semantics

Copying, Conversions, and Temporaries

Roadmap

Assignment and initialization
Copy operations and class mechanism
Copying problems
Implementing copy operations user-defined conversions
Unintended conversions and `explicit`
Conversions, temporaries, and efficiency
Temporaries and copy construction
Return value optimizations
Conversions, temporaries, and references
Temporary lifetime and correctness

Assignment and Initialization

Assignment has nothing to do with initialization! They are two separate, very different, operations.

There are several initialization contexts:

- declarations (and implicitly for compiler-generated temporaries)
- formal arguments
- returns
- member initialization lists
- initialization of the parameter of an exception handler

There is one assignment context:

- assignment

The difference between assignment and initialization is most obvious in the cases of

- references
- constants
- class objects with constructors

Assignment and Initialization

Assignment is not initialization.

However, it is generally important that these operations produce identical results.

```
String original( "Hello" );
String copy1( original ); // initialization
String copy2;
copy2 = original; // assignment
```

- Some libraries may substitute initialization for assignment.
- Exception safe assignment is often implemented with copy construction.

Generally, it's best if these operations don't have side effects.

- Copy constructor calls are often "optimized" away.

Copy Operations

Copying is defined by the operations X(const X &) and operator =(const X &), the copy constructor and the copy assignment operator.

If copy operations are not explicitly defined for a class, the compiler supplies them implicitly, if needed.

The default semantics are memberwise copy. However,

- internal class mechanisms, such as virtual function table pointers and virtual base class pointers are not affected by the object from which the copy is made
- in assignment, these mechanisms are not changed
- in initialization, they are set by the constructor without regard for the corresponding values in the initializer

Note that memberwise copy will invoke the appropriate copy constructor for initialization and assignment operator for assignment if they are defined for a given member.

Note that memberwise copy defaults to bitwise copy in the case of C-style structs (PODs).

Assignment Operations and Class Mechanism

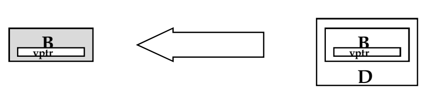
Assignment does not affect internal object structures.

- virtual function table pointers
- virtual base class pointers/offsets

These are set by the object's constructor.

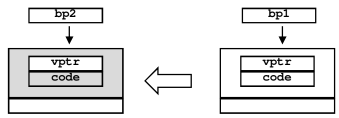
Once an object is created, its type does not, or should not, change.

```
class B {
    virtual void f();
    // ...
};
class D : public B {
    void f();
    // ...
};

B b;
D d;
b = d; // slice!
```



Aside: Type Codes and Copying

```
class Base {
  public:
    enum Tcode { DER1, DER2, DER3 };
      Base( Tcode c ) : code( c ) { }
      Tcode tcode( ) const { return code; }
      virtual void f( ) = 0;
  private:
      Tcode code;
};

class Der1 : public Base {
  public:
      Der1( ) : Base( DER1 ) { }
      void f( );
};
// ...
Base *bp1 = new Der1;
Base *bp2 = new Der2;
// ...
*bp2 = *bp1; // disaster!
```

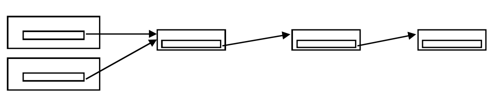


Aside: Deep and Shallow Copy

The terms "deep" and "shallow" copy are often employed in discussions of copy semantics.

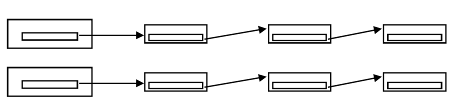
A shallow copy copies only the class object itself.

- not decapsulated storage (or other resources) to which it refers
- results in aliasing of the decapsulated storage



A deep copy recursively copies any decapsulated storage.

- no aliasing occurs



Either approach may be appropriate in context; but it's essential to document which approach is taken.

Compiler-Supplied Default Copy Semantics

If a class does not explicitly provide copy operations, the compiler will provide default versions implicitly, if needed.

The default copy constructor performs memberwise initialization.

- If the subobject or member in question has a copy constructor (implicit or explicit) that is invoked to perform the initialization.
- Otherwise bitwise initialization is performed.

