SECTION 02
Move semantics

© Feabhas Ltd 2012

Copying may be expensive

Returning objects from functions

Some algorithms (e.g. swap)

Dynamically-allocating containers

Copying objects may not be the best solution in many situations. In such cases we typically just want to transfer data from one object to another, rather than make an (expensive, and then discarded) copy.

© Feabhas Ltd 2012

Resource management classes

```
class SocketManager
{
public:
  SocketManager();
  ~SocketManager();

  // Deep-copy semantics
  SocketManager& operator=(const SocketManger& rhs);
  SocketManager(const SocketManager& src);

private:
  Socket* pSocket;
};

SocketManager::SocketManager() : pSocket(new Socket)
{
  cout << "SocketManager ctor" << endl;
  pSocket->open();
}

SocketManager::~SocketManager()
{
  pSocket->close();
  delete pSocket;
}
```

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.

The SocketManager class implements both the copy constructor and assignment operator, providing a 'deep copy' policy.

© Feabhas Ltd 2012

The cost of copy constructors

```
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());
}
```

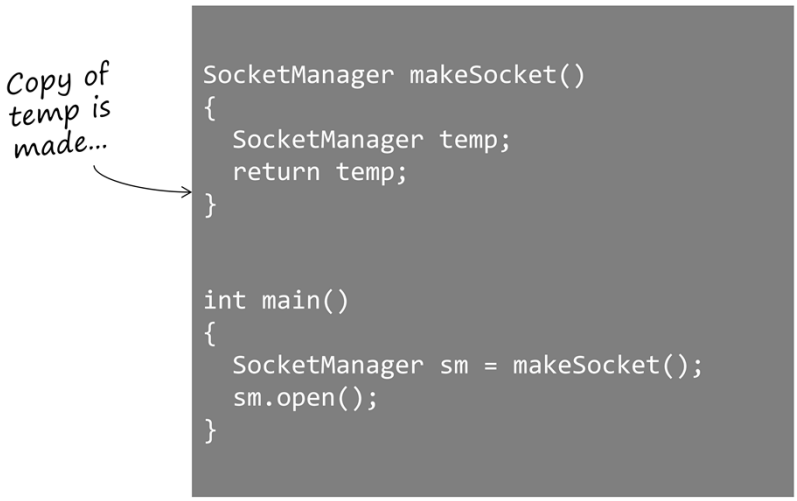
Output

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

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

© Feabhas Ltd 2012

Resource pilfering, not copying



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'.

© Feabhas Ltd 2012

lvalues and rvalues

lvalue and rvalue is a property of *expressions*, not objects

lvalues persist beyond a single expression - In other words, *named objects*

rvalues do not persist beyond their statement - In other words, *unnamed objects*

- Temporary values
- Objects returned from functions

lvalues and rvalues may be modifiable or non-modifiable

The simple test for determining whether an expression returns an lvalue or rvalue is: Can I take the address of the object? If you can, it's an lvalue; otherwise it's an rvalue

For example:

```
&*ptr
&array[i]
&++x
```

are all valid (if not particularly useful); whereas:

```
&(x + y)
&x++
&17.6
```

are not valid.

A function call is deemed to be an lvalue expression only if it returns a reference

© Feabhas Ltd 2012

lvalue references

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

void byRef(int& i) { /* … */}

void byRef(const int& i) { /* … */}

int main()
{
  int i = 10;
  int& r1 = i;                // OK: Can bind to lvalue
  const int& r2 = 17;         // OK: Can bind to a literal

  int& a = func(10, 20);      // ERROR: cannot bind lvalue reference
                              // to return object.

  const int& b = func(10,20); // OK: Return value converted to const int&

  byRef(r1);                  // Calls byRef(int&)
  byRef(func(10, 10));        // Calls byRef(const int&) with rvalue
}
```

C++98 has the concept of the reference (referred to as an lvalue reference in C++11).

An lvalue reference can be bound to a modifiable object; but not to an rvalue or a constant object

a const-reference can be bound to an lvalue or an rvalue.

© Feabhas Ltd 2012

rvalue references

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

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


int main()
{
  int&& rval = func(10, 20);  // OK: rval is modifiable rvalue reference

  byRef(rval);                // Calls byRef(int&&)
  byRef(func(10, 10));        // Calls byRef(int&&) with modifiable rvalue
  byRef(17);                  // Calls byRef(const int&)
}
```

An rvalue reference can be explicitly bound to an rvalue.

