ENGR101 T1 2024

Cleaner C++ for big programs

Arthur Roberts

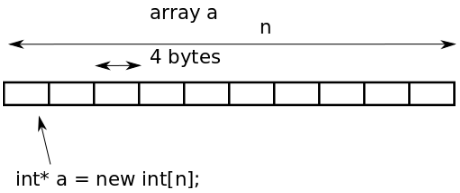
School of Engineering and Computer Science
Victoria University of Wellington

Today

We will touch a bit on some practical C++ tricks to make code shorter. C++ keeps changing and in last 10 years it moved very far from what we used so far.
We will talk about some developments from 90s:

- Containers, **Standard Template Library, STL**
- STL functions
- Functional approach

Reminder: arrays in memory, pointers



- **a** is declared as a pointer(memory address) - type **int* a**
- If we print it we get number of memory cell/address (0x5635e7c3ae70)
- If we do **a=a+1** pointer (i.e. address) will be incremented by size of each element in the array (4 if size of each element is 4 - int).
- ***(a+1)** gives value (*) at address (a+1)

file a0.cpp

Reminder: Vectors

Once program reserved memory (new int[n]), its size can not be changed. Can be bad thing - Project 1 had to read file twice to determine how many notes are there.

There is more flexible alternative to the array - **vector.** To use vectors:

- Include vector library: **#include <vector>**
- Declare vector of size 0: **std::vector<int> a;**
- Declare empty vector of some size: **std::vector<int> a(5);**

What you can do with vectors?

- add elements to the end of it - **a.push_back(new_element)**
- remove all elements from it - **v.clear()**
- get one element at position i - **v.at()** or **v[]**
- get the size of it - **a.size()**
- remove one elements at certain position - **v.erase(v.begin()+3)** (more on begin() function shortly)
- Does not check for vector size when accessing element (still unsafe)

file v0

Containers

You can see some similarity between **array** and **vector**. They both belong to same class - called **container**.
Often you have to do similar things for the container. There are many other containers.

- set
- map
- list
- queu
- etc ...

STL algorithms work for all containers no matter what type of the element it contains.

Standard Template Library

What is motivation for STL?
We have to do code same algorithms again and again: find_max() for doubles, the another one for int, another for pixels, then another one for strings.
We do not want to give up types, because types make programming much safer, but we do not want to have many versions of same code either.
STL is made of:

- containers
- iterators
- algorithms

Algorithms can be applied to any type.

STL: Iterator - replacement fot [i]

Iterator is a generalization of a pointer (memory address).

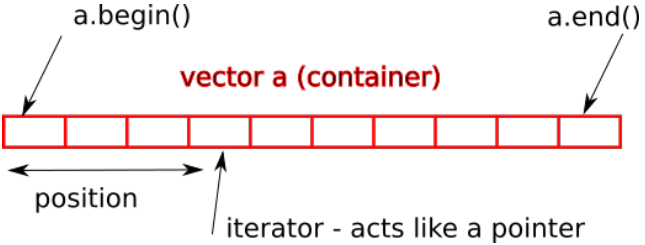


Figure: Iterator

Simpler access to container element - use star (*) in front of the iterator.

STL: How to use an iterator?

Listing 1: iterator

```
// includes
int main() {
    // Declaring a vector
    vector<int> v = { 1, 2, 3,4,5,3,2,1 };
  // Declaring an iterator - will point to the element of vector of ints
    vector<int>::iterator i;
    cout <<std::endl<<"*(v.begin()+3)="<< *(v.begin() + 3);
    // Accessing the elements using iterators
    for (i = v.begin(); i != v.end(); ++i) {
        cout << *i << "␣";
    }
}
```

Notice * - used when we need value in memory which iterator/pointer is pointing to. Why such complication? Container cam contain variables of any type, so using address makes sense.
iterator

STL: Algorithms

There are many **algorithms** in STL. We will go through some which can be useful for Project 3.

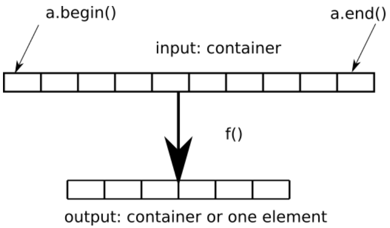


Figure:

- There is an input container
- Function **f()** is applied to the elements of the input container
- Result goes into another output container or it can be single value

Short examples as well as Project 2 done this way.

STL: Algorithms, **for_each**, example

Example of the algorithm usage:
Apply function to **each** element of the container.

