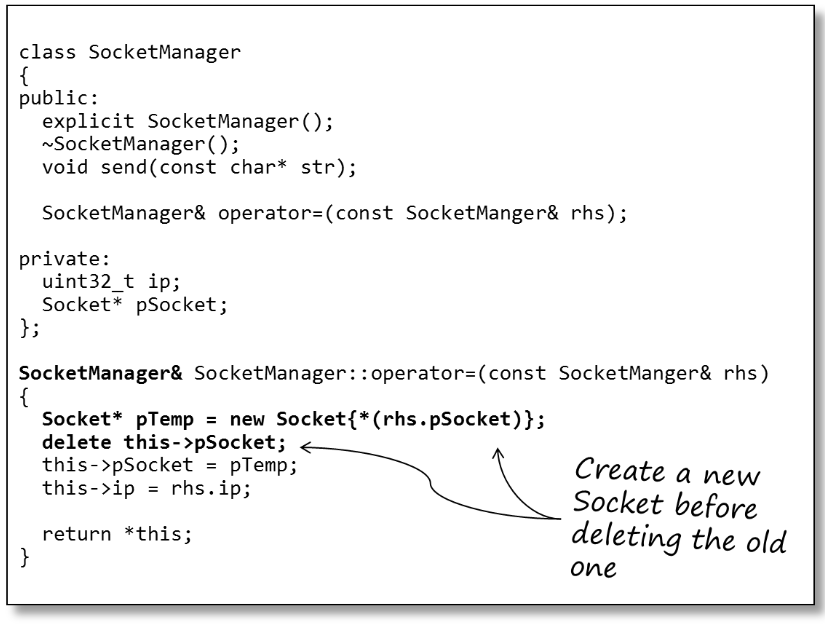
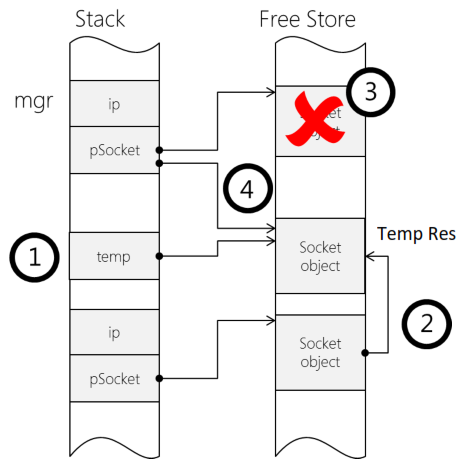


The problem is the creation of the temporary SocketManager object in func(). Since the default assignment operator copies mgr's pointer to its Socket into temp. When temp goes out of func(), it is destroyed. When mgr goes out of main(), it is destroyed too. However, that region of memory is already deleted by temp, so you will get a double-deleted error. Also, the region of memory allocated by temp is now unmanaged, which cause memory leaks.



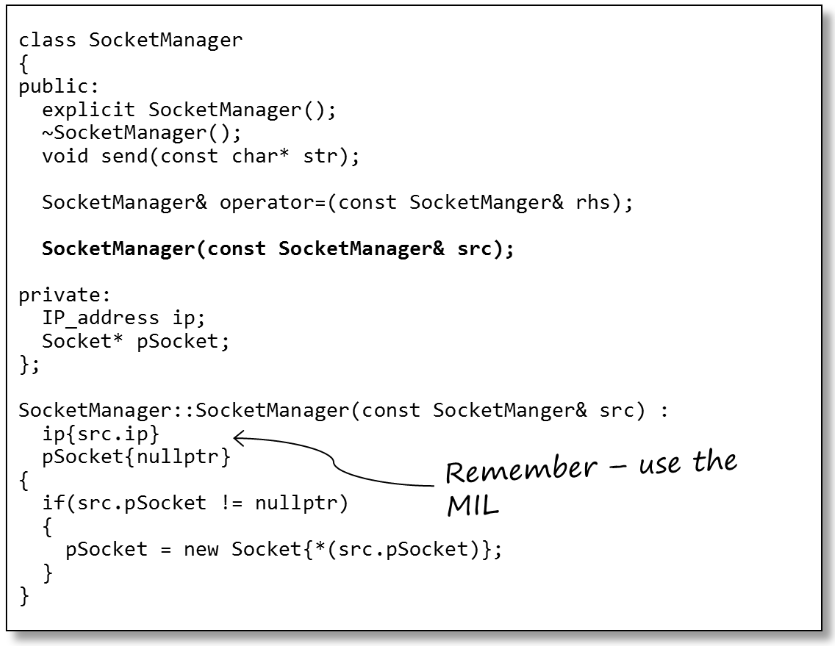
To eliminate this problem, we will **manually define our own copy assignment operator**. The implementation is as follows:

