Temporaries, Return Value Optimization, Rvalue References

In these slides, I discuss three related topics:

1. Temporaries in expression evaluation.

2. The return value optimization (RVO).

3. Rvalue references.

Translation of Function Calls

Consider a function definition

```
X f( Y1 y1, ..., Yn yn ) { .......   return x; }
```

The compiler creates a procedure of form

```
f( X* res, Y1 y1, ... Yn yn ).
```

Pointer `X* res` points to a memory location that has place to hold an `X`. When the function knows the result, it initializes `*res` and returns.

The calling context decides where the results comes.

Translation of a function call `f(t1,t2)`

1. The context of expression `f(t1,t2)` has already decided where the result will be written: Into a just declared variable, a temporary variable, or a local variable of the next function.

2. Reserve space for the local variable `y2` on the stack.

3. Call the translation of `t2`. The result will be written into `y2`.

4. Reserve space for the local variable `y1` on the stack.

5. Call the translation of `t1`. The result will be written into `y1`.

6. Call `f( res, &y1 )` (`f` knows that `y2` comes after `y1`.)

7. Create code that cleans up the local variables `y1,y2`. If `Y1,Y2` have destructors, they are called.

Intermediate Results

Consider a function

```
X g1( X x )
   ==>
g1( X* res, X* x )
{
   // If there are other arguments, they are behind X.
   // If necessary, we can reserve more space on the stack.
   ...
   *res = ... (returning is initialization.)

   // If necessary, clean up stack space.

   return;
};
```

4

```
X g2( const X& x );
   ==>
g2( X* res, X** x )
{
   // At low level, reference is the same as pointer.
   // const can be forgotten, once it was checked.


   ...
   *res = ... (returning is initialization.)
   // Clean up stack space, if necessary.
   return;
};
```

```
X& g3( const X& x );
    ==>
g3( X** res, X** x )
{
    ...
    *res = ... (pointer to an X.)
    // Clean up, if necessary.
    return;
};
```

Translating `X x1 = g1(g2(x2))` is unproblematic.

Translating `g1(g3(x))` is also fine, but one needs a copy constructor between `g1` and `g3`:

```
X( X& );
==>
X_copy_constructor( X* res, X** x )
{
    (Copy *x into res.)
    return;
};
```

The compiler first replaces `g1(g3(x))` by `g1(X(g3(x)))`, and then translates as before. If class `X` has no copy constructor, the expression cannot be compiled.

Temporaries

What about `g2(g2(x))`?

(Inner) function `g2` needs to have a place where it can write its `X`.

The local variables of (outer) `g2` only have place for an `X*`.

One needs a temporary place to store an `X`, so that its address can be passed to `gg`.

Such intermediate place is called a temporary.

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 complete? 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 innner 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.

C^{++} uses (2).

Temporary Values in Expressions

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

Assume that we have declarations

```
bigint operator + ( const bigint& n1, const bigint& n2 );
void operator = ( bigint& n1, 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;
```

Problems with Return Values

Consider an implementation of `operator *` on the previous slide.

```
bigint operator * ( const bigint& n1, const bigint& n2 )
{
    bigint res;
    ... (some complicated computation)
    return res;
}
times( bigint* res, bigint* n1, bigint* n2 ) {
    bigint res;
    ...
    *res = bigint(res);    // Call of copy constructor.
}
```

Variable `res` was created after calling *, the place for the result was certainly created before calling *, so they are necessarily different.

Return Value Optimization

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 position for 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^{++}.

Rvalue References

Despite the return value optimization, it can still happen that
return values have to be copied.

```
bignum gcd( bignum b1, bignum b2 )
{
   if( b1 < 0 ) b1 = -b1;
   if( b2 < 0 ) b2 = -b2;
   while( true )
   {
      if( b1 > b2 ) b1 = b1 - b2;
      if( b2 > b1 ) b2 = b2 - b1;
      if( b1 == 0 ) return b2; // Not local variable, and
      if( b2 == 0 ) return b1; // unpredictable which.
   }
}
```

Moving

What happens? Programmers start worrying about this, possibly they will write ugly code to avoid the copying.

This is incompatible with the main goals of C^{++} : Avoid the dilemma between good code and efficient code.

Many big objects (e.g. our **string** class, and probably also **bignum**) have their main data on the heap.

```
struct bignum
{
    unsigned int* val;
        // True representation is on the heap.
};
```

Why not simply pass the pointer? $\Rightarrow$ Because it messes up unique ownership invariant.

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?

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.

Non-pointer fields can have arbitrary values.

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, assignment can be implemented by exchange:

```
void operator = ( stack&& s )
{
    std::swap( tab, s. tab );
    current_size = s. current_size;
    current_capacity = s. current_capacity;
}
```

This method breaks the class invariant of s

Will it work?

