Exercise Compiler Construction (8)

Hans de Nivelle

Due: December 16th 2009

The first step of C/C^{++}-compilation is to make the implicit calls of constructors/adress operators explicit, and to select the right version of polymorphic operators. The purpose of this exercise is to get some feeling for this.

1. Give type expressions for the following C^{++} declarations. (Use refto for references type)

```
int p;
int *p;
int * const p;
const int* p;

const std::string& s;
   // You may assume that std::string is a primitive type.
   // It is some kind of struct/class and these have to be treated
   // as primitive.
std::string s[100];
```

2. Consider the following declarations,

```
class X
{
   X( int );       // constructor from int.
   X( const X& );  // copy constructor.
   X& operator = ( const X& );
   X& operator ++ ( ) ;          // ++ x;
   X operator ++ ( int );        // x ++ ;
};
std::ostream& operator << ( std::ostream&, const X& );
X operator + ( X, X );
bool operator == ( const X&, const X& );
```

In the following translation of the declarations, member functions have received an explicit `this` argument, and the C^{++}-types have been replaced by the types of Defininition 1.1. In order to make things not too complicated, we ignore **const**ness.

```
X Xconst( int x );               // constructor from int.
X Xconst( refpointerto(X) x );   // the copy constructor for X.
refpointerto(X) operator= ( pointerto(X) this, refpointerto(X) x );
refpointerto(X) operator++( pointerto(X) this );
X operator++( pointerto(X) this );

refpointer( std::ostream ) operator<< ( refpointer(std::ostream),
                                        refpointer(X) );
X operator+ ( X, X );
bool operator == ( refpointer(X), refpointer(X) );

refpointerto(X) makeref(X)
pointerto(X) makepointer( X ).
   // Functions that takes adres of X, sometimes inserted by compiler.
pointerto(X) makepointer( refpointerto(X))
   // Function that makes a usual pointer from a ref pointer.
   // The function is inserted by the compiler. It actually does
   // nothing.
```

Here is an example of how C^{++} the implicit conversions could be made explicit in some C^{++}-statements:

```
X x1;
X x2;

x2 = 3;  // operator=( makepointer(x2),
                       makeref( Xconst(3) ));
x1 = x2; // operator=( makepointer(x1), x2 );
```

Variables `x1,x2,`are of type refpointer(X). The constant 3 is of type int.

(a) Why does `operator++( )` have type `X&`, while operator `operator++(int)` has type `X`?

(b) Write translations for the following statements:

```
X x;
std::cout << ( x ++ );
std::cout << ( ++ x );
std::cout << ( x + x );
```

```
X x1 x2; // Don't translate the declarations.

x = ( x1 = ( x2 ++ ));
   // No sane programmer would write this.
if( x == f(x1,x2))  ...
```

3. In *C*, there is a quite intricate relation between pointers and arrays. The difference can be understood as follows: Each of the primitive data types `X` has a a copy constructor that transforms `const X&` or `X&` into `X`. Array types do not have copy constructors by default. Instead, arrays have a conversion operator from `(X[])&` to `X*`

```
int intcopy( refpointerto(int) );
   // Copy constructor for int.

pointerto(int) pointercopy( refpointerto( pointerto(int ));
   // Copy constructor for pointer to int.

pointerto(int) arraycopy( refpointer( array(N,int)));
   // Conversion that transforms array into pointer.

refpointerto(int) operator[] ( pointerto(int), int );
   // Indexing operator [].

(Some more functions will be needed)


int * * p1;
int *( p2[] );
int (*p3) [] ;
int p4 [ ] [] ;

p1[1][2];
p2[1][1];
p3[2][2];
p4[2][3];
```

   (a) Make the types of `p1,p2,p3,p4` explicit.

   (b) Make the four indexing expressions explicit. Give the declarations of the missing indexing and conversion (from refpointerto to pointerto) operators.