1. A new resource (Socket in this case) is allocated.
2. The contents of the right-hand-side's resource are copied into the temporary resource (which may incur another deep copy itself!)
3. We delete the old (no longer needed) resource and free up its memory
4. We take ownership of the temporary resource. It is now owned and managed by us.



### 3. Copy Constructor

Same problem – the compiler-version copy constructor also does a member-wise copy. So, again we **define our own copy constructor to override the compiler-version one**.

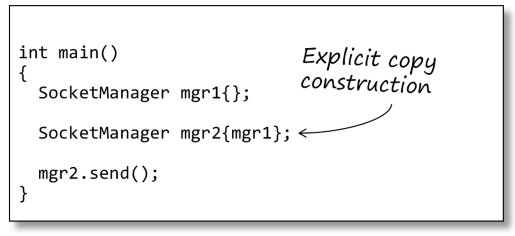


In case of the copy constructor, we can be certain the SocketManager being created has no resource; so we don't need (and never should try) to delete it.

But when the copy constructor is called? Unlike the assignment operator, the copy constructor is often called ‘invisibly’ by the compiler. Here are the four scenarios the copy constructor is invoked:

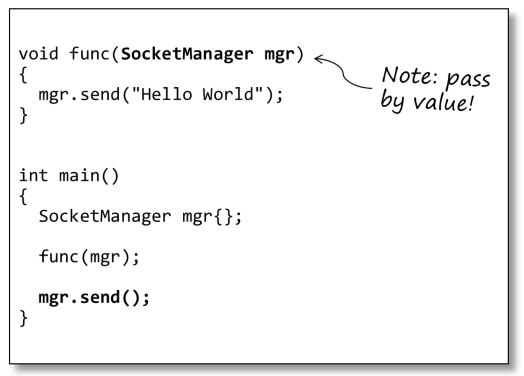
**a) Pass-by-reference**

Pass in another object as a parameter.



**b) Pass-by-value**

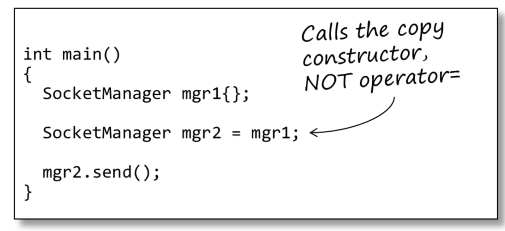
When objects are passed to functions by value a copy of the caller’s object is made. This new object has a constructor called, in this case the copy constructor.



This extra overhead (memory + constructor call time) is why it’s always better to pass objects to functions by reference.

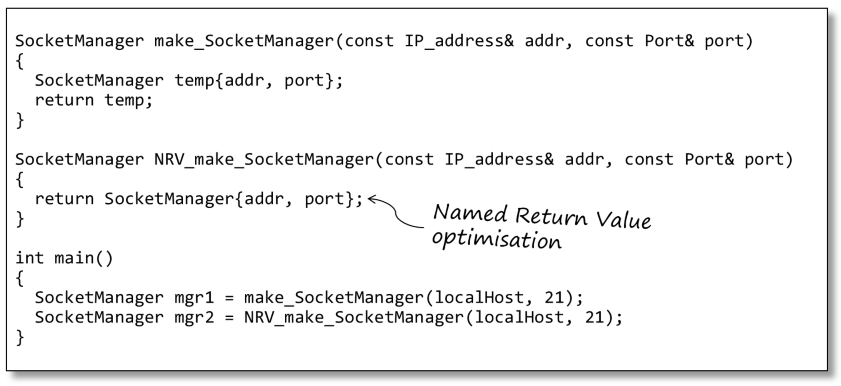
**c) Object initialization**

C++ makes the distinction between initialization and assignment. If an object is being initialized, the compiler will call a constructor, rather than the assignment operator.



