Temporaries, Return Value Optimization, Rvalue References

In these slides, I discuss three related topics:

1. The return value optimization (RVO).
2. Temporaries in expressions.
3. Rvalue references.

Problems with Heap Objects

Remember the implementation of **string**:

```
size_t len;
char *p;
```

We are interested in the following:

1. How do you return a string from a function?

```
string s = f(  );    // Function call.

string f( ) { string res; ... ; return res; }
   // return = Implicit copy constructor.
   // Followed by destruction of res.
```

2. How do you exchange two strings?

```
{ string help = s1; s1 = s2; s2 = help; }
```

Is this efficient?

What about:

1.
```
string s = f( );

string f( ) { string res;
              s.p = res.p; res.p = nullptr;
              s.len = res.len; res.len = 0; }
                 // Destructor of res does nothing.
```

2. Exchange:

```
char* pp = s1.p; s1.p = s2.p; s2.p = pp;
size_t ll = s1.len; s1.len = s2.len; s2.len = ll;
```

Both implementations are constant time. But: User has to know implementation of **string**, which breaks fundamental principle of C^{++}.

Return Value Optimization (RVO)

The following code pattern is very frequent:

```
X f( Y1 y1, ... Yn Yn )
{
   X x = .... ;

   computations involving x.

   return x;
}
```

RVO (2)

Compilation results in:

```
f( Y1 y1, ..., Yn yn )
{
   Construct variable x of type X.

   (computations involving x.)

   Use copy constructor (of X) to copy variable x
   to place of function result.

   Destroy variable x.
}
```

RVO (3)

The RVO states that the compiler has the right to use the position of the function result as variable `x` from the beginning.

```
f( Y1 y1, ..., Yn yn )
{
   Construct an X on the place of the function result.

   Do the computations with x on the place
   of the function result.

   return; // No copy is necessary.
      // No destruction is necessary.
}
```

RVO (4)

If the first local variable of a function has the return type of the function, then don't allocate this variable together with the other local variables of the function, but make it equal to the place of the return value, that was allocated by the calling environment.

It saves one call of the copy constructor.

This is part of the standard of C^{++}. To be precise:

The compiler has the right (not the obligation!) to compile in the second way (remove a copy constructor application), even if the copy constructor has side effects.

RVO has been present in C^{++} very long.

Move Semantics

Now consider a function of the following form:

```
X f( Y1 y1, ... Yn Yn )
{
   X x1 = ... ;
   X x2 = ... ;
   X x3 = ... ;

   computations involving x1,x2,x3.

   if( .. ) return x1;
   else if( ... ) return x2;
   return x3;
}
```

RVO won't help much.

Moving

The problems on the previous slides (Inefficiency of **return** when a datatype holds resources on the heap, inefficiency of exhange) can be solved by **move** semantics:

```
{  string help = std::move(s1);
   s1 = std::move(s2);
   s2 = std::move( help ); }


   return res;
      // Will return std::move(res) automatically.
      // No need to write it.
```

Moving (2)

Introduce a special reference && to objects that are about to be overwritten or destroyed. Such reference is called rvalue reference.

```
string( string&& s ) noexcept
  : len{ s.len },
    p{ s. p }
  { s. len = 0; s. p = nullptr; }

const string& operator = ( string&& s ) noexcept
{
   std::swap( p, s. p );
   std::swap( len, s. len );
   return *this;
}
```

Moving (3)

What are the conditions:

1. A moving operator may change the value of its `&&`-argument in random way.
2. But, it must preserve global invariants and should preserve class invariants.
3. The operator must not destroy the object, because it will be destroyed or overwritten later.