The default assignment operator performs memberwise assignment.

- If the subobject or member in question has an assignment operator (implicit or otherwise) that is invoked to perform the assignment.
- Otherwise bitwise assignment is performed.

Default copy operations exhibit shallow copy semantics.

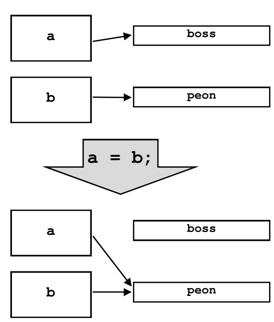
Not Thinking About Copy Operations

Copy operations must always be considered.

- accept the compiler default
- supply your own
- disallow them

The compiler will write copy operations for free.

You get what you pay for.




```
class Employee {
  public:
      Employee( const Name &name,
          const Address &address );
      ~Employee( );
      void adoptRole( Role *newRole );
      const Role *getRole( ) const;
      // ...
  private:
      Name _name;
      Address _address;
      Role *_role;
};
```

Providing Only Half the Copy Semantics

Assignment and initialization are different operations. If copy semantics are important in one, chances are they're important in the other.

Forgetting to consider both assignment and initialization will result in bugs.

```
class SloppyCopy {
        T *ptr;
  public:
        SloppyCopy &operator =( const SloppyCopy & );
        // Note: compiler default
        //     SloppyCopy(const SloppyCopy &)...
};

void f( SloppyCopy ); // pass by value...

SloppyCopy sc;
f( sc );    // alias what ptr points to!
```

Advice: If you supply one copy operation, consider (supply, default, remove) the other.

Copying Problems

```
template <typename T>
class Stack { // Implementation #1
  public:
    Stack( );
    ~Stack( );
    bool empty( ) const;
    void push( const T & );
    T pop( );
  private:
    enum { MAX = 1024 };
    int top;
    T s[ MAX ];
};
```

```
template <typename T>
class Stack { // Implementation #2
  public:
    Stack( );
    ~Stack( );
    bool empty( ) const;
    void push( const T & );
    T pop( );
  private:
    int top, max;
    T *s;
};
```

```
Stack<int> a, b;
a = b;
a.push( 12 );
b.push( 24 );
int i = a.pop( );
```

Avoiding Copying Problems

```
template <typename T>
class Stack { // Implementation #2
  public:
    Stack( );
    ~Stack( );
    bool empty( ) const;
    void push( const T & );
    T pop( );
  private:
    Stack( const Stack & );
    Stack &operator =( const Stack & );
    int top, max;
    T *s;
};
```


Compiler-Supplied Copy Operations

Sometimes the compiler can write a more efficient copy operation than we can.

```
class Big {
  public:
    // Big( const Big & );
    // Big &operator =( const Big & );
    //…
  private:
    int a[1000];
    char name[32];
    int index;
};
```

The compiler knows the class layout, and can use efficient bitwise copy, or machine-specific instructions wherever appropriate.

Remember that the generated operations will be both `public` and `inline`.

Bitwise Copy

Avoid bitwise copy!

```
Big &Big::operator =( const Big &that )
  { memmove( this, &that, sizeof(Big) ); }
```

The compiler will detect changes in layout, but you won't.

```
class Big {
    virtual void f( );
    //…
  private:
    int a[1000];
    string name;
    int index;
};
```

Layout changes can occur during maintenance.

Layout changes can occur without source code changes when porting, getting a new version of a compiler, or by changing compilation options.

Coding Copy Constructors

Any constructor that can accept an object of the same type as the class is a copy constructor.

```
X( X &, int = 0, X * = 0 ); // strange...
```

However, the usual and proper declaration for a copy constructor is as follows.

```
X( const X & ); // typical
```

The argument type is a const reference to the class, because (presumably) the initializer is not going to be changed.

Remember that you are responsible for implementing the "expected" semantics.

Coding Copy Assignment Operators

The assignment operator must be a member. The argument type is generally a reference to const, since assignment shouldn't change the source of the assignment.

```
X &operator =( const X & );
```

The return value is usually a non-const reference to the object being assigned. This allows assignments to be "chained" in the same way the assignments for the predefined types are. The return statement is invariably

```
return *this;
```

Often, check for assignment to self. Failure to do so may result in either error or inefficiency.

```
X &X::operator =( const X &x ) {
    if( this != &x ) {
        // copy
    }
    return *this;
}
```


Initialize Memory, But Assign Objects!

