C++11: The Rule of the Big Five

Resource Management

The dynamic creation and destruction of objects was always one of the bugbears of C. It required the programmer to (manually) control the allocation of memory for the object, handle the object's initialisation, then ensure that the object was safely cleaned-up after use and its memory returned to the heap. Because many C programmers weren't educated in the potential problems (or were just plain lazy or delinquent in their programming) C got a reputation in some quarters for being an unsafe, memory-leaking language.

C++ improved matters significantly by introducing an idiom known (snappily) as RAII/RRID – Resource Acquisition Is Initialisation / Resource Release Is Destruction*. The idiom makes use of the fact that every time an object is created a constructor is called; and when that object goes out of scope a destructor is called. The constructor/destructor pair can be used to create an object that automatically allocates and initialises another object (known as the *managed* object) and cleans up the managed object when it (the manager) goes out of scope. This mechanism is generically referred to as resource management. A resource could be any object that required dynamic creation/deletion – memory, files, sockets, mutexes, etc.

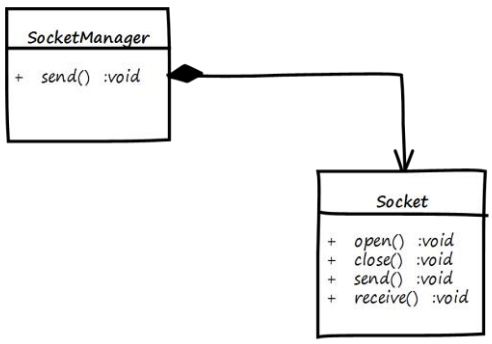
Resource management frees the client from having to worry about the lifetime of the managed object, potentially eliminating memory leaks and other problems in C++ code.

However, RAII/RRID doesn't come without cost (to be fair, what does?) Introducing a 'manager' object can lead to potential problems – particularly if the 'manager' class is passed around the system (it is just another object, after all). Copying a 'manager' object is the first issue. Cline, Lomow and Girou[1] coined the phrase "The Rule of The Big Three" to highlight the issues of attempting to copy a resource-managing object. However, in C++11 the concept of *move semantics* was added to the language. This adds another aspect of complication; and leads to "The Rule of The Big Five"**

The Rule of The Big Five states that if you have to write *one* of the functions (below) then you have to have a policy for *all* of them.

1 – The destructor

For our example we will use a `SocketManager` class that owns (manages) the lifetime of a `Socket` class.

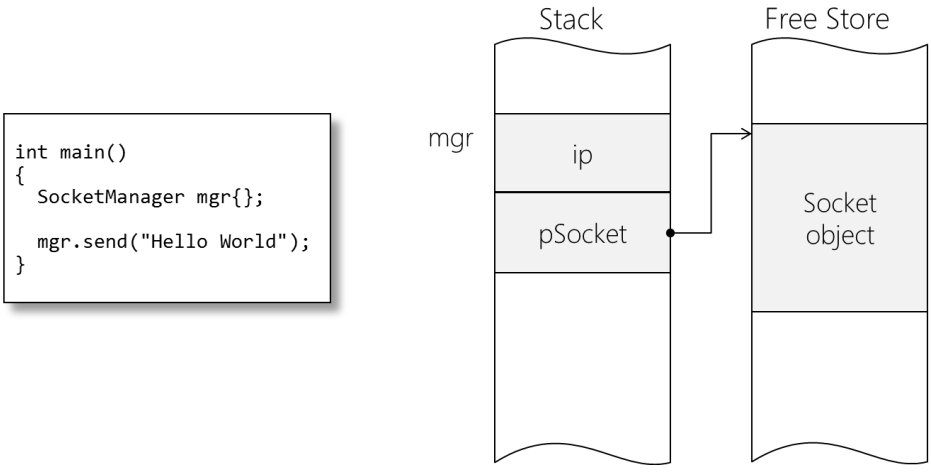


The `SocketManager` is responsible for the lifetime of its `Socket` object. The `Socket` is allocated in the `SocketManager` constructor; and de-allocated in the destructor.