Moving (4)

Exchange becomes:

```
{  help. p = s1.p; help.len = s1.len;
   s1.p = nullptr; s1.len = 0;

   std::swap( s1.p, s2.p );
   std::swap( s1.len, s2.len );

   std::swap( s2.p, help.p );
   std::swap( s2.len, help.len );
}
```

(optimizer will clean up)

Automatic Move

There are two cases, where the compiler considers rvalue parameters automatically:

1. In a statement `return x` with `x` a variable of type `X`, the compiler will use the moving copy constructor for type `X` if there is one. (and RVO is not used.)
2. In an expression `f( ..., g( ), ... )`, where `g( )` returns a value `X`, and `f` has a definition with corresponding argument of type `X&&`, this definition of `f` will be considered.

   Such argument is called a temporary. I will discuss them separately.

In other cases, one must use `std::move(x)`. Note that `move` does not move anything. It only gives the compiler a permission to consider arguments of type `X&&`, when `x` has type `X&`.

Temporaries

When expressions are being evaluated, there are all kinds of implicit variables, but only one type is called temporary, namely when a function returns a value (of some type `X`), and the next function requires a reference as argument, either of type `const X&` or `X&&`.

```
X xx( );
void g1( const X& );
void g2( X&& );
void g3( X& )

g1( xx( ));  // Contains temporary.
g2( xx( ));  // Contains temporary.
g3( xx( ));  // Not allowed, because
             // X& is intended for meaningful output.
```

Why temporaries are special

Other combinations require no special treatment:

```
X& xxref( );
void g4( X x );

g4( xx( ));   // xx writes its return value into
              // already allocated local variable
              // x of g4.

g4( xxref( )); // Compiler inserts copy constructor
               // for X, if there is one.
```

Temporaries are not artificial. They occur all the time, e.g.
`std::cout << s1 + s2 << "\n";` for our string class.

Life Time of Temporaries

How long should a temporary exist?

1. Until next function is completed? This seems natural, because it corresponds to the life time of local variables, but it is too short. (See next slide.)
2. Until expression is complete?
3. Until block is complete. Slightly better than previous, but temporaries that exist too long require too much space. Programmers will create artificial blocks.

C^{++} uses option 2.

References can be passed through a Function Call

Consider

```
const bigint& max( const bigint& b1, const bigint& b2 )
{
   if( b1 > b2 )
      return b1;
   else
      return b2;
}
```

Consider expression

`m = max( max( i1 + i2, j1 + j2 ), max( k1 + k2, l1 + l2 ))`

Cleaning up the temporaries on the inner level of `max` is not possible.

It follows that **(1)** is too early.

Option (3) is too long, because programmers don't like it when temporaries exist too long.

```
m1 = max(i1,i2);
m2 = max(j1,j2);
```

Instead, they will write

```
{ m1 = max(i1,i2); }
{ m2 = max(j1,j2); }
```

which is bad code.

One could imagine a recursive procedure where the effect is much worse.

Therefore, C^{++} uses (2).

Temporary Values in Expressions

The examples on the previous slides are not artificial. Assume a class **bigint** representing big integers, so that we can evaluate 70! or 2^{100} exactly.

Assume that we have declarations

```
bigint operator + ( const bigint& n1, const bigint& n2 );
const bigint& bigint::operator = ( const bigint& n2 );
bigint operator * ( const bigint& n1, const bigint& n2 );
bigint( const bigint& n );
bigint( int i );
~bigint( );
```

Consider expressions

```
bigint i = 4;
bigint j = i * i * i * ... * i;
j = j + j + j + ... + j;
```

Rvalue References

A variable of type `X&&` is called rvalue reference, because it is obviously a reference, but not a reference that you can assign to.

It is also important to observe that inside the function (that has `X&&` as argument), variable `x` is just a regular reference `X&`.

Usual rules of slide 13 apply.

Rvalue Reference

An Rvalue reference is a reference to an object that will be overwritten or destroyed by its owner.

The function that has the Rvalue reference is the last user of the current value of the object before the owner of the object overwrites or destroys it.

The notation for Rvalue reference is `X&&`.