Listing 2: for_each

```
#include <vector>
#include <iostream>
#include <algorithm>

void print_double(double ii){ std::cout<<ii<<"␣";}

int main(){
        std::vector<double> a = {0, 0.1,0.2,0.3,0.4,0.5};
        std::for_each(a.begin(),a.end(),print_double);
}
```

v3_for_each

for_each: Function as an argument for another function

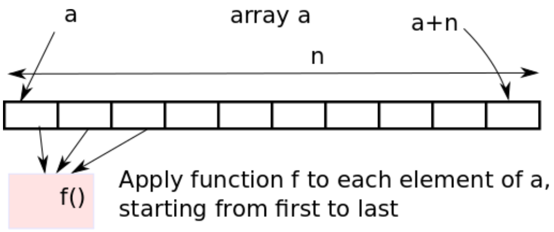

a
array a
a+n
n
f()
Apply function f to each element of a, starting from first to last

Figure: Function applied to all elements of the array

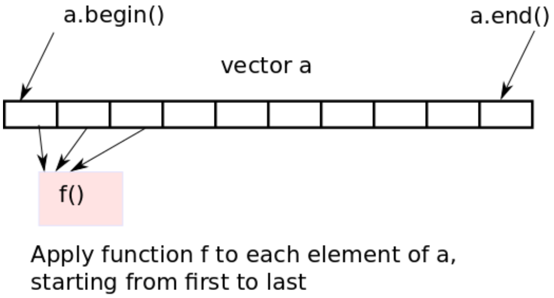
Listing 3: Caption

```
std::for_each(a,a+n,print_double);
```

That is the pattern common for STL - function **print_double** is passed to another function - **for_each** here.

for_each also works for vectors

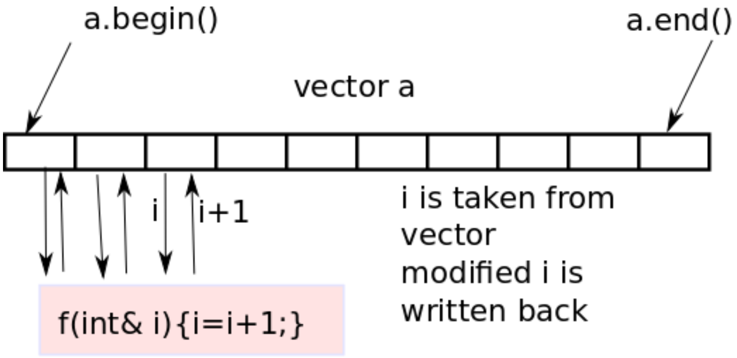
For vectors we can use **begin()** and **end()** - less things to worry about.


a.begin()
a.end()
vector a
f()
Apply function f to each element of a,
starting from first to last

Listing 4: for_each for vectors

```
void print_int(int ii){
  std::cout<<ii<<"_";
}
// ...
std::vector<int> a;
std::for_each(a.begin(),a.end(),print_int);
```

for_each: modify vector in-place



Listing 5: argument by reference

```
void add_one(int& ii){ ii = ii+1;}
// ...
std::for_each(a.begin(), a.end(), add_one);
```

for_each: modify vector in-place

Can **for_each()** be used to modify the vector in place? Pass argument by reference.

Listing 6: Modify in-place

```
#include <vector>
#include <iostream>
#include <algorithm>

void print_double(double ii){ std::cout<<ii<<" ";}
void add_one(double& ii){ ii=ii+1;}

int main(){
        std::vector<double> a = {0.0, 0.1,0.2,0.3,0.4,0.5};
        std::for_each(a.begin(),a.end(),print_double);
        std::cout<<std::endl;
        std::for_each(a.begin(),a.end(),add_one);
        std::for_each(a.begin(),a.end(),print_double);
}
```

v3_for_each_mod.cpp

Algorithms: **count** specific elements in array/vector

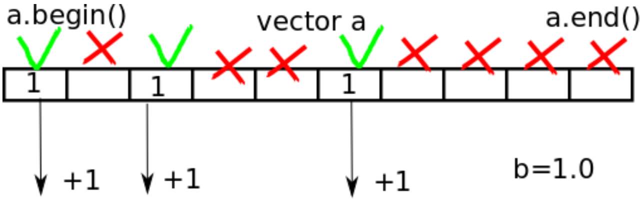
Listing 7: Counting number of elements with specific value

```
#include <vector>
#include <iostream>
#include <algorithm>

void print_double(double ii){ std::cout<<ii<<"␣";}
int main(){
 std::vector<double> a;
 a.push_back(0);              a.push_back(1);
 a.push_back(0); a.push_back(0);
 a.push_back(1);  a.push_back(1);
 std::for_each(a.begin(),a.end(),print_double);
 std::cout<<std::endl;
 std::cout<<std::count(a.begin(), a.end(), 1.0);
}
```