A small warning here: make sure the new and delete operators 'match': that is, if the resource is allocated with `new`, then use `delete`; if the resource is allocated as an array (`new[]`) make sure array delete is used (`delete[]`) Failure to do so will lead to 'Bad Things' happening.

Also, if your resource manager class is going to be used polymorphically (that is, in an inheritance hierarchy) it is good practice to make the destructor virtual. This will ensure that any derived class constructors, if defined, will be called even if the client has a pointer to a (more) base class.

Here's the memory layout for a `SocketManager` object:



When `mgr` (above) goes out of scope (in this case at the end of `main`; but in general at the end of the enclosing block) its destructor will be automatically called. The destructor calls `delete` on the `pSocket` pointer, which automatically calls the `Socket`'s destructor *before* releasing the memory for the `Socket`.

2 – The assignment operator

It is possible to assign two objects of like type. If you do not provide one the compiler creates a default assignment operator. The default assignment operation is a member-wise copy function; each data member is copied in turn.

At first glance the code below appears sound:

```
void func(const SocketManager& mgr)
{
  SocketManager temp{};
  temp = mgr;
  temp.send("Hello World");
}


int main()
{
  SocketManager mgr{};

  func(mgr);

  mgr.send();
}
```

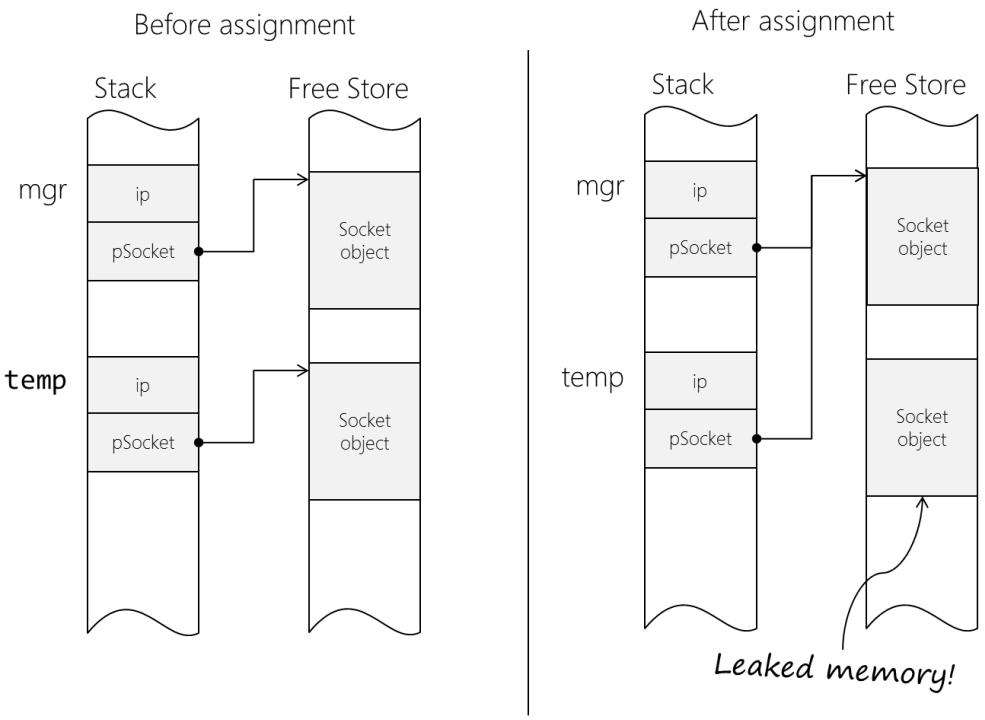
The problem is the creation of the temporary `SocketManager` object in `func()`.

Since the default assignment operator performs a member-wise copy this means `mgr`'s pointer to its `Socket` is copied into `temp`. When `temp` goes out of scope (at the end of `func()`) it is destroyed and it deletes its pointer – just as it should.

When `mgr` goes out of scope at the end of `main()` it, too, tries to delete its pointer. However, that region of memory has already be deleted (by `temp`) so you will get a run-time error!

