Playfair Cipher

Mark Slater

UNIVERSITY OF BIRMINGHAM

Implementing the Playfair Cipher

- We will spend the afternoon going through the steps to implement another cipher for your library – the Playfair Cipher

- This is another plain text cipher that has similar restrictions as the Caesar Cipher but is more complex

- After setting up the initial class stub, we will then introduce Iterators, Algorithms and Lambdas and use these to implement the actual cipher

The Playfair Cipher

- The Playfair is based around the idea of exchanging pairs of letters based on the positions in a 5x5 grid

- The grid contains a key phrase with repeated letters removed and then any remaining letters of the alphabet not contained added to the end and J replaced with I

- As an example, below is the grid for the key 'Playfair Example':

P L A Y F A
I R E X A M PLE A
B C D EF G H I=J
K LM N O P Q R S
T U V W XY Z

The Playfair Cipher

- To encrypt a message, the following is applied:
    - ➜ Any repeated characters in a pair are separated by 'X' or a 'Q' if the pair is already 'XX'
    - ➜ If there are an odd number of characters, a 'Z' is appended
    - ➜ The message is then broken down into pairs of letters ('Digraphs')
- The following is an example:

Hello World → HE LX LO WO RL DZ

Letters split into pairs

X added between the double 'L's of the digraph

Z appended as odd number of characters

The Playfair Cipher

- After this preparation, the letters in a Digraph are found on the 5x5 table and the following rules applied:

  1. If letters are on the same row, replace with letter to the right

  2. If in same column, replace with letters directly below

  3. If they form a rectangle, replace with ones from corner on the same row

- To decrypt, simply use the inverse of these 3 rules

P	L	A	Y	F
I	R	E › X › M		
B	C	D	G	H
K	N	O	Q	S
T	U	V	W	Z

EX

Shape: Row
Rule: Pick Items to Right of Each Letter, Wrap to Left if Needed

XM

P	L	A	Y	F
I	R	E	X	M
B	C	D	G	H
K	N	O	Q	S
T	U	V	W	Z

DE

Shape: Column
Rule: Pick Items Below Each Letter, Wrap to Top if Needed

OD

P	L	A	Y	F
I	R	E	X	M
B	C	D	G	H
K	N	O	Q	S
T	U	V	W	Z

HI

Shape: Rectangle
Rule: Pick Same Rows, Opposite Corners

BM

Exercise 1 – Add Playfair Boiler Plate

- In the repo for today, we have added an additional command line option to request the playfair or caesar cipher (use –help to check!). To actually create one though, we need to create a Playfair Cipher class just like the CaesarCipher

- Go through the following steps

  1. Create a very basic 'PlayfairCipher' class skeleton that:

     ➔ Holds a std::string key that is assigned with a setKey function

     ➔ Has a constructor that takes a std::string key and calls the setKey function

     ➔ Has an applyCipher function that just prints a message at the moment

  2. Create this class with the given key if specified on the cmd line. Note that the key for the Playfair cipher is a string not an int!

  3. Check that you can call the applyCipher function correctly

- Below, you can see the function definitions to be used:

```
PlayfairCipher::PlayfairCipher(const std::string& key) {...}

void PlayfairCipher::setKey(const std::string& key) {…}

std::string PlayfairCipher::applyCipher(const std::string& inputText, const CipherMode cipherMode ) const
```

Exercise 2 – Implementation Steps

- Though the cipher is significantly more complicated than the Caesar Cipher, we can break down everything in to several easier steps

- Copy the following comments into your 'setKey' and 'applyCipher' functions as placeholders for the actual code:

```cpp
void PlayfairCipher::setKey( \
    const std::string& key)
{
    // store the original key
    key_ = key;

    // Append the alphabet

    // Make sure the key is upper case

    // Remove non-alpha characters

    // Change J -> I

    // Remove duplicated letters

    // Store the coords of each letter

    // Store the playfair cihper key map

}
```

```cpp
std::string PlayfairCipher::applyCipher( \
    const std::string& inputText, \
    const CipherMode cipherMode ) const
{
    // Change J → I

    // If repeated chars in a digraph add an X or Q if XX

    // if the size of input is odd, add a trailing Z

    // Loop over the input in Digraphs

    //   - Find the coords in the grid for each digraph

    //   - Apply the rules to these coords to get 'new' coords

    //   - Find the letter associated with the new coords

    // return the text
    return input;
}
```