1. It differs from `const X&`, because we can change it.
2. It differs from `X&`, because `X&` is intended for meaningful output.

In which state can an Rvalue function leave the object?

We assume **(1)** of Slide 28.

After a call of `f( X&& x )`, variable `x` will be either destroyed or overwritten.

This means that we can spoil all invariants of `X`, as long as preconditions of `X::operator =( )` and `~X( )` are preserved.

Pointer fields that assume unique ownership must point to something that can deallocated or overwritten, and preserve unique ownership.

Pointer fields that assume sharing with reference counting must point to something that has a valid reference counter.

operator =( )

Very often, moving assignment can be implemented by exchange:

```
void operator = ( string&& s )
{
   std::swap( p, s.p );
   std::swap( len, s. len );
}
```

Rvalue Copy Constructor

```
string( string&& s )
  : len{ s. len },
    p{ s. p }
{
   s. p = nullptr;
   s. len = 0;
}
```

Possible because **nullptr** is almost the same as pointer to zero length segment.

Following also works:

```
string( string&& s )
   : string( )         // Use default constructor.
{
   *this = std::move(s);
      // It's an assignment.
}
```

A Different View of Assignment

Before we used: **assignment** = **destructor** + **copy constructor**.

```
const string& operator = ( const string& s )
{
   if( len != s.len )
   {
      delete[] p;
      p = new char[ s.len ];
      len = s.len;
   }
   for( size_t i = 0; i != len; ++ i )
      p[i] = s.p[i];
   return *this;
}
```

There is repeated code between assignment and copy constructor.

Different View of Assignment (2)

Alternatively, one can use: **assignment** =
copy constr. + **rvalue assignment** + **destruction**.

```
void operator = ( const string& s )
{
   return ( *this = string(s) );
      // CC makes a copy which is a temporary, which
      // will be passed operator = ( string&& ).
}
```

Probably, rvalue assignment should be considered as more elementary than copying assignment.

What is moving?

It is difficult to give a general definition of what 'moving' is. I think there are two forms:

1. We have a type X that holds resources, and the user does not know about it (**string**, **vector**). In such cases, some operations (passing temporary argument to function, returning a value, exchanging variables) can be implemented more efficiently when ownership is transfered. The user of X should not see this.

2. Some types hold resources are part of their specification. Examples are filehandles, mutexes, smart pointers.

   In that cases, the user sometimes may decide to transfer ownership. In this, the user expects it, and relies on it.

When to define moving functions

When the semantics of your type makes passing of resource meaningful (type **(2)**), or you can improve efficiency (type **(1)**).

Type **1** applies to types that have most of their data on the heap.

`X&&` will convert into `const X&` if no function fits to `X&&`, so nothing bad will happen if you don't define moving operations.

std::move

Suppose you have the following class:

```
struct stringpair
{
   string s1;
   string s2;

   stringpair( stringpair&& p )
      : s1{ std::move( p.s1 ) },
        s2{ std::move( p.s2 ) }
    { }
}
```

Without `std::move`, the compiler wouldn't be able use rvalues for `t.s1` and `t.s2`, because it is afraid to guess when is the last use.

Moving Constructor should not throw

If you define a moving constructor of type `X( X&& )`, then declare it `noexcept`. This is nearly always possible, because it doesn't allocate anything.

The advantage of `noexcept` is that `std::vector<>` will use it when reallocating.

Final Remarks

- Write rvalue methods only when you think that it gains something. If you don't write them, usual methods will be used.
- If you redefine one standard operator, redefine them all. (You can use `= default`.)
- Copying assignment can be implemented through rvalue assignment. This may become (but it is too early to say this in general) the best way to implement assignment in the future.
- Never write `const X&&`. It makes no sense.
- If you write an rvalue method, check that moving semantics is really used. It is easy to forget an `std::move( )` somewhere.
- Rvalue copy constructors and assignments should be `noexcept`, if possible.