It works fine for destructor, but for assignment, it depends on the implementation.

Ok:

```
void operator = ( const stack& s )
{
   if( tab != s. tab )
   {
      delete[] tab;
      tab = new double[ s. current_size ];
      (copy s.tab into tab)
      current_capacity = s. current_capacity;
      current_size = s. current_size;
   }
}
```

Wrong:

```
void operator = ( const stack& s )
{
   if( current_capacity < s. current_size )
   {
      delete[] tab;
      tab = new double[ s. current_size ];
      current_capacity = s. current_size;
   }
   (copy s. tab into tab)
}
```

May crash, because `current_capacity` does not correspond to true size of `tab`.

Recommendation: Rvalue methods shouldn't break the class invariants too much. It is possible, but dangerous. Don't try to find the border!

Do a complete exchange:

```
void operator = ( stack&& s )
{
    std::swap( current_size, s. current_size );
    std::swap( current_capacity, s. current_capacity );
    std::swap( tab, s. tab );
}
```

If exchange is more costly than simple assignment (this applies to simple structs without pointers), then don't write Rvalue methods.

X&& will convert into const X&.

Rvalue Copy Constructor

```
stack( stack&& s )
    : current_size{ s. current_size },
      current_capacity{ s. current_capacity },
      tab{ s. tab }
{
    s. current_size = 0;
    s. current_capacity = 0;
    s. tab = nullptr;
}
```

(Can be viewed as exchange with empty object.)

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

Difference between Nullpointer and Pointer to Memory Segment of length 0

It turns out that a pointer to a heap array of size 0 is almost indistinguishable from the null pointer:

- For every value of variable i, `p[i]` is undefined.

- `delete p` works on both of them. (Because `delete` ignores the null pointer.)

It follows that the second implementation is also possible.

You will find such code sometimes in examples. It looks like destruction of `s`, but it is not!

Rvalue methods cannot destroy! They must preserve allocation invariants.

A Different View of Assignment

Before, we used:

$$\mathbf{assignment} = \mathbf{destructor} + \mathbf{copy\ constructor}.$$

```
void operator = ( const stack& s )
{
   if( tab != s. tab )
   {
      delete[] tab;
      // Now *this is unitialized, proceed as in CC:
      tab = new double[ s. current_size ];
      (copy contents from s. tab to tab.)
   }
}
```

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 stack& s )
{
    *this = stack(s);
        // CC makes a copy which will be passed as
        // Rvalue reference to operator = ( stack&& ).
}
```

1. No self-assignment or subtree assignment problem.

2. Less repeated code.

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

Creation of Rvalue References

Rvalue references are automatically created by the compiler in the following situations:

- When a reference to a temporary is created.

- When a `return` statement returns a local variable, and the return value optimization cannot be used, the compiler tries to find a copy constructor that has an Rvalue argument.

In all other cases, you have to write `std::move( )` if you want an Rvalue reference. Note `std::move( )` is just a cast from `X&` to `X&&`.

std::move

Suppose you have the following (ridiculous) class:

```
struct twostacks
{
    stack s1;
    stack s2;

    twostacks( twostacks&& t )
        : s1{ std::move( t. s1 ) },
          s2{ std::move( t. s2 ) }
      { }
}
```

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

Moving Constructor must 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.

Automated Generation of Copy/Move Operators

The compiler automatically generates default copy/move operators, whenever the types of the fields allow this: If all fields can be copied/moved/assigned/move assigned, the corresponding operator will be generated.

There is one complication: You are not supposed to use these default operators, as soon as you have defined one copy/move/assignment/moving assignment operator by yourself, or a destructor.

What is going on here?

The standard committee would have prefered to delete the defaults when you define an operator by yourself, but was afraid to break existing code.

Automated Generation of Copy/Move Operators

Existence of one user copy/move/assignment/moving assignment or destructor implies that the class has non-standard resource invariants. Because of this, very probably all 5 operators will be non-standard, and they should not be defaults.

But as said before, the standard committee didn't want to break existing code when move semantics was added in 2011.

Be a good boy/girl, and define all operators when you define one.

Default Definitions

Assignment operators and constructors can be explicitly defined as default. This is useful when you define some of them by yourself, while others are still default.

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

If you want that your class does not have any of these operators, you can use `= delete;`.

Implementation of std::swap

```
template <class T> void swap( T& a, T& b )
{
    T c = std::move(a);
    a = std::move(b);
    b = std::move(c);
}
```

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 Rvalues are really used. It is easy to forget an `std::move( )` somewhere on the way.

- Rvalue copy constructors and assignments must be `noexcept`.