In addition, any attempt to use `mgr` after the call to `func()` will result in undefined (and almost certainly fatal) behaviour.

Here we see the memory maps, showing what happens during `func()`.

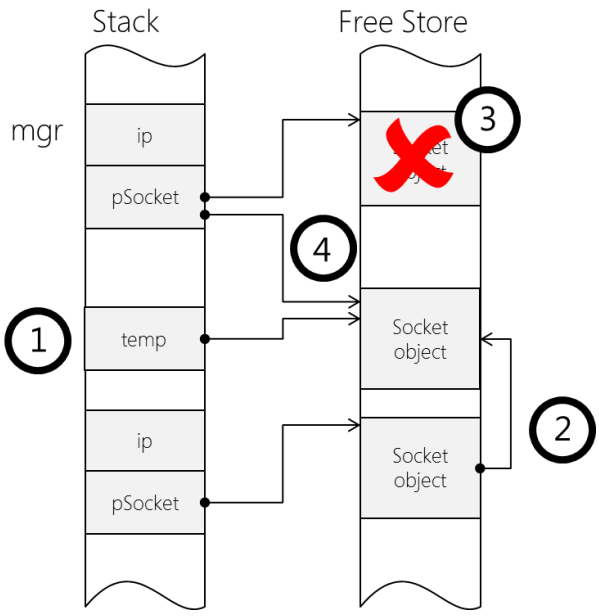


At the beginning of `func()` a temporary object is created, with its own `Socket`. `mgr` also has its own `Socket`.

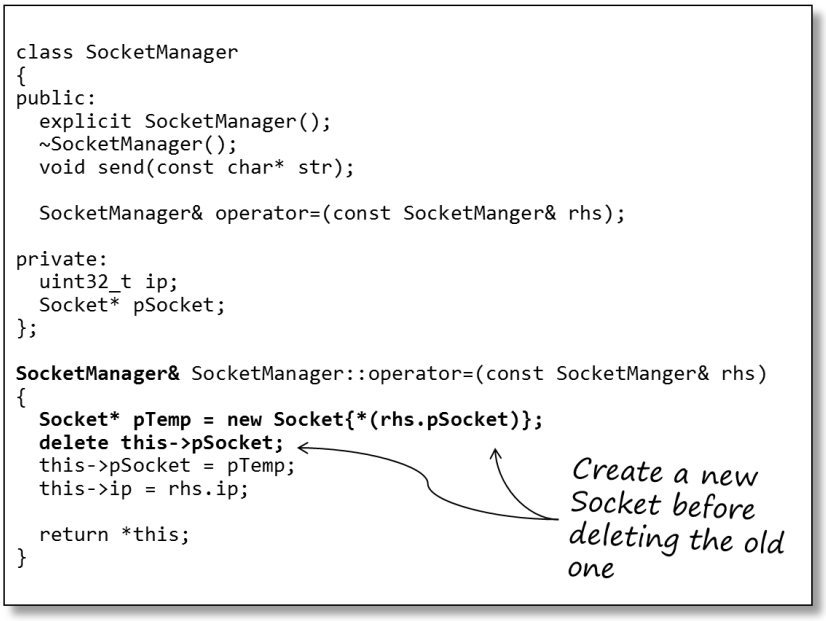
The default assignment does a member-wise copy of all the elements of `mgr`. At this point we get our first problem - a memory leak! `temp`'s original pointer to its `Socket` has been over-written, meaning we can never delete that memory.

Second, when `temp` goes out of scope it will delete whatever its `pSocket` pointer is addressing - which is the same memory owned by `mgr`.

To eliminate this problem we must write our own, more sophisticated, assignment operator. The implementation of the assignment operator proceeds as follows:



1. A new resource (a `Socket` in this case) is allocated. We do this before deleting our old resource just in case the allocation fails - we'd be left with a 'broken' object.
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.



Our assignment operator function now implements the correct class copy semantics ("deep copy"). In this example I've used the (more explicit) `this->` notation to specify the receiving object; and `src` for the source object. This is purely for clarity and is not required.