v3_count

count specific elements in array/vector



Listing 8: count

```
std::cout<<std::count(a.begin(), a.end(), b);
```

Counts number of elements im **a** value of which equals **b**

More general version of count: using custom condition of what to count

Listing 9: Counting elements which satisfy some criteria

```
#include <vector>
#include <iostream>
#include <algorithm>

bool gre(double b){return (b>1.0) ;}
void print_double(double ii){ std::cout<<ii<<"␣";}

int main(){
        std::vector<double> a = {0.0, 10.0, 20.0 , 0.0 ,3.0};
        std::for_each(a.begin(), a.end(), print_double);
        std::cout<<std::endl;
        std::cout<<std::count_if(a.begin(), a.end(), gre);
}
```

Counts number of elements for which function **pos()** return true.

v3_count_if

Transform one array/vector into another

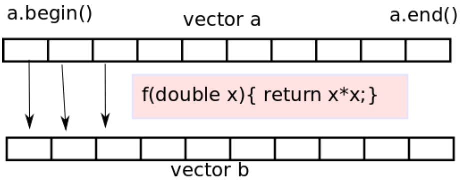
Listing 10: transform one vector into another

```
#include <vector>
#include <iostream>
#include <algorithm>

void print_double(double ii){ std::cout<<ii<<"␣";}
double square(double x){return x*x;}

int main(){
  std::vector<double> a;
  a.push_back(0);          a.push_back(0.1);
  a.push_back(0.2); a.push_back(0.3);
  a.push_back(0.4);  a.push_back(0.5);
  std::vector<double> b(a.size());
  std::transform(a.begin(), a.end(), b.begin(),square);
  std::for_each(b.begin(),b.end(),print_double);
}
```

Algorithms: **transform** one array/vector into another



Listing 11: Caption

```
double square(double x){return x*x;}
//...
 std::transform(a.begin(), a.end(), b.begin(),square);
```

- Take each element of **a** from **a.begin()** until **a.end()**.
- Apply function to it.
- Put result into **b**, starting from **b.begin()**.

One useful function - accumulate

This function requires **numeric** library.

Listing 12: accumulate

```
#include <iostream>       // std::cout
#include <algorithm>      // std::find_if
#include <vector>         // std::vector
#include <numeric>

int main () {
  std::vector<int> a;
  a.push_back(10);
  a.push_back(10);
  a.push_back(10);
  a.push_back(25);
  a.push_back(40);
  a.push_back(55);
  int result = std::accumulate(a.begin(),a.end(),0);
  std::cout<<"result␣=␣"<<result<<std::endl;
  return 0;
}
```

accumulate: adding all elements together

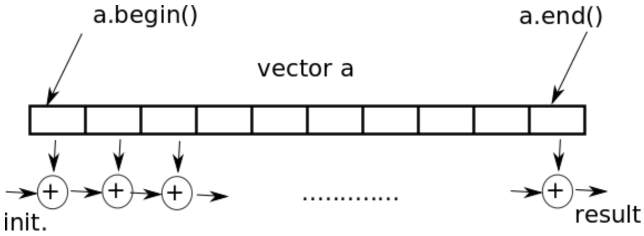


Figure:

Listing 13: accumulate

```
int result = std::accumulate(a.begin(),a.end(),0);
```

accumulate with custom function

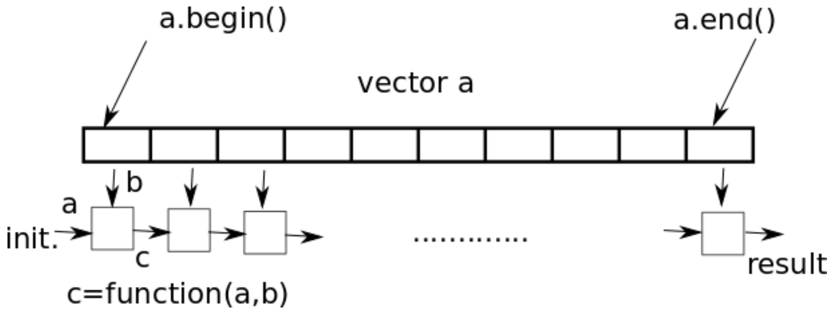
Listing 14: accumulate with custom function

```
#include <iostream>       // std::cout
#include <algorithm>      // std::find_if
#include <vector>         // std::vector
#include <numeric>

int f1(int a,int b){
return a*b;
}

int main () {
 std::vector<int> a;
 a.push_back(10);
 a.push_back(10);
 a.push_back(10);
 int result = std::accumulate(a.begin(),a.end(),1,f1);
 std::cout<<"result␣=␣"<<result<<std::endl;
 return 0;
}
```