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

The compiler only favours rvalue reference overloads for modifiable rvalues; for constant rvalues it always prefers constant-references (This means overloading for const T&& has no real application)

Rather confusingly an rvalue reference, since it is a named object, is actually an lvalue!

© Feabhas Ltd 2012

Move constructors

A move constructor takes an rvalue reference as the parameter.

Discards the object's current state (if it exists)

Transfers the ownership of the rvalue resources into the receiver.

Puts the rvalue object into an 'empty' state.

As we can now distinguish between lvalue and rvalue objects we can overload the constuctor (and, later, assignment operator) to support resource pilfering.

© Feabhas Ltd 2012

Transferring ownership

```
class SocketManager
{
public:
  SocketManager();
  ~SocketManager();
  SocketManager& operator=(const SocketManger& rhs);
  SocketManager(const SocketManager& src);

  SocketManager(SocketManager&& rvalue);

  void send(const char* str);

private:
  Socket* pSocket;
};

SocketManager::SocketManager(SocketManager&& rvalue) :
  pSocket(rvalue.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 rvalue. By setting the rvalue pSocket to nullptr, when the rvalue object goes out of scope its destructor will do nothing.

© Feabhas Ltd 2012

The container problem, revisited

```
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
```

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.

© Feabhas Ltd 2012

Move assignment

```
SocketManager& SocketManager::operator=(SocketManager&& rhs)
{
  if (this != &rhs)          // ALWAYS check for self-assignment.
  {
    delete pSocket;
    pSocket = rhs.Socket;

    rhs.ptr = nullptr;       // Again, set the rhs to an 'empty'
  }                          // state.
  return *this;
}
```

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

int main()
{
  SocketManager s;
  s = makeSocket();  // Calls operator=(SocketManager&&)
}
```

The assignment operator can also be overloaded for rvalue references.

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.

(As an aside the assignment operator is considered an lvalue since it returns a reference to an existing object (*this))

© Feabhas Ltd 2012

rvalue references to const objects

```
const SocketManager makeSocket()
{
  return SocketManager;  // Overload resoultion rules will force
                         // the compiler to invoke SocketManager's
                         // copy constructor, not its move constructor
}

int main()
{
  const SocketManager s = makeSocket();
}
```

Be careful - returning a const value from a function will result in the copy constructor being called, even if you have a move constructor defined

(NB: Visual Studio 11 appears to call the move constructor rather than the copy constructor; which it shouldn't)

© Feabhas Ltd 2012

Moving lvalue objects

```
class SocketManager
{
  // As before…
};

int main()
{
  vector<SocketManager> v;

  SocketManager s;
  v.push_back(std::move(s));  // Invoke move constructor

  v[0].send("Hello");         // All future access via vector
  v[0].send("World");
}
```

`std::move` doesn't *actually* do any moving, it converts an lvalue into an rvalue. This forces the compiler to use the object's move constructor (if defined) rather than the copy constructor.

Use `std::move` if the lvalue object is going to be discarded immediately - either it won't be used again, or it is going out of scope.

© Feabhas Ltd 2012

Moving derived classes

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


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

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.

The same is true with derived class assignment operators.

© Feabhas Ltd 2012

Key points

Excessive copying of objects - particularly those that are own resource - can be unnecessarily expensive

rvalue references allow the implementation of move constructors

A move constructor 'takes ownership' of an rvalue's resources and leaves it in an 'empty' state.

Move semantics may also be applied to assignment operators on a class, too.

std::move allows move semantics to be applied to lvalue objects

© Feabhas Ltd 2012