Subject: Question about pick behaviour
Posted by ross_tang on Mon, 28 Mar 2011 09:25:05 GMT
View Forum Message <> Reply to Message

(I am not sure if the question should be posted here.)

From this two articles:
i. Pick Behaviour Explained
ii. Transfer semantics

It seems to me that one main problem 'pick' solves is the return of an complex object. As in this example in Transfer semantics:

```
class IntArray {
    int count;
    int *array;
    void Copy(const IntArray& src) {
        array = new int[count = src.count];
        memcpy(array, src.array, count * sizeof(int));
    }
public:
    int& operator[](int i)       { return array[i]; }
    int  GetCount() const        { return count; }
    IntArray(int n)              { count = n; array = new int[n]; }
    IntArray(const IntArray& src) { Copy(src); }
    IntArray& operator=(const IntArray& src)
                                 { delete[] array; Copy(src) }
    ~IntArray()                  { delete[] array; }
};

IntArray MakeArray(int n) {
    IntArray a(n);
    for(int i = 0; i < n; i++)
        a[i] = i;
    return a;
}
```

If we run this step: IntArray y = MakeArray(1000), it would generate an unnecessary deep copy.

Then the article proceeded to this:

```
class IntArray {
    int count;
    mutable int *array;
    void Pick(pick_ IntArray& src) {
        count = src.count;
        array = src.array;
        src.array = NULL;
```

```
    }
public:
    int& operator[](int i)        { return array[i]; }
    int  GetCount() const         { return count; }
    IntArray(int n)               { count = n; array = new int[n]; }
    IntArray(pick_ IntArray& src) { Pick(src); }
    IntArray& operator=(pick_ IntArray& src)
                                  { if(array) delete[] array;
                                    Pick(src) }
    ~IntArray()                   { if(array) delete[] array; }
};
```

which uses the pick semantics and avoided deep copy in function return. But it has the adverse effect of invaliding the original variable in an assignment.

However I think it is possible to preserve the original variable, while retaining the pick semantics. We can just add a flag to the object to indicate if the object is picked or not.

```
class IntArray {
    int count;
    bool picked;
    mutable int *array;
    void Pick(pick_ IntArray& src) {
        picked = src.picked;  /* updated from picked = false, since src maybe picked as well. In
principle, we only want one instance of the
 object unpicked while all others are picked. In this way, only the destruction of the unpicked one
would deallocate all memory in used. */
        src.picked = true;
        count = src.count;
        array = src.array;
    }
public:
    int& operator[](int i)        { return array[i]; }
    int  GetCount() const         { return count; }
    IntArray(int n)               { count = n; array = new int[n];picked = false;}
    IntArray(pick_ IntArray& src) { Pick(src); }
    IntArray& operator=(pick_ IntArray& src)
                                  { if(!picked) delete[] array;
                                    Pick(src) }
    ~IntArray()                   { if(!picked) delete[] array; }
};

IntArray MakeArray(int n) {
    IntArray a(n);
    for(int i = 0; i < n; i++)
        a[i] = i;
    return a;
}
```

If we use this method, first the original variable won't be changed except the picked flag. Next if we run:

IntArray y = MakeArray(1000);

the field array of IntArray in a won't be deallocated, since the destructor only deallocate if it is not picked.

Python uses reference extensively. For example:

```
a = [1,2,3]
b = a
a[1] = 10
```

If we run the above code, it is totally valid(But in U++, it is an error). And now b is [1, 10, 3] since it shares the same data with a. And I expected U++ to do the same too.

I hope someone can let me know if I understand 'pick' correctly or wrong.
