USING ALGORITHMS

Himanshu Maurya / November 20, 2015

AUDIENCE

- C++ programmers
- Python programmers
- Clean code enthusiasts

OBJECTIVE

- Better code
- Avoid raw loops
- Use standards
- DRY

AVOID RAW LOOPS

Why? not because loop is hard to write
or you cant write terse loop in c++

```
std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; //this line wil
//before c++11
std::vector<int>::iterator it(v.begin());
for (; it != v.end(); ++it ) {
    //do
    std::cout <<"val:" << *it <<std::endl;
    //logic
}
```

```
//c++11 onwards
std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for (auto it : v) {
    //do
    std::cout << "val: " << it <<std::endl;
    //logic
}
```

WHY NO RAW LOOPS?

...difficult to reason about and prove the post conditions

...error prone

...complicates reasoning about surrounding code

...can also cause non-obvious performance problem

TINY INTRODUCTION TO LAMBDA

very tiny infact.. but it will serve the purpose..
basically lambdas are a way to create function objects
without the extra boiler plate code

FUNCTIONS

This is how we declare regular functions body

```
return_type func(params type) { body }
```

in c++

```
int increment(int aNum) { return aNum+1; }
```

remove the function name because its lambda

```
int anonymous(int aNum) { return aNum+1; }
```

remove the return type because compiler can deduce it

```
anonymous(int aNum) { return aNum+1; }
```

remove the name

```
[ capture_list ](int aNum) { return aNum+1; }
```

[caputure_list] is a way to create closures

```
{
    int a = 1, b = 1, c = 1;
    [ a, &b, c=myC ](int aNum) { return a + b + myC + aNum; }
}
```

lambda in c++

```
[ capture_list ] (int aNum) { return aNum+1; }
```

That's it, time to go back to problems.

PROBLEMS

let's start..

PROBLEM 1

A list of widgets

```
Vector<Widgets*> widgetList;
```

Passed to us by some Api e.g. could contain a selected widgets by user or a list of assets loaded in premo

And this Widget contains some attributes that we are interested and we want to filter them out

PROBLEM 1_A

Find that list should not contain a un-selected widget (a test case may be)

SOLUTION 1_A

Use std::find_if_not
(return the item where the predicate return false)

```
auto resultIter = std::find_if_not(widgets.cbegin(),
                  widgets.cend(),
                  [ ](Widgets *wid) { return wid->selected(); });

assert(resultIter == widgets.end());
```

PROBLEM 1_B

Count the number of selected widgets in the list

SOLUTION 1_B

Use std::count_if

```
auto selCount = std::count_if( widgets.begin(),
                widgets.end(),
                [ ](Widgets *wid) { return wid->selected(); });
```

PROBLEM 1_C

Given the list of widgets create a new list from the original list based on given predicate
e.g. create a list of widgets from the given list if visible only

SOLUTION 1_C

Use std::copy_if

```
Vector<Widgets*> srcList;
//make sure dstList is atleast the size of srcList
Vector<Widgets*> dstList(srcList.size());
auto it = std::copy_if (srcList.cbegin(), srcList.cend(),
          dstList.begin(),
          [ ](Widgets *wid) { return  wid->isVisible(); });
std::resize(std::distance(dstList.begin(), it);
```

PROBLEM 1_D

Given a list of widget you want to sum or accumulate some numeric property
e.g. construct a bounding rectangle from all the widgets in the list

SOLUTION 1_D

Use std::accumulate

a simple ex: use default +operator

```
std::vector<int> int_list {1,2,3,4,5,6,7,8,9,10};
auto sum = std::accumulate(int_list.begin(),
           int_list.end(),
           0 /*init value*/);
```

SOLUTION 1_D ..

use provided operator

```
std::vector<Widget *wid> widgets;
Rect boundingRect = std::accumulate(widgets.begin(),
                    widgets.end(),
                    Rect(),
                    [ ] (Rect a, Rect b) { return a |= b; });
```

PROBLEM 1_E

Remove certain elements from list
e.g. remove selected elements from given widget list (very frequent operation e.g. user moved some selected widgets in the gui)

SOLUTION 1_E

Use std::remove_if (but this does not remove elements ..hmm..)

```
std::vector<Widget*> list;
auto it = std::remove_if(list.begin(),
          list.end(),
          [ ](Widget* wid) { return wid->selected(); })
list.erase(it, list.end());
```