**d) Function return**

If a function returns an object from a function (by value), then a copy of the object is made. The copy constructor is invoked on return.

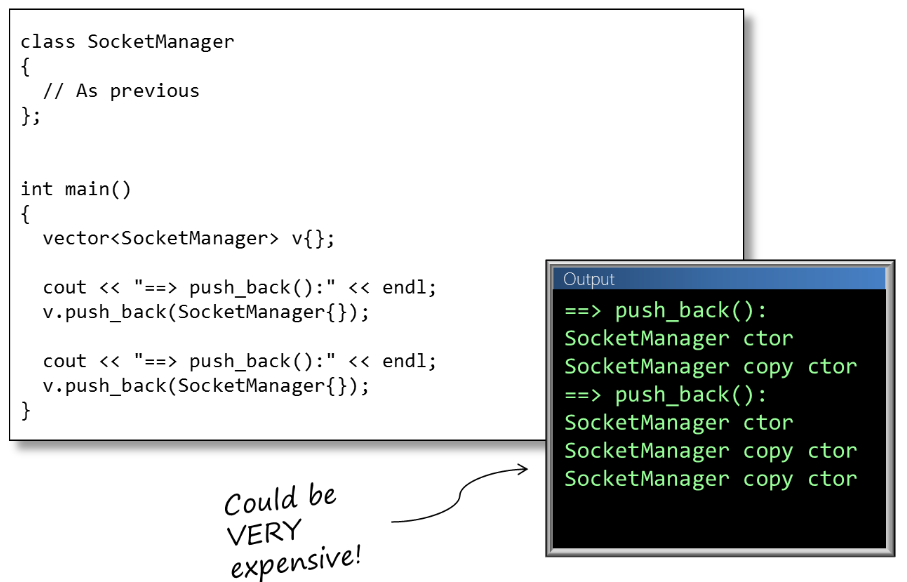


There are two exceptions to this:

1. If the object has a move constructor defined (see below) then that will be called in preference.
2. If the return object is constructed directly as part of the return statement (as in NRV\_make\_SocketManager() above) the compiler can optimize this and construct the return object directly into the callers object. In this case, a ‘normal’ constructor is called rather than a copy constructor. This mechanism is referred to as the *Named Return Value (NRV)* optimization.

### 4. Move Constructor

Copying objects may not be the best solution in many situations. It can be very expensive in terms of creating, copying and then destroying temporary objects. For example:

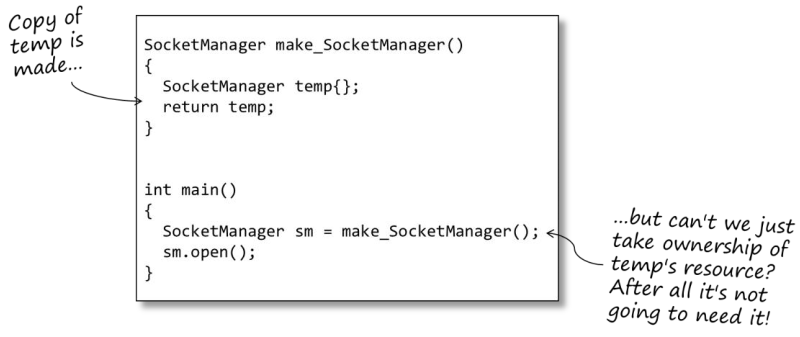


The second copy constructor is caused by the vector memory allocator creating space for two new objects then copying the objects across (since the vector must keep the elements contiguous). This is an expensive operation since every copy requires de-allocation / re-allocation of memory and copying of contents.