The code presented here is exception safe because the request for more memory is made before the existing memory is deleted.

The assignment operator returns a reference to itself. This is so expressions like this work:

```
mgr1 = mgr2 = mgr3;
```

There is a special case of copy to self. This code works, although the storage for the string will move, and it is wasteful. A test for `this != &src` could be added, but how often does this happen? The code to trap the special case becomes an overhead on the general case.

3 – The copy constructor

The compiler-supplied copy constructor does a member-wise copy of all the `SocketManager`'s attributes. Of course, this is the same situation as previously.

Again, in this situation we require a deep-copy of the `SocketManager`'s attributes.

You can provide your own copy constructor, which overrides the compiler-supplied one. Note the signature of the copy constructor – it takes a reference to a `const SocketManager` object.

```
class SocketManager
{
public:
  explicit SocketManager();
  ~SocketManager();
  void send(const char* str);

  SocketManager& operator=(const SocketManger& rhs);

  SocketManager(const SocketManager& src);

private:
  IP_address ip;
  Socket* pSocket;
};

SocketManager::SocketManager(const SocketManger& src) :
  ip{src.ip}
  pSocket{nullptr}
{
  if(src.pSocket != nullptr)
  {
    pSocket = new Socket{*(src.pSocket)};
  }
}
```

Remember – use the MIL

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

Notice that we check if the source object actually has a resource allocated, otherwise we'll get a run-time fault when we try and copy from an uninitialised object.

So when in your code will the copy constructor be 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:

1 – Explicit copy construction

The most explicit way to invoke the copy constructor on an object is to create said object, passing in another object (of the same type) as a parameter.

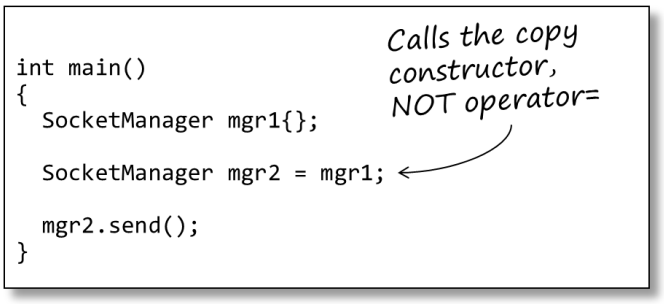
```
int main()
{
  SocketManager mgr1{};

  SocketManager mgr2{mgr1};

  mgr2.send();
}
```

Explicit copy construction

This is not a particularly common way to construct objects; compared with below.

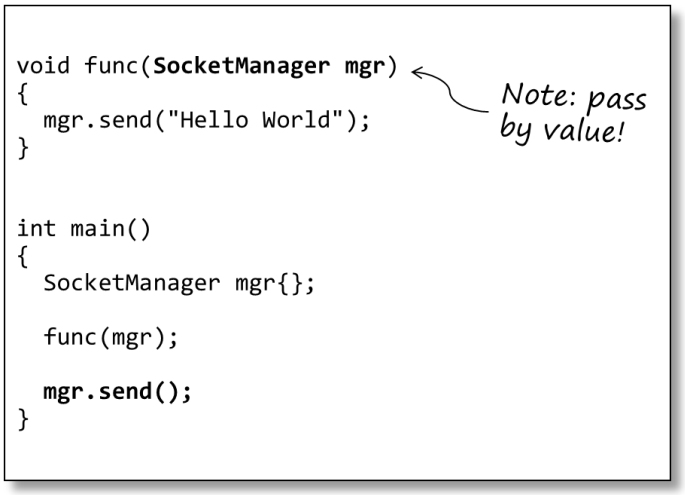
2 – Object initialisation

C++ makes the distinction between *initialisation* and *assignment*. If an object is being initialised the compiler will call a constructor, rather than the assignment operator. In the code below the copy constructor for `mgr2` is called:



3 - Pass-by-value parameters

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.

