Exercise Compiler Construction (List 8)

Hans de Nivelle

16.12.2015

1. Give type expressions (using **pntr**, **ref**, **const**) for the following C^{++} declarations. You may treat string as a primitive type. (Because classes and structs are treated as primitive.)

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

const char* s1 = "Warum ist ueberhaupt Seiendes";
std::string s2 = " und nicht vielmehr Nichts?";
const std::string& s3 = s2;

   // 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 s4[10][50];
```

2. In C, there is a subtle relation between pointers and arrays because arrays are always implicitly converted into pointers. We are going to study this process. Consider the following operators:

```
load<T> : func( T, ref( const( T ))
   // Load (Copy constructor) for primitive types and
   // for pointers.

assign<T> : func( T, ref(T), T );
   // Assignment operator, should be available for
   // primitive types and pointers.

array2pntr<T> : func( pntr(T), ref( array( n, T )) );
array2pntr<T> : func( pntr(T), ref( array(T)) );
   // Conversion operator that transforms an array of T into
   // a pntr to T.
```

```
star<T> :  func( ref(T), pntr(T) );
   // unary * changes pointer into reference.

addr<T> : func( pntr(T), ref(T) );
   // Unary & changes reference into pointer.
   // (not needed in the exercise)

addpntr<T> : func( pntr(T), pntr(T), int/unsigned/size_t );
   // + on pointers.

operator << : func( ref(ostream), ref(ostream), int );
operator << : func( ref(ostream), ref(ostream), pntr( const(char)) );
```

(a) Give the types of `p1,p2,p3,p4`, using the notation on the slides.

```
int ** p1;
int *( p2[] );
int (*p3) [10] ;
int p4 [5][6] ;
```

(b) Write the following expressions as tree, type check them, insert the conversions, and resolve the overloads. Assume that `p[i]` means `*(p+i)`.

```
std::cout << p1[1][2];
std::cout << p2[3][4];
std::cout << p3[5][6];
std::cout << p4[7][i+j];
```

3. Consider the following declarations: (think of `X` as some type of big integer.)

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

X operator + ( X, X );
X operator * ( X, int );
X operator * ( int, X );
```

```
bool operator == ( const X&, const X& );
bool operator != ( const X&, const X& );
```

(a) Give the explicit types of these declarations, using the notation on the slides. A constructor of class `X` is just a function returning an `X`. Members of a class `X` (that are not constructors) have an additional argument of type `ref(X)`.

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

(c) Type check (write them as trees, resolve the overloads, insert the conversions) the following expressions:

```
X x,y;
std::cout << ( x ++ ) << "\n";
std::cout << x << "\n";
std::cout << x << " " << y << "\n";
```

(d) Also type check the following list of expressions. The expressions use temporaries. Assume a box operator `box<T> : func( ref(const(T)), T )`, which transforms a value into a const reference. (In order to make things not too complicated, we ignore rvalue references.)

```
std::cout << ( x + x ) << "\n";
std::cout << ( 2 * x ) << "\n";
std::cout << ( x + ( x + 4 )) << "\n";

x = ( y = ( x ++ ));
   // No sane programmer should ever write this.
if( x == (x+y) ) std::cout << x;
```