In such cases, we typically just want to transfer data from one object to another, rather than make an (expensive, and then discarded) copy.

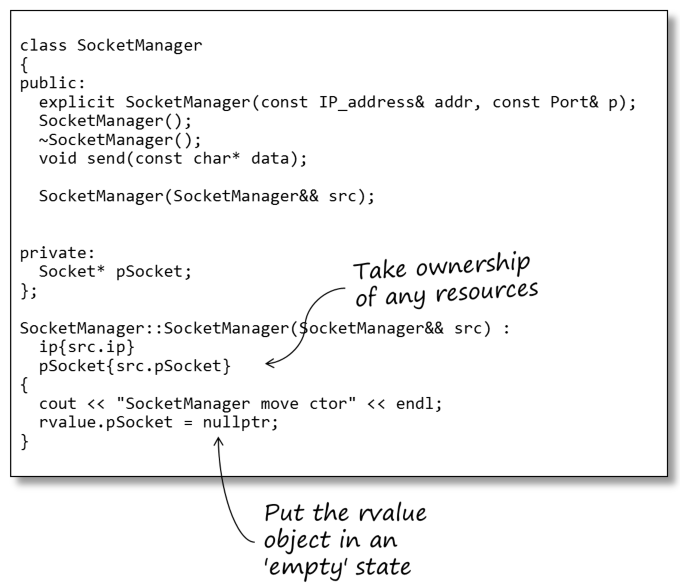
C++98 favors copying. When temporary objects are created, for example, they are copied (using the copy constructor). In a lot of cases, copying is an unnecessary overhead since the temporary is going out of scope and can't use its resources anymore.

It would be nicer if we could **'take ownership' of the temporary's resources**, freeing it of the burden of de-allocating the resource, and freeing us of the burden of re-allocating. This is sometimes known as 'resource pilfering'.



The move constructor:

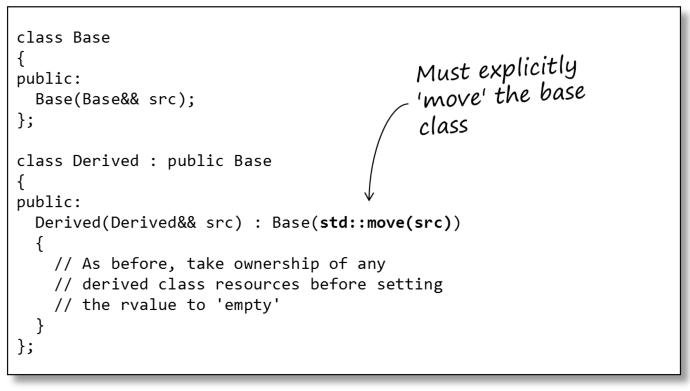
* Takes an r-value reference as a parameter.
* Discards the object’s current state.
* Transfers ownership of the r-value object into the receiver
* Puts the r-value object into an 'empty' state.



**Note**: The parameter for the move constructor is not const - we need to modify the parameter.

The move constructor 'claims' the resource of the supplied r-value. By setting the r-value pSocket to nullptr, when the r-value object goes out of scope, its destructor will do nothing.

**Note**: There are some things to be aware of if you want to include move semantics with derived classes. If you want to use the base class move semantics from the derived class you must explicitly invoke it; otherwise the copy constructor will be called.