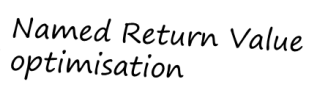


This extra overhead (memory + constructor call time) is why it's always better to pass objects to functions by reference.

4 – Function return value

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.

```
SocketManager make_SocketManager(const IP_address& addr, const Port& port)
{
  SocketManager temp{addr, port};
  return temp;
}

SocketManager NRV_make_SocketManager(const IP_address& addr, const Port& port)
{
  return SocketManager{addr, port};
}

int main()
{
  SocketManager mgr1 = make_SocketManager(localHost, 21);
  SocketManager mgr2 = NRV_make_SocketManager(localHost, 21);
}
```



There are two exceptions to this:

If the object has a move constructor defined (see below) then that will be called in preference.

If the return object is constructed directly as part of the return statement (as in `NRV_make_SocketManager()`, above) the compiler can optimise 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) optimisation.

4 – The 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:

- Returning objects from functions
- Some algorithms (for example, swap)
- Dynamically-allocating containers.

Let's take a vector of `SocketManager`s as an example:

```
class SocketManager
{
  // As previous
};


int main()
{
  vector<SocketManager> v{};

  cout << "==> push_back():" << endl;
  v.push_back(SocketManager{});

  cout << "==> push_back():" << endl;
  v.push_back(SocketManager{});
}
```

```
Output
==> push_back():
SocketManager ctor
SocketManager copy ctor
==> push_back():
SocketManager ctor
SocketManager copy ctor
SocketManager copy ctor
```

Could be VERY expensive!

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 favours 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 just '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'.

Copy of temp is made...

```
SocketManager make_SocketManager()
{
  SocketManager temp{};
  return temp;
}


int main()
{
  SocketManager sm = make_SocketManager();
  sm.open();
}
```

...but can't we just take ownership of temp's resource? After all it's not going to need it!

What we need is some way of distinguishing between temporary objects (that are going to go out of scope soon) and objects that will persist beyond the end of the statement.

C++11 introduced the concept of the *r-value reference* for such circumstances.

An r-value reference can be explicitly bound to an r-value. An r-value is an *unnamed object*. A temporary object, in other words. (For a much more detailed description of l-value and r-value objects, see http://blog.feabhas.com/2013/02/l-values-r-values-expressions-and-types/)

The r-value reference, while not in and of itself particularly useful (like the l-value reference, actually), can be used to overload functions to respond differently depending on whether a parameter is an l-value or an r-value type, giving different behaviour in each case.

```
int func(int a, int b){ return a * b;}

void byRef(int& i)        { /* … */ }
void byRef(const int& i)  { /* … */ }
void byRef(int&& i)       { /* … */ }
```


```
int main()
{
  int&& rval = func(10, 20);  // OK: rval is modifiable r-value reference
  int i = 100;

  byRef(i);                   // calls byRef(int&)
  byRef(rval);                // Calls byRef(int&&)
  byRef(func(10, 10));        // Calls byRef(int&&) with modifiable r-value
  byRef(17);                  // Calls byRef(const int&)
}
```

The compiler only favours r-value reference overloads for modifiable r-values; for constant r-values it always prefers constant l-value references (This means overloading for `const T&&` has no real application)

As we can now distinguish between l-value and r-value objects we can overload the constructor (and, later, assignment operator) to support 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.

```
class SocketManager
{
public:
  explicit SocketManager(const IP_address& addr, const Port& p);
  SocketManager();
  ~SocketManager();
  void send(const char* data);

  SocketManager(SocketManager&& src);


private:
  Socket* pSocket;
};

SocketManager::SocketManager(SocketManager&& src) :
  ip{src.ip}
  pSocket{src.pSocket}
{
  cout << "SocketManager move ctor" << endl;
  rvalue.pSocket = nullptr;
}
```

Take ownership of any resources

Put the rvalue object in 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.