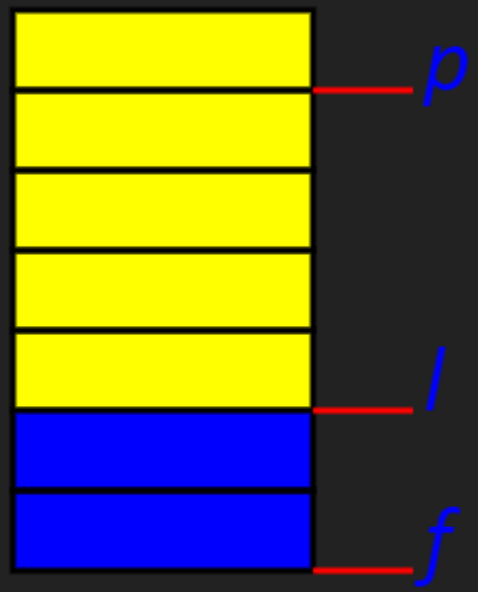
ALL_OF, NONE_OF, ANY_OF

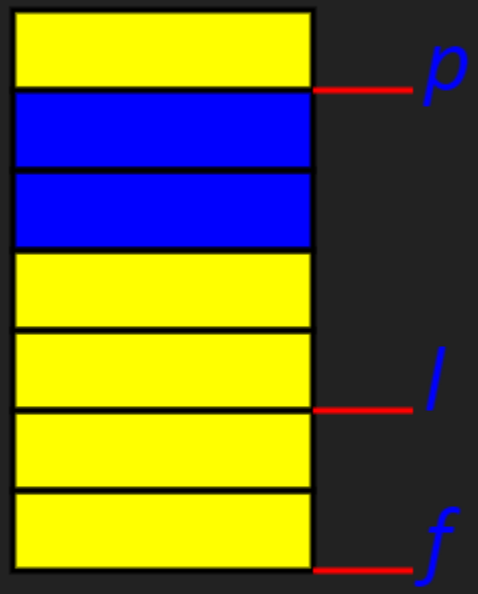
apply some predicate on list

```
std::vector<Widget*> list;
auto it = std::all_of(list.begin(),
          list.end(),
          [ ](Widget* wid) { return wid->isInitialized(); });
```

NEXT PROBLEM IS TAKEN FROM SEAN PARENT'S TALK ON CPP SEASONING

This problem comes in one form or other.
coincidentally when I watched this I was around the same problem with free pages in premo