Note that the target of the assignment must be "cleaned up" before the source can be copied into it.

```
a = b; // "destroy" a, then "initialize" it
```

This means that you shouldn't try to assign to uninitialized storage!

```
char buf[ sizeof(string)];
string *sp = static_cast<string *>(buf);
*sp = "Goodbye, cruel world!"; // oops!
```

In this case, you probably want to *initialize* the memory.

```
char buf[sizeof(string)];
string *sp = new (buf) string( "Hello, World!" );
```

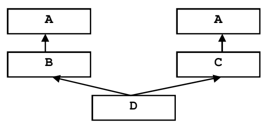
Of course, it's OK to assign to uninitialized objects of predefined types.

Assignment in a Hierarchy

The proper coding of assignment for classes in a hierarchy is *still* an item of debate.

One typical "assignment operator discipline" is as follows:

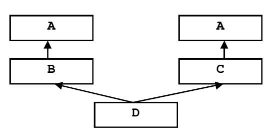
- Base class assignment is protected (and base classes are abstract!)
- Derived classes call the assignment operators for their immediate base classes.
- Then the derived class assignment operator assigns the derived class members.
- Derived class assignment operators do not touch base class members. Base class assignment operators to not touch derived class members, even indirectly by calling a virtual function.



This results in a nicely layered approach.

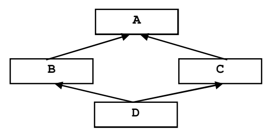
Assigning a Hierarchy

```
class A { protected: A &operator =( const A & ); };
class B : public A { protected: B &operator =( const B & ); };
class C : public A { protected: C &operator =( const C & ); };
class D : public B, public C { public: D &operator =( const D & ); };

B &B::operator =( const B &b ) {
    if( &b != this ) {
        A::operator =( b );
        // assign local members...
    }
    return *this;
}
D &D::operator =( const D &d ) {
    if( &d != this ) {
        B::operator =( d );
        C::operator =( d );
        // assign local members...
    }
    return *this;
}
```



Assignment and Virtual Bases

Our assignment operator discipline will not work if virtual bases are involved, however.



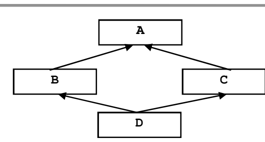
If we apply our discipline to the above hierarchy, the virtual base A will be assigned twice.

As usual, the presence of a virtual base class is intrusive.

The D::operator = must explicitly assign the virtual base as well as its immediate base classes. This also implies that classes B and C must supply "local" assignment functions.

This is similar to the behavior of constructors under virtual inheritance, but we must code the behavior explicitly.

Assigning Virtual Bases




```
class A {
  // ...
};
class B : public virtual A {
  public:
    B &operator =( const B & );
  protected:
    void doLocals( const B & );
};
class C : public virtual A {
  // ...
};
class D : public B, public C {
  public:
    D &operator =( const D & );
};
```

```
B &B::operator =( const B &b ) {
    if( &b != this ) {
        A::operator =( b );
        doLocals( b );
    }
    return *this;
}
D &D::operator =( const D &d ) {
    if( &d != this ) {
        A::operator =( d );
        B::doLocals( d );
        C::doLocals( d );
        // assign local members
    }
    return *this;
}
```


Dealing with Virtual Bases

Note that compiler-generated assignment may (or may not!) assign multiple times to a shared virtual base subobject!

Often, it is a good idea that virtual base classes have no non-static data members.

- The will consist entirely of class mechanism.
- Class mechanism is not affected by assignment, only construction.
- Assignment to such a virtual base is a no-op.

Compiler-generated assignment will work correctly.

The non-virtual hierarchy assignment discipline will work properly!

Recommendation: If you can, only use interface classes as virtual bases.

Another recommendation: Avoid virtual base classes if you don't really need them.