accumulate with custom function



Listing 15: Caption

```
int function(int a, int b) { /* */  };
// ....
int result = std::accumulate(a.begin(),a.end(),init,function);
```

Find certain element

Listing 16: Caption

```
#include <iostream>      // std::cout
#include <algorithm>     // std::find_if
#include <vector>        // std::vector

bool is_odd(int i) { return ((i%2)==1); }

int main () {
 std::vector<int> a;
 a.push_back(10);  a.push_back(10);
 a.push_back(10);  a.push_back(25);
 a.push_back(40);  a.push_back(55);

 std::vector<int>::iterator it=std::find_if (a.begin(), a.end(), is_odd);
 std::cout << "The␣first␣odd␣value␣is␣" << *it << '\n';
 std::cout << "␣position="<<it-a.begin();
 return 0;
}
```

iota - fill vector

Listing 17: Caption

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <numeric>

int main(){
 std::vector<double> a(6);
 std::cout<<"␣size="<<a.size()<<std::endl;
 std::iota(a.begin(),a.end(),10.0);
 std::for_each(a.begin(), a.end(), [](double ae) {std::cout<<ae<<"␣";});
}
```

There is new thing in this listing: **[](double ae) std::cout«ae«" ";.**
What is that? iota.cpp

Functions which are used only once

You can notice that time and again in code above we write functions which are used only once.
Is it worth it to write separate function?
There is a shortcut for doing this. Instead of defining function explicitly

Listing 18: traditional

```
bool pos(double a){return a>0.0;}
//... something
std::cout<<std::count_if(a.begin(), a.end(),pos);
```

you can write

Listing 19: function without name: lambda

```
//... something
std::cout<<std::count_if(a.begin(),a.end(),[](double a){return a>0.0;});
```

This notation **[capture](argument){body}** is called **lambda** - function only used right there where it is defined.

Using lambdas - code without lambda

Listing 20: transform without lambdas

```
#include <vector>
#include <iostream>
#include <algorithm>

void print_double(double ii){ std::cout<<ii<<"␣";}

int main(){
  std::vector<double> a;
  a.push_back(0);            a.push_back(0.1);
  a.push_back(0.2); a.push_back(0.3);
  a.push_back(0.4);  a.push_back(0.5);
  std::for_each(a.begin(),a.end(),print_double);
}
```

Using lambdas - code with lambda

Listing 21: lambdas->shorter code

```
#include <vector>
#include <iostream>
#include <algorithm>
int main(){
  std::vector<double> a;
  a.push_back(0); a.push_back(0.1);
  a.push_back(0.2); a.push_back(0.3);
  a.push_back(0.4);  a.push_back(0.5);
  std::for_each(a.begin(),a.end(),
                 [](double a_ele){std::cout<<a_ele<<"␣";});
 }
```

Code is shorter, everything is local.

What about **[capture]**?

v2_lambdas.cpp

Capture - use variables from outside the lambda

Functions which use **begin()** and **end()** can take only one argument. What if we need more?

Listing 22: Value capture

```
#include <vector>
#include <iostream>
#include <algorithm>

int main(){
 std::vector<double> a;
 a.push_back(0);                a.push_back(1);
 a.push_back(0); a.push_back(1);
 a.push_back(1);  a.push_back(1);
 double thr = 0.5;
 std::cout<<std::count_if(a.begin(),a.end(),
                            [thr](double ae){return ae>thr;});
}
```

v3_countif_lambda.cpp

More modern C++

If you did the Challenges and looked at cpprefernce, stackoverflow and likes, you could notice that practically nobody actually writes the C++ code the way we did so far.
C++ keeps changing (C++11, C++17, C++20) and in last 10 years a lot of new features were added to make it more functional(?)
Two major types or programming languages.

- Program as a sequence of steps modifying variables
- Program as a sequence of expressions(functions). Minimum or no variables. Functions taking another functions as an arguments.

Questions?