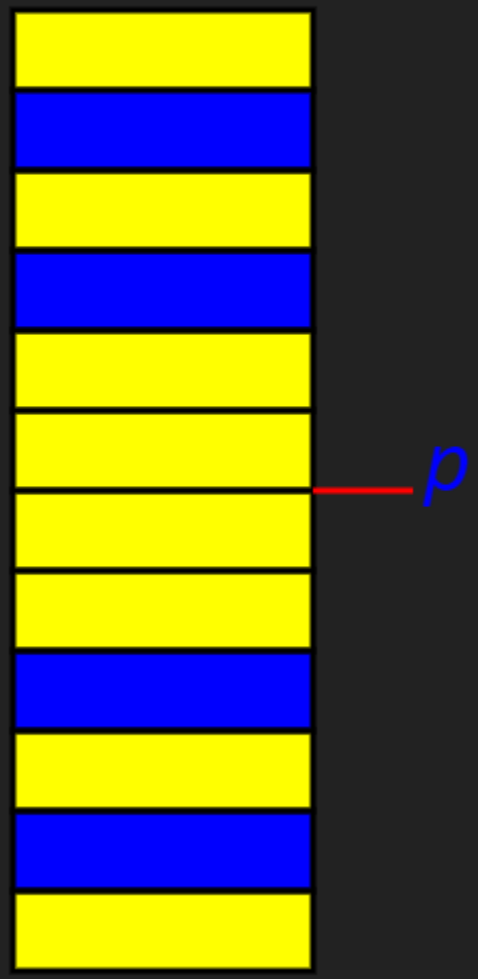

p
l
f

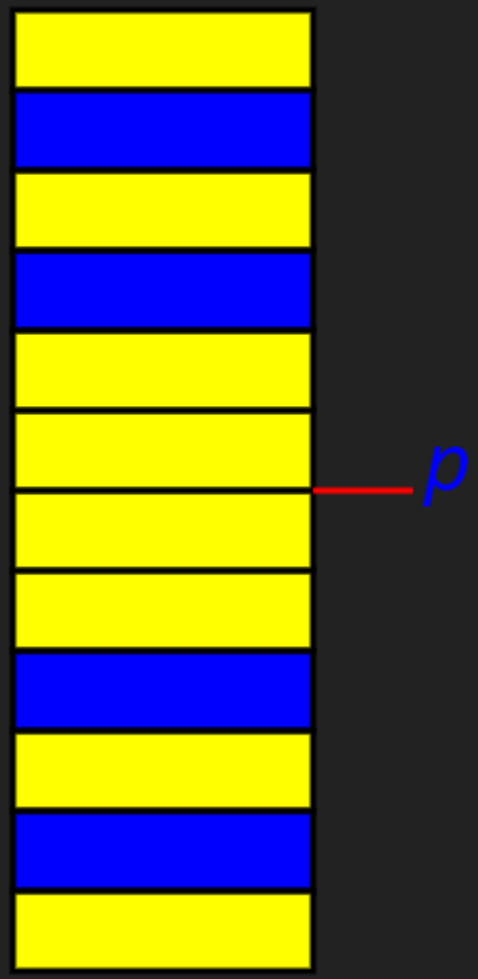

p
l
f

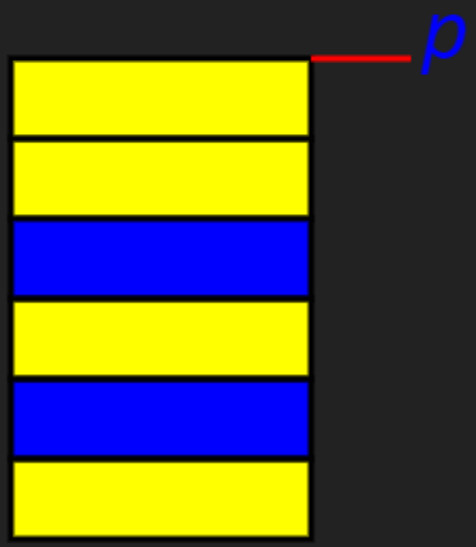
USE STD::ROTATE

std::rotate(first, position, last) and returns a iterator to first + (last - position).

```
std::vector<Widget*> list;
auto it = std::rotate(f, l, p);
//f = iterator to begining of the list
//l = iterator to selected element
//p = iterator to the destination
```

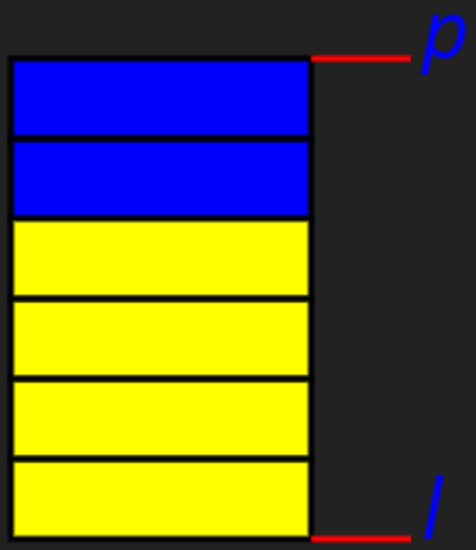

p


p


p

USE STD::STABLE_PARTITION

```
std::vector<Widget*> list;
auto it = std::stable_partition(p, l,
        [] (Widgets* wid) { return wid->isSelected();});
//p = iterator to begining of the list
//l = iterator to selected element
```


p
l

USE GATHER

is present in boost lib

```
template <typename I, // I models BidirectionalIterator
          typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> std::pair<I, I>
{
     using value_type = typename std::iterator_traits<I>::value_type;
     auto itFirst = std::stable_partition(f, p,
                    [&](const value_type& x){ return !s(x); });
     auto itSecond = std::stable_partition(p, l, s);
     return std::make_pair(itFirst, itSecond);

}
```

EXAMPLE

```
std::vector<int> list{1,2,4,5,0,6,9,0,0,11,0};

auto isZero = [](int a) { return a == 0;};

gather(list.begin(), list.end(), std::next(it,3), isZero);
```

```
//output
1 2 4 0 0 0 0 5 6 9 11
```

ALTERNATIVES TO RAW LOOPS

- Use an existing algorithm
  - prefer standard library if available
- Implement known algorithm as a function

THE END