Template Constructors and Copying

A constructor that is a member function template is never a copy constructor.

```
template <class Currency>
class Money {
  public:
    Money( double amt = 0.0 );
    template <class OtherCurrency>
    Money( const Money<OtherCurrency> & );
    // Money( const Money & );
    //…
};

Money<Yen> acct1( 1000000.00 );
Money<DM> acct2( 123.45 );
Money<Yen> acct3( acct2 ); // template ctor
Money<Yen> acct4( acct1 ); // compiler-generated copy!
```

In the absence of an explicitly-declared copy constructor, the compiler will still generate one.

Virtual Assignment

Assignment may be virtual.

```
template <typename T>
class Container {
  public:
    virtual Container &operator =( const T & ) = 0;
    //…
};
template <typename T>
class List : public Container<T> {
    List &operator =( const T & );
    //…
};
template <typename T>
class Array : public Container<T> {
    Array &operator =( const T & );
    //…
};

Container<int> &c( getCurrentContainer( ) );
c = 12; // but what does this mean?
```


Virtual Copy Assignment

Copy assignment may be virtual, but this is rarely a good idea.

The derived class copy assignment operator does not override the base class copy assignment.

```
template <class T>
class Container {
  public:
    virtual Container &operator =( const Container & ) = 0;
    //…
};

template <class T>
class List : public Container<T> {
    List &operator =( const List & ); // doesn't override!
    List &operator =( const Container & ); // overrides…
    //…
};
```


Virtual Copy Meaning

There's also the question of exactly what virtual copy assignment means.

```
Container &c1 = getMeAList( );
Container &c2 = getMeAnArray( );
c1 = c2; // assign an array to a list?!?
```

Typically, it's better to just say what you mean.

```
Container &c1 = getMeAList( );
Container &c2 = getMeAnArray( );
c1.copyContentOf( c2 ); // oh...
```

The standard library achieves flexibility without virtuality.

```
list<Blob> blist;
…
vector<Blob> bvec( blist.begin(), blist.end() );
…
bvec.assign( blist.begin(), blist.end() );
```


Be Clear

The same observation applies to our earlier use of assignment.

```
Container<int> &c( getCurrentContainer( ) );
c = 12; // what does this mean?
```

It's time to stop being clever and be clear.

```
Container<int> &c( getCurrentContainer( ) );
c.setAll( 12 );
c.resize( 12 );
c.setFirst( 12 );
```

Copying Summary

Assignment and initialization are totally different operations.

Copy operations are special, and are defined by the copy constructor and copy assignment operator.

The compiler will write these operations for you; make sure that's what you want.

Copy construction and copy assignment should produce identical results.

Never write copy operations that tamper with class mechanism; avoid memcpy-like operations unless you really, really know what you're doing.

Use a standard mechanism to code copy assignment in a hierarchy; abstract bases have protected copy assignment.

Understand template copy constructors.

Avoid virtual copy assignment.

Overuse of Conversion Operators

Overuse of conversion operators increases code complexity.

```
class Cell {
  public:
    operator int( ) const;
    operator double( ) const;
    operator const char *( ) const;
    //…
};
```

Because the conversion operators are applied implicitly, there will often be ambiguity.

```
void process( long );
…
Cell c;
...
process( c ); // error!
```

Even if there is no ambiguity, it's heard to tell exactly what conversion is taking place, since the call is implicit.

Minimize Conversion Operators

It's generally better and always clearer to use explicit conversion functions.

```
class Cell {
  public:
    int toInt( ) const;
    double toDouble( ) const;
    const char *toCharArray( ) const;
    //…
};
```

Most classes should not have conversion operators.

A single conversion operator is sometimes warranted.

Multiple conversion operators are almost always bad design.

```
void process( long );
…
Cell c;
...
process( c.toInt() ); // better...
```


Value-Added Conversions

Conversion operators are for type conversion, not for taking the place of other member functions.