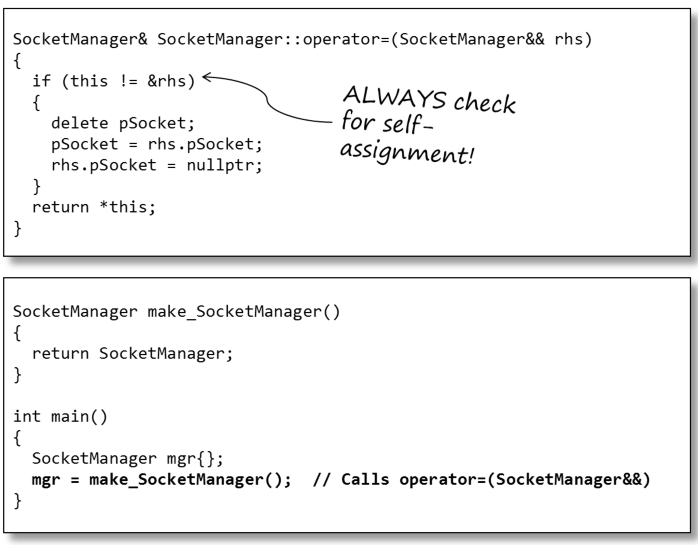


std::move doesn't actually do any moving; it just converts an l-value into an r-value. This forces the

compiler to use the object's move constructor (if defined) rather than the copy constructor.

### 5. Move Assignment Operator

Occasionally, it is useful to only have one resource at a time and to transfer ownership of that resource from one manager to another (this is what std::unique\_ptr does). In such cases, you may want to provide a move assignment operator.



The assignment operator must always check for self-assignment. Although this is extremely rare in hand-written code certain algorithms (e.g.: std::sort) may make such assignments.

Note the difference between this code and the Object Initialization example (above). In this case, since mgr has already been initialized the line mgr = make\_SocketManager() is performing an assignment.

Also, realize that the original object is left in an 'empty' state, so attempting to use it could result in some unpleasant surprises. Be careful with explicitly moving objects in your application code; make sure they won’t be used any more.

# Always check null pointer

In any case, checking whether a pointer is null before using it is always a good practice. Some of these cases include:

**1. Pointer as a function parameter**

The following code **avoids crash** in case inFilePath is null.

bool ReadFile(char\* *inFilePath*)

{

if (*inFilePath* == nullptr) {

// Can show error log here

return false;

}

// Code to read file...

}

**2. Pointer as an object**

The following code **avoids crash** in cased the file is not loaded successfully:

FILE\* pFile = LoadFileInfo(...);

if (pFile != nullptr) {

pFile->GetName();

}

**3. Pointer used to allocate memory**

The following code **avoids re-allocating memory** for the same pointer, which might cause wrong behaviors or even crashes:

bool CAutoSplice::CreateDataIF()

{

// Do not allocate memory if it already exists

if (m\_data != nullptr) {

CDataIF\* data = new CDataIF();

m\_data = dynamic\_cast<CBaseDataIF\*>(data);

}

return m\_data != nullptr;

}

# Prefer smart pointers

For why, check *C++ Tutorial*, chapter *Smart Pointers*, header *Advantages of Smart Pointers Over Raw Pointers*.

# Cannot return a local object by pointer or reference

**Example 1:**

char\* GetName() {

char name[10] = "TriHo"; // local variable

return name; // return by pointer

}

int main() {

const char\* n = GetName(); // WRONG: random characters, not "TriHo"

return 0;

}

The n will now point to the local name. But where is the name after GetName() is executed? Nowhere. It was located on the stack, but after the function returned, the stack was unwrapped and all local variables and objects from the function were deleted. This will eventually result in an undefined behavior, even for primitive types.

Here is the way to fix this issue:

std::string GetName() {

char name[10] = "TriHo"; // local variable

std::string nameStr(name);

return nameStr; // neither pointer nor reference

}

int main() {

std::string n = GetName(); // CORRECT: "TriHo"

return 0;

}

**Example 2:**

Complex& SumComplex(const Complex& a, const Complex& b) {

Complex result(a.real + b.real, a.imaginar + b.imaginar);

return result; // return an object by reference

}

int main() {

// input a and b

Complex& sum = SumComplex(a, b); // WRONG sum

return 0;

}

Here is the way to fix it:

Complex SumComplex(const Complex& a, const Complex& b) {

return Complex(a.real + b.real, a.imaginar + b.imaginar);

}

int main() {

// input a and b

Complex sum = SumComplex(a, b); // CORRECT

return 0;

}

For most of today’s compilers, if a return line contains a constructor of an object, the code will be optimized to avoid all unnecessary copying - the constructor will be executed directly on the sum object.