Iterators, Algorithms, Lambdas, Maps

Mark Slater

UNIVERSITY OF BIRMINGHAM

1. Iterators

Iterators

- There are several different STL containers apart from string and vector

- However, some containers can't be accessed by an incremental index variable which means you can't just have an index number to loop over the elements (e.g. map) so you need a more general method

- Iterators give a powerful and more generic mechanism for accessing containers. They:

  → Point to an element of a container

  → Know how to move from one element to the next

  → Can be 'dereferenced' to access the element it points to

- Each STL container class provides at least one iterator type as well as special functions that return iterators for the first and last elements in a container

Iterators Example

```cpp
#include <vector>

int main()
{
    // initialise a vector
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // create an iterator
    std::vector<int>::iterator iter1{ vec.begin() };

    // Use iterators to loop – range based loops use this behind the scenes
    for (auto iter2 { std::begin(vec) }; iter2 != vec.end(); ++iter2)
    {
        // dereference to get the element
        std::cout << *iter2 << std::endl;
    }

    // Can also add/subtract from iterators
    auto iter3 { std::begin(vec) + 1 };

}
```

This is an iterator type specifically for the integer vector class – you can't have interchangeable iterators

The 'begin()' method of std::vector returns an iterator that points to the beginning of the container

The 'end()' method of std::vector returns an iterator that points to ONE PAST the last element of the container

Can also use 'auto' which will become a lot more useful when dealing with these long named types!

Use the '*' operator to dereference the iterator and obtain the element it points to

11

Const Iterators

- As we have said, it's always best to keep variables const unless you definitely need to change it

- Iterators are a bit different because you usually need to change the iterator (inc/decrement it) but you will want to keep the thing it points to constant. This is where const_iterators are used.
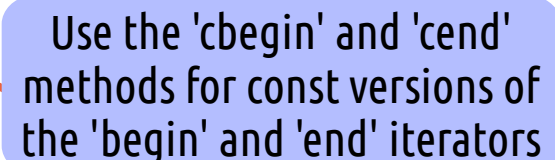
```cpp
#include <vector>

int main()
{
    // create a vector
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // create a const_iterator to point to it
    std::vector<int>::const_iterator iter;
    iter = vec.cbegin();

    // This is OK
    iter++;

    // This isn't
    *iter = 10;
}
```

Use the 'cbegin' and 'cend' methods for const versions of the 'begin' and 'end' iterators

Iterators of Other Objects

- You can create iterators from any objects that satisfy the requirements of an iterator and can then be used in the algorithms, etc. we'll see later

- This first of these we'll briefly touch on is a ostream iterator

- This can output things to the given stream (with optional delimiter) by assignment:

```cpp
#include <iostream>
#include <vector>
#include <iterator>

int main()
{
    // create an iterator based on std::cout
    std::ostream_iterator<int> cout_iter{std::cout, "\n"};

    // output something
    cout_iter = 5;
}
```

Note that you have to say what type you're streaming to and from

Assign to the iterator to actually perform the input/ouput

13

Iterators of Other Objects

- The second iterator we'll look at specifically is an insertion iterator – back_insert_iterator

- This will add elements (equivalent to doing push_back in this case) to the given vector on assignment

- There are many others so do look at the documentation!

```cpp
#include <iostream>
#include <vector>
#include <iterator>

int main()
{
    // create an iterator to insert elements
    std::vector<int> vec;
    std::back_insert_iterator< std::vector<int> > iter1{vec};
    auto iter2 { std::back_inserter( vec ) };

    // add an element
    iter2 = 5;
}
```

Create an iterator that will push_back to the given vector

Assign to the iterator to add the element

14

2. Algorithms

Algorithms

- At the moment, Iterators probably seem rather over-engineered for just looping over elements of a container

- Where they really show their power is when used in Algorithms

- These are generic programming tasks that use iterators to operate on containers.