Use of a conversion operator for anything but type conversion is confusing and will lead to ambiguity.

```
class complex {
    // ...
    operator double( ) const;
};
complex velocity = x + y;
double speed = velocity;

class Container {
    //...
    virtual operator Iterator *( ) const = 0;
};
Container &c = getNewContainer( );
Iterator *i = c;
```


Value-Added Conversions

Say what you mean, and stop trying to show off.

```
class complex {
    // ...
    double magnitude( ) const;
};

complex velocity = x + y;
double speed = velocity.magnitude( );

class Container {
    //...
    virtual Iterator *genIterator( ) const = 0;
};

Container &c = getNewContainer( );
Iterator *i = c.genIterator( );
```


I'm OK, You're OK

This even applies to a simple "OK?" question.

```
class X {
  public:
    operator bool( ) const; // are we OK?
    //…
};
//…
extern X &anX;
if( anX ) {
    // anX is OK...
```

What does it mean to be "OK"? Later, this may change or be augmented.

```
class X {
  public:
    bool isValid( ) const;
    bool isUsable( ) const;
    //…
};
```

Unintended Conversions

We might also inspire some intentional or unintentional "cleverness."

```
extern vector<X> xs;
vector<X>::const_iterator i = xs.begin( );
iterator_traits<vector<X>::const_iterator>::difference_type count = 0;
while( i != xs.end( ) )
    count += *i++; // ???
```

The iostream library tries to avoid this problem by implementing the "OK" function as an `operator void *`.

```
if( cout ) // same as cout.operator void *( )
    //...
```

This helps to prevent some problems.

```
cout >> 12; // won't compile, fortunately
```

But it can still be abused.

```
cout << cout << cin << cerr; // legal!
```

Why don't we just say what we mean?

```
if( !cout.fail( ) ) //...
```


An Exception: Null Pointers

However, it is idiomatic for a smart pointer to have an implicit null/non-null conversion operator.

As we've seen, a numeric conversion is often not a good idea.

One possibility is a conversion to `void *`.

Another candidate is a conversion to a pointer to data member!

In many parts of the standard library and TR1 extensions this conversion is implementation-defined.

Unintended Constructor Conversions

Most single argument constructors should not specify conversions.

```
template <typename T>
class Stack {
  public:
    Stack();
    Stack( int maxSize );  // problematic...
    bool operator ==( const Stack & );
    bool full( ) const;
    bool empty( ) const;
    void push( const T & );
    void pop( );
    T &top( );
    //...
};
```


Unintended Constructor Conversions

Consider the following code.

```
Stack<int> s;
s.push( 12 );
if( s == 12 ) // oops!
  //...
```

The code is, unfortunately, legal.

```
Stack temp; bool result;
if( (temp.Stack<int>(12), result = s.operator ==(temp),
   temp.~Stack<int>( ), result) ) //...
```

Constructors and explicit

The constructor should indicate that it may not be invoked implicitly as a conversion.

```
template <class T>
class Stack {
  public:
    explicit Stack( int maxSize );
    //...
};
Stack<int> s;
s.push( 12 );
if( s == 12 ) // error! no conversion
    //...
if( s == static_cast< Stack<int> >( 12 ) )
    // you can still hang yourself...
    // but you have to ask for the rope.
```


explicit and Initialization

explicit affects the set of legal initialization syntaxes of Stack.

```
Stack<float> a( 1024 ); // OK...
Stack<float> b = 1024; // error! conversion
Stack<float> c = Stack<float>( 1024 ); // legal, but a bad idea
```

We’ll see shortly that this is a difference between *direct* and *copy* initialization.

Conversions, Temporaries, and Efficiency

Consider a String class that defines equality operators.

```
class String {
  public:
    String( const char * = "" );
    ~String( );
    friend bool operator ==( const String &, const String & );
    friend bool operator !=( const String &, const String & );
    …
  private:
    char *s_;
};

inline bool
operator ==( const String &a, const String &b )
      { return strcmp( a.s_, b.s_ ) == 0; }

inline bool
operator !=(const String &a, const String &b )
      { return !(a == b); }
```


Conversions and Efficiency

```
String s = "Hello, World!";
String t = "Yo!";

if( s == t ) {
      // ...
}
else if( s == "Howdy!" ) {
      // ...
}
```