Returning to our vector problem: with the move constructor in place the `SocketManager` objects are moved rather than copied. This code could be significantly more efficient, if there is a lot of insertion in the vector.

```
class SocketManager
{
  // As previous
};


int main()
{
  vector<SocketManager> v{};

  cout << "==> push_back():" << endl;
  v.push_back(SocketManager{});

  cout << "==> push_back():" << endl;
  v.push_back(SocketManager{});
}
```

Output

```
==> push_back():
SocketManager ctor
SocketManager move ctor
==> push_back():
SocketManager ctor
SocketManager move ctor
SocketManager move ctor
```

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.

```
class Base
{
public:
  Base(Base&& src);
};

class Derived : public Base
{
public:
  Derived(Derived&& src) : Base(std::move(src))
  {
    // As before, take ownership of any
    // derived class resources before setting
    // the rvalue to 'empty'
  }
};
```

Must explicitly 'move' the base class

`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 – The 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 (for example, this is what `std::unique_ptr` does). In such cases you may want to provide a move assignment operator.

```
SocketManager& SocketManager::operator=(SocketManager&& rhs)
{
  if (this != &rhs)
  {
    delete pSocket;
    pSocket = rhs.pSocket;
    rhs.pSocket = nullptr;
  }
  return *this;
}
```

ALWAYS check for self-assignment!

```
SocketManager make_SocketManager()
{
  return SocketManager;
}

int main()
{
  SocketManager mgr{};
  mgr = make_SocketManager();  // Calls operator=(SocketManager&&)
}
```

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

Note the difference between this code and the *Object Initialisation* example (above). In this case, since `mgr` has already been initialised the line `mgr = make_SocketManager()` is performing an *assignment*.

Also, realise 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.

Your resource management policy

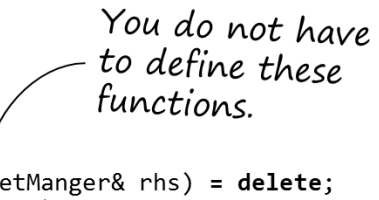
The Rule of The Big Five says if you've written one of the above functions then you must have a policy about the others. It doesn't say you *have to write them*. In fact, you have to have a resource management policy for every class you create. Your policy can be one of the following:

- Use the compiler-provided versions of these functions (well, the first three, anyway; the compiler doesn't automatically provide a move constructor and move-assignment operator). In other words, you're not doing any resource management in the class.
- Write your own copy functions to perform deep copy, but don't provide move semantics.
- Write your own move functions, but don't support copying.
- Disable copying and move semantics for the class, because it doesn't make sense to allow it.

Suppressing move and copy is straightforward; and there are two ways to do it:

- Make the appropriate function declarations private. (C++98)
- Use the `=delete` notation (C++11)

```
class SocketManager
{
public:
  explicit SocketManager();
  ~SocketManager();
  void send(const char* str);

  // Copy policy
  //
  SocketManager& operator=(const SocketManger& rhs) = delete;
  SocketManager(const SocketManager& src)           = delete;

  // Move policy
  //
  SocketManager& operator=(SocketManager&& rhs)     = delete;
  SocketManager(SocketManager& src)                 = delete;


  uint32_t ip;
  Socket* pSocket;
};
```



Conclusion

Resource management – making use of C++'s RAII/RRID mechanism – is a key practice for building efficient, maintainable, bug-free code. Defining your copy and move policy for each type in your system is a key part of the software design. The Rule of The Big Five exists as an aide memoir for your copy/move policy. Feel free to ignore it; but do so at your peril.

Finally, examining move and copy policies also leads to two supplemental good practices:

- A class should only manage at most one resource.
- Simplify your types by making use of extant types that perform resource management for you, such as 'smart pointers'.

**Bjarne Stroustrup joked once that he should never, ever work in marketing, after coming up with an unpronounceable acronym like RAII/RRID!*

***With apologies to Messer's Cline, Lomow and Girou for the blatant rip-off of their phrase. Please take it as a measure of the high esteem I hold their rule in.*

References

1 *C++ FAQs 2ND Edition*
Cline, Lomow, Girou
Addison-Wesley Professional (December 21, 1998)
ISBN-13: 978-0201309836