- There are many algorithms available, some of which are:

  - → copy, copy_if: Copy elements from one range to another
  - → find, find_if, find_if_not: Find an element in a range
  - → generate: Save the result of a function into a range
  - → max_element: returns the max element in the range

- For a full list, see:

  http://en.cppreference.com/w/cpp/algorithm

Algorithms – reverse_copy Example

- In this example, we use an algorithm to fill a vector with the reverse of another vector, i.e. the first element becomes the last, etc.

- This would normally involve a 'for' loop with some non-trivial logic within it but with the algorithm is reduced to one line!

```cpp
#include <vector>
#include <algorithm>

int main()
{
    // create a vector
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // create one to take the reverse
    // Note: need to create and then resize!
    std::vector<int> rev;
    rev.resize( vec.size() );

    // fill it – rev will now contain 5, 4, 3, 2, 1
    std::reverse_copy( vec.begin(), vec.end(), rev.begin() );

    // or use a back_inserter instead
    std::vector<int> rev2;
    std::reverse_copy( vec.begin(), vec.end(), std::back_inserter( rev2 ) );
}
```

The reverse vector must be the same size as the range use to fill it

Specify the ranges to be used using iterators – begin() and end() in this case

Can also use a back_inserter to fill an empty vector

17

Algorithms – sort and copy Example

- For this example, we'll use the 'sort' algorithm to sort a vector in place

- We will them use 'copy' and a std::cout iterator to output the result

```cpp
#include <vector>
#include <algorithm>
#include <iterator>

int main()
{
    // create a vector
    std::vector<int> vec = {1, 20, 3, 40, 5, 50};

    // sort the vector in place
    std::sort( vec.begin(), vec.end() );

    // create an iterator for std::cout
    std::ostream_iterator<int> cout_iter{ std::cout, "\n" };

    // output values
    std::copy( vec.begin(), vec.end(), cout_iter );
}
```

'copy' using the std::cout iterator just outputs each element copied

Algorithms – Why use them?

- You may be thinking 'Why should I use algorithms? – I can just do my own loops'

- Scott Meyers 'Effective STL' book gives three reasons for preferring algorithms over hand written loops:

  - ➔ Efficiency: Quite probably more efficient (Not guaranteed but likely!)
  - ➔ Correctness: Less code written means fewer places for bugs
  - ➔ Maintainability: Code is often clearer and more straightforward

- Scientific software can shy away from algorithms because of efficiency concerns (or lack of knowledge about the efficiency)

- However, you should generally prefer clear and simple code until a performance problem is found - don't prematurely optimise!

- There are times when algorithms are less efficient, but you should be sure about this through testing and profiling before changing the code!

Exercise 3 – Playfair Cipher Implementation

- So we can now start doing the implementation of the Playfair Cipher using Algorithms. We'll start with the setKey function:

```cpp
void PlayfairCipher::setKey(const std::string& key)
{
    // store the original key
    key_ = key;

    // Append the alphabet

    // Make sure the key is upper case

    // Remove non-alpha characters

    // Change J -> I

    // Remove duplicated letters

    // Store the coords of each letter

    // Store the playfair cihper key map
}
```

Storing the key and appending the alphabet can be done without algorithms

Use the std::transform algorithm with the ::toupper (NOT std::toupper!) function to change to upper case

3. Lambdas

What are Lambdas?

- Lambdas can basically be thought of as 'inline' function definitions then can then be passed around just like any other variable

- In other words, they allow you to define a function within a code block just as you would any other object and pass it to a function or assign to a variable

- This can become very useful for extending algorithms or a providing a way for the calling code to specify the precise implementation of a given programming concept without the overhead of a 'formal' function definition

- For example, you may have an address book class that provides a generic search algorithm but the specifics of how you search can be decided by the calling code, e.g. by first or last name, address, etc.

- You can just give a general search function that takes a lambda (i.e. function definition) and calls this function when performing the search

- This can then be provided either by a usual function definition or as a lambda

- Lambdas have one major advantage over traditional functions as well – they can access variables that were defined in the scope of the calling code

Generalising Algorithms

- So how does that help us with our algorithm use?