Exact Matches

```
class String {
  public:
      String( const char * = "" );
      ~String( );
      friend bool operator ==( const String &, const String & );
      friend bool operator !=( const String &, const String & );
      friend bool operator ==( const String &, const char * );
      friend bool operator !=( const String &, const char * );
      friend bool operator ==( const char *, const String & );
      friend bool operator !=( const char *, const String & );
      // …
};

inline bool
operator ==( const String &a, const String &b )
      { return strcmp( a.s_, b.s_ ) == 0; }

inline bool
operator ==( const char *a, const String &b )
      { return strcmp( a, b.s_ ) == 0; }
```


Naive Implementations

```
class String {
  public:
      friend String operator +( const String &, const String & );
      //…
  private:
      char *s_;
};

String operator +( const String &a, const String &b ) {
      size_t len = strlen(a.s) + strlen(b.s) + 1;
      char *buf = new char[ len ];
      strcat( strcpy( buf, a.s_ ), b.s_ );
      String retval( buf );
      delete [ ] buf;
      return retval;
}
//…
void f() {
      String s1, s2 = "World!";
      s1 = "Hello, " + s2;
}
```

Computational Constructors

Sometimes it makes sense to employ a constructor as a helper function.

– A constructor can assume that it is working with uninitialized storage!

These are known as “computational” constructors.

```
String::String( const char *a, const char *b ) {
    size_t len = strlen(a) + strlen(b) + 1;
    s_ = strcat( strcpy(new char[ len ], a ), b );
}
```

Computational constructors are typically not part of a class’s public interface.

Using a Computational Constructor

```
class String {
  public:
    //...
    friend String operator +( const String &, const String & );
    friend String operator +( const char *, const String & );
    friend String operator +( const String &, const char * );
  private:
    char *s_;
    String( const char *, const char * ); // computational
};

String operator +( const String &a, const String &b )
    { return String( a.s_, b.s_ ); }

String operator +( const char *a, const String &b )
    { return String( a, b.s_ ); }

void f( ) {
    String s1, s2 = "World!";
    s1 = "Hello, " + s2;
}
```

Direct vs. Copy Initialization

Consider the following simple class.

```
class X {
 public:
    X( int );
    ~X( );
};
```

We have a variety of different ways of accomplishing the "same" initialization.

```
X a( 42 ); // direct init
X b = 42;
X c = X( 42 );
```

The first initialization specifies that the constructor that takes an int formal argument be called to do the initialization.

This is a direct initialization.

Copy Initialization

The remaining two initializations specify the following:

- init a temp of type X, using the constructor that takes a single int argument
- use the copy constructor to initialize the X being declared
- call the X destructor to destroy the temporary

These are copy initializations.

```
X a( 42 ); // direct init
X b = 42; // copy init
X c = X(42); // copy init
```

Temporaries and Copy Construction

However, the compiler is allowed to optimize away the temporary and its associated construction and destruction.

```
X a( 42 );
X b = 42; // same result as X b(42);
X c = X(42); // same result as X c(42);
X d = (X)42; // same result as X d(42);
```

Most compilers will perform the optimization.

However, a compiler does not have to, and the behavior and efficiency of your code may vary from platform to platform.

It is best to say precisely what you mean.

```
X a( 42 ); // preferred
```

For predefined types, use whichever form you think is clearest.

```
int i = 12;
int j( 12 );
```


Access and explicit

The compiler is still required to check the access of the calls to the functions that are optimized away.

```
class Y {
  public:
     Y( int );
     ~Y( );
  private:
     Y( const Y & );
};
Y e = Y( 1066 ); // error!
Y f( 1066 ); // OK
```

A single argument constructor that is explicit further restricts the initialization syntax.

```
class Z {
  public:
     explicit Z( int );
     Z( const Z & );
     //…
};
```

```
Z g = 1066; // error! conversion
Z h( 1066 ); // OK, direct init, no conversion
Z i = Z( 1066 ); // OK, explicit conversion and copy init
```

Temporaries and Return By Value

It is often necessary to return the result of a function by value.

- This logically necessitates the creation of some temporary value in the body of the calling function, and a copy construction of the return result.

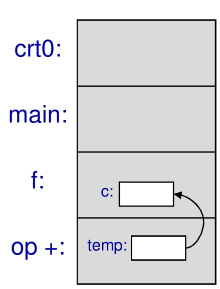
The canonical example is the implementation of an overloaded binary operator.

```
String operator +( const String &a, const String &b ) {
    String temp( a );
    temp += b;
    return temp;  // copy ctor
}
```

The return value initialization is often implemented with a simple transformation.

```
void operator +( String &ret, const String &a, const String &b ) {
    String temp( a );
    temp += b;
    ret.String( temp ); // copy ctor (compiler can do this, we can't!)
    return;
}
```


Return By Value




```
void f( const String &a, const String & b ) {
    …
    String c( a + b );
   …
}
String operator +( const String &a, const String &b ) {
    String temp( a );
    temp += b;
    return temp;  // copy ctor
}
```


The Return Value Optimization

A programmer can achieve performance gains by use of an anonymous temporary return.

```
inline String operator +( const String &a, const String &b )
      { return String( a, b ); }
```

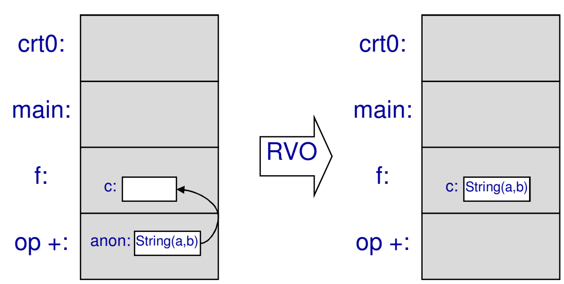
The compiler is allowed to eliminate creation of the anonymous temporary.

The performance gain can be significant.

```
String a = "Hello";
String b = "Bye";
String c = a + b; // c.String(a.s_,b.s_)
```


RVO

```
void f( const String &a, const String & b ) {
   …
   String c( a + b );
  …
}
String operator +( const String &a, const String &b )
      { return String( a.s_, b.s_ ); }
```



Prefer Initialization to Assignment

When you initialize, there's nothing to clean up!

When you assign, typically you have to first "destroy" an object before you "reinitialize" it.

Therefore the RVO cannot copy directly into the target of an assignment.

```
c = a + b;
```

```
String tmp;
tmp.String( a.s_, b.s_ );
c = tmp;
tmp.~String();
```

Initialization will generally be more efficient than assignment.

The Named Return Value Optimization

The named return value optimization (NRV) is a commonly-supported optimization.

The compiler is free to apply the optimization if all the return expressions are the same named local object, and the return type has a copy constructor defined.

For example, the following function

```
X f( int a ) {
    X local( a );
    // do stuff with local...
    return local;
}
```

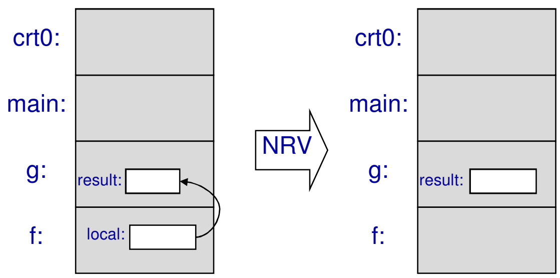
may be modified by the compiler to

```
void f( X &result, int a ) {
    result.X::X(a);
    // do stuff with result...
    return;
}
```

The performance gain for this optimization can be significant, but the compiler is not obliged to perform it.

NRV

```
void g( ) {
    …
    X result( f( 12 ) );
   …
}
X f( int a ) {
        X local( a );
        // do stuff with local...
        return local;
}
```



Lifetime of Temporaries and Correctness

The lifetime of a temporary is from its point of creation to the end of the largest enclosing expression.

Do not depend on the continued existence of a temporary after the expression in which it was created.

```
String s1, s2;
cout << s1+s2;
const char *p = s1+s2; // assumues operator char *();
cout << p;
```

OK, temporary destroyed after function call

Not OK, temporary destroyed before function call

However, note that a temporary used to initialize a reference will live as long as the reference.

```
const String &ab = a + b; // safe...
```

Temporaries and References

A reference to const may actually refer to an initialized temporary if the initializer is not an lvalue or is not of the correct type.

```
short s = 12;
const int &ri1 = s; // temporary
const int &ri2 = 12; // temporary
void f( const int &ri );
f( s ); // temporary
int &ri3 = s; // error!  not reference to const
```

This can give rise to subtle errors if the lifetime of the temporary is shorter that that of the reference to it.