- At present, the algorithms we've seen are good for their specific purpose but probably seem a bit limited

- You are either restricted to using already available functions or writing a standalone function away from where it is needed just for a one line use

- This is where Lambdas can be used: Instead of having to define a function well away from the calling scope that is probably only relevant for that scope, it can be put 'inline'

- Plus, as just mentioned, you can provide different functions that can manipulate locally defined variables

Lambda Syntax and Declaration

- Lambdas are defined using the '[]' syntax followed by a normal function definition

```cpp
int main()
{
    // use the generate algorithm with a lambda to
    // fill a vector with 7s
    std::vector<int> vec{};
    vec.resize(10);
    std::generate( vec.begin(), vec.end(), [] () { return 7; } );

    // create a vector
    std::vector<int> vec = {1, 20, 3, 40, 5, 50};

    // create an iterator for std::cout
    std::ostream_iterator<int> cout_iter{ std::cout, "\n" };

    // output values greater than 10
    auto func = [] (int val) {
        if (val > 9)
            return true;
        else
            return false;
    };
    std::copy_if( vec.begin(), vec.end(), cout_iter, func);
}
```

Create a lambda function that returns the number to store and pass this as the function that 'generate' calls

As the return value can be determined by the compiler in this case, you don't have to give it explicitly

Create and store a lambda function that checks if a the given value is greater than 9

copy_if requires a function that takes the same argument type as that held by the container

Exercise 4 – Playfair Cipher Implementation

- Now we can use lambdas, we'll return to the Playfair Cipher and do the next bit of implementation!

```cpp
void PlayfairCipher::setKey(const std::string& key)
{
    // store the original key
    key_ = key;

    // Append the alphabet

    // Make sure the key is upper case

    // Remove non-alpha characters

    // Change J -> I

    // Remove duplicated letters

    // Store the coords of each letter

    // Store the playfair cihper key map

}
```

Use the std::remove_if algorithm with a lambda that simply returns the opposite of isalpha. NOTE: this doesn't actually remove anything! It reorders the container with the objects to be kept at the beginning. It the returns an iterator that can be used with 'erase'

```cpp
std::string str1 = "Text with some   spaces";
// reorder string and return iter to start of chars to erase
auto iter = std::remove(str1.begin(), str1.end(), ' ')
// actually erase
str1.erase(iter,str1.end());
```

You can now use a lambda along with the 'transform' algorithm to perform this

Variable Capture

- What makes lambdas even more powerful is the idea of variable capture

- This means that you can use local variables in the lambda function, something you couldn't do with a normal function declared outside the scope

- To do this, you simply add an option between the brackets to indicate what capture you want to do:

  - → [] - Don't capture anything
  - → [&] - Capture any referenced variable by reference
  - → [=] - Capture any referenced variable by value (i.e. make a copy)
  - → [foo, &bar] – Capture 'foo' by value and 'bar' by reference
  - → [this] – Capture the 'this' pointer of the enclosing class

- This gives you a lot of power for using locally declared variables in algorithms and outside the calling code

- Be careful with capturing by reference and storing the lambda – the captured variables would be destroyed on leaving the scope and any further calls to the lambda would fail

Variable Capture Example

```cpp
#include <vector>

int main()
{
    // create a vector
    std::vector<int> vec = {1, 2, 3, 4, 5};
    int i{0}, j{10};

    // capture by reference - you get 7, 14, 21...
    std::generate( vec.begin(), vec.end(), [&] () { i+=7; return i; } );

    // capture by value and reference
    std::generate( vec.begin(), vec.end(), [i,&j] () { j+=7; return i*j; } );

    // capture by value - this will fail as 'i' is read-only
    std::generate( vec.begin(), vec.end(), [=] () { i+=7; return i; } );
}
```

Exercise 5 – Playfair Cipher Implementation

- We can now continue on to the next part of the Playfair Cipher

```cpp
void PlayfairCipher::setKey(const std::string& key)
{
    // store the original key
    key_ = key;

    // Append the alphabet

    // Make sure the key is upper case

    // Remove non-alpha characters

    // Change J -> I

    // Remove duplicated letters

    // Store the coords of each letter

    // Store the playfair cihper key map

}
```
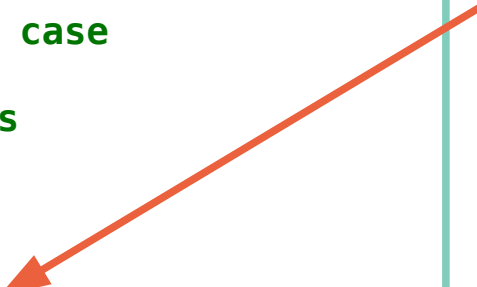
This will be another use of string.erase and remove_if as you did with the non-alpha characters.