Temporaries and References

One subtle implication of the lifetime of temporaries is that it's not a good idea for a function to return a reference to const that has been passed to it as an argument.

```
class X { public: X( int ); };
…
const X &f( const X &a ) { return a; } // dangerous!
…
const X &r = f( 123 ); // subtle error
cout << r; // disaster!
```


Summary

Don’t overuse conversion operators.

Be aware of the cost of implicit user-defined conversions.

Sometimes it is profitable to special case by providing exact matches for certain combinations of operand types.

If no other solution presents itself, consider restricting or modifying the interface in order to prevent the user from writing very inefficient code.

C++ compilers commonly provide optimizations that eliminate temporaries and their attendant constructor and destructor calls.

Consider facilitating commonly-available compiler optimizations.

Understand how the lifetime of temporary objects can affect the meaning of your code.

semantics

Exercises

Exercises

Exercise 1, directory Initialization

- Consider the very simple class X in file inits.cpp.
- Identify the direct initializations, the copy initializations, and the assignments of X objects in the file.
- Change the declaration of X's single-argument integer constructor to be explicit. Which initializations and assignments should now be illegal? Did your compiler get it completely right?
- Make X's copy operations private. Now what code should break? How'd your compiler do this time?

Exercises

Exercise 2, directory Rational

- Design a Rational class that represents rational numbers. Implement the class with a pair of integers for numerator and denominator.
- Don't do a full implementation, provide just initialization, copying, multiplication, comparison, and iostream output. Don't bother factoring or normalizing numerator and denominator.
- Go for efficiency in the implementation!

Exercise 3, directory RationalTemplate

- Turn your Rational class into a Rational class template, where the type of the numerator and denominator may be any integral type, including a user-defined integral type.
- What happened to your conversions? Fix them.

Exercises

Exercise 4, directory SmallString

- Refer to the file ssostring.h.
- The class template SSOString is used to generate strings that employ the "small string optimization." For example, the declaration "SSOString<15> str("hello!");" declares str to be a string that will store character strings of up to length 15 (not counting a terminating '\0') within itself. Longer strings are stored in a heap-allocated buffer. An SSOString<3> will do the same for strings of length up to 3, etc. Note that SSOString<n> and SSOString<m> are completely different class types if n != m.
- Play around with the SSOString template, creating and using SSOStrings with different sizes of small string optimization.
- Can you mix and match different-length SSOStrings in the same expression? What conversions are being implicitly applied?
- Are these conversions efficient enough?
- Have a look at the comments in the code, and try to answer the questions they pose.
- Warning: This exercise is more difficult than it looks!

Exercises

Exercise 5:

- Consider the following code snippet:

```
String String::subString(unsigned startPos,
                         unsigned len, char padCharacter) const
{
    if( startPos > 0 )
        --startPos;
    if ( startPos >= length() || len == 0 )
        return "";
    String res=substr(startPos, len);
    if( startPos+len > length() )
        res.append( (startPos+len)-length(), padCharacter );
    return res;
}
```

- Why doesn't it encourage the compiler to apply RVO or NRV?
- Show how it could be rewritten to encourage the RVO or NRV.

Answers

Exercise 5:

– Here's one possibility that encourages application of the NRV:

```
String String::subString(size_t startPos, size_t len, char
                          padCharacter) const {
    if( startPos > 0 ) // perform magic calculation
        --startPos;
    const size_t thisLen = length();
    if ( startPos >= thisLen || len == 0 )
        startPos = len = 0;

    String res=substr( startPos, len );

    if( startPos+len > thisLen )
        res.append( (startPos+len)-thisLen, padCharacter );

    return res;
}
```


Answers

Exercise 5 (Cont.):

– Here's a second approach that encourages the RVO with an anonymous temporary return:

```
String String::subString(size_t startPos, size_t len,
                         char padCharacter ) const
{
    if( startPos > 0 ) // perform magic calculation
        --startPos;

    if ( startPos >= length() || len == 0 )
        startPos = len = 0;

    return substr( startPos, len ).resize( len, padCharacter );
}
```