However, this time you'll need a lambda function that checks against a stored string containing all the encountered letters so far (string.find is useful here). This is where variable capture is needed – declare the encountered letters string BEFORE the function and then use it in the lambda function so the same encountered characters are added to the same string each iteration

4. Maps and Pairs

Maps and Pairs

- Up until now, we've only dealt with sequence containers like std::string and std::vector but there are also Associative Containers like std::map

- Each value stored is also associated with a key which allows fast retrieval of elements based on that key

- These key-value combinations in maps are grouped together using the std::pair type from which you can access the 'first' or 'second' elements of the pair

- These 'pairs' can also be useful in other situations, not just with std::maps

- As with vectors, elements can be added and iterators used to cycle through the them though in this case, the iterators point to std::pairs

- They work in a very similar way to dictionaries in python

- Note that when dealing with maps, it can become very useful to use 'using .. = ..' or typedefs – this will create new 'labels' for types to save typing

Map and Pair Example

```cpp
#include <map>
#include <iostream>

int main()
{
    // Create a new label for the type – could also use 'typedef'
    using Str2IntMap = std::map<std::string, int>;

    // create an instance of this map
    Str2IntMap mymap;

    // create a pair and insert it using either pair or value_type
    std::pair< std::string, int > p0{ "A", 1 };
    auto p1 { std::make_pair( "B", 2 ) };
    Str2IntMap::value_type p2{ "C", 3 };
    mymap.insert( p0 );
    mymap.insert( p1 );
    mymap.insert( p2 );

    // Use the subscript notation instead
    mymap["C"] = 3;

    // Find elements in the map
    auto iter = mymap.find("A");
    std::cout << (*iter).first << ": " << (*iter).second << std::endl;

    // Use range based for loop to print the map
    for ( auto p : mymap )
    {
        std::cout << p.first << ": " << p.second << std::endl;
    }
}
```

Use auto to avoid long type names

'find' returns an iterator that points to the appropriate std::pair – note you should (almost) always check it's not equal to the end of the container!

31

Exercise 6 – Playfair Cipher Implementation

- We can now tackle the last part of the setKey function in the Playfair Cipher

```cpp
void PlayfairCipher::setKey(const std::string& key)
{
    // store the original key
    key_ = key;

    // Append the alphabet

    // Make sure the key is upper case

    // Remove non-alpha characters

    // Change J -> I

    // Remove duplicated letters

    // Store the coords of each letter

    // Store the playfair cihper key map

}
```

Loop over each letter, calculate the row and column numbers and then store both the letter and a std::pair of the coordinates in a map. You will need two maps stored as members of the class – one to go from letter → coord and another to go from coord → letter

Exercise 7 – Playfair Cipher Implementation

- You can now complete the Playfair Cipher by implementing the encrypt function

- You are free to do this how you wish but try to use what you've learned today!

- There are some hints below:

```cpp
std::string PlayfairCipher::applyCipher( \
    const std::string& inputText, \
    const CipherMode cipherMode ) const
{
    // Change J → I

    // If repeated chars in a digraph add an X or Q if XX

    // if the size of input is odd, add a trailing Z

    // Loop over the input in Digraphs

    //   – Find the coords in the grid for each digraph

    //   – Apply the rules to these coords to get 'new' coords

    //   – Find the letter associated with the new coords

    // return the text
    return input;
}
```

Can't really be done with algorithms – Use a loop, check if current char is the same as previous char. If so, store X+current char

Can be done by using += 2 on the iterator/loop counter rather than just ++

Use 'find' on the appropriate map to get the coords. Then from that calculate the new position coords using the rules and use the other map to get back to the en/decrypted letter