Intermediate Programming

Day 26

Outline

- Exercise 25
- References
- Dynamic Memory Allocation
- Review questions

Exercise 25

Complete the abbreviate function.

abbrev.cpp

```
...
string abbreviate( string word )
{
	string result;
	bool last_was_vowel = false;

	for( size_t i=0 ; i<word.size() ; i++ )
	{
		bool cur_is_vowel = is_vowel( word[i] );

		if( !cur_is_vowel ) result.push_back( word[i] );
		else if( !last_was_vowel ) result.push_back( '\'' );

		last_was_vowel = cur_is_vowel;
	}

	return result;
}
...
```

Exercise 25

Invoke the abbreviate function.

abbrev.cpp

```
...
int main( int argc , char **argv )
{
    if( argc!=3 )
    {
        cerr << "Usage: abbrev <infile> <outfile>" << endl;
        return 1;
    }

    ifstream in( argv[1] );
    ofstream out( argv[2] );
    string line;
    while( getline( in , line ) )
    {
        stringstream ss(line);
        string word;
        while( ss >> word ) out << abbreviate( word ) << " ";
        out << endl;
    }

    return 0;
}
```

Exercise 25

Determine and count the token types.

classify.cpp

```
…
int main( void )
{
        …
        while( cin >> token )
        {
                stringstream sstream( token );
                double fp ; int i ; string s;

                if( ( sstream >> i ) && !( sstream >> s ) ) sum_i += i;
                else
                {
                        sstream = stringstream( token );
                        if( sstream >> fp ) sum_fp += fp;
                        else
                        {
                                sstream = stringstream( token );
                                if( sstream >> s ) ntok++ , ntok_c += s.length();
                        }
                }
        }
        …
}
```

Exercise 25

Count the frequencies of the different letters.

letter_freq.cpp

```cpp
...
struct Bucket{ char letter ; unsigned count; };
int main( int argc , char **argv )
{
        if( argc!=2 )
        {
                cerr << "Usage: abbrev <infile>" << endl;
                return 1;
        }
        ifstream in( argv[1] );
        char c;
        vector< Bucket > hist;
        hist.resize( 26 );
        for( unsigned int i=0 ; i<26 ; i++ )
        {
                hist[i].count = 0;
                hist[i].letter = 'a'+i;
        }

        while( in.get(c) )
        {
                if( c>='a' && c<='z' ) hist[c-'a'].count++;
                else if( c>='A' && c<='Z' ) hist[c-'A'].count++;
        }
        ...
}
```

Exercise 25

Sort and print the frequencies.

letter_freq.cpp

```
...
struct Bucket{ char letter ; unsigned count; };

bool compare_buckets( const Bucket &left , const Bucket &right )
{
	return left.count>right.count;
}

int main(int argc, char **argv)
{
	...
	vector< Bucket > hist;
	...
	sort( hist.begin() , hist.end() , compare_buckets );

	for( unsigned int i=0 ; i<hist.size() && hist[i].count ; i++ )
		cout << hist[i].letter << ": " << hist[i].count << endl;
	...
}
```

Outline

- Exercise 25
- References
- Dynamic Memory Allocation
- Review Questions

References

- In C, we could pass arguments by *value* or by *address*
  - By value
    - creates a copy of the contents
  - By address
    - allows us to modify the callee’s variables
    - requires dereferencing to access

main.c

```
#include <stdio.h>
void swap( int *a , int *b )
{
	int tmp = *a;
	*a = *b;
	*b = tmp;
}
int main( void )
{
	int i1 = 1 , i2 = 2;
	swap( &i1 , &i2 );
	printf( "%d %d\n" , i1 , i2 );
	return 0;
}
```

```
>>./a.out
2 1
>>
```

References

- C++ also allows us to pass arguments by *reference*
  - Look like values
    - no need to dereference
  - Act like pointers
    - function sees the argument, not a copy
  - Unlike pointers:
    - A reference can't be NULL
    - A reference must be initialized when it's declared
    - A reference cannot be reassigned

main.cpp

```
#include <iostream>
void swap( int &a , int &b )
{
	int tmp = a;
	a = b;
	b = tmp;
}
int main( void )
{
	int i1 = 1 , i2 = 2;
	swap( i1 , i2 );
	std::cout << i1 << " " << i2 << std::endl;
	return 0;
}
```

```
>>./a.out
2 1
>>
```

References

- C++ also allows us to pass arguments by *reference*
  - References are declared using a "&" after the type / class they refer to

main.cpp

```
#include <iostream>
int main( void )
{
        int i = 1 , j =10;
        int &r = i;
        r = j;
        std::cout << i << std::endl;
        return 0;
}
```

References

- C++ also allows us to pass arguments by *reference*
  - References are declared using a "&" after the type / class they refer to
  - Must be defined as soon as they are declared

main.cpp

```
#include <iostream>
int main( void )
{
        int i = 1 , j =10;
        int &r = i;
        r = j;
        std::cout << i << std::endl;
        return 0;
}
```

References

- C++ also allows us to pass arguments by *reference*
  - References are declared using a "&" after the type / class they refer to
  - Must be defined as soon as they are declared
  - Note that the line "r=j";
    - Does not make r a reference to j
    - It copies the contents of j into what r refers to

main.cpp

```
#include <iostream>
int main( void )
{
        int i = 1 , j =10;
        int &r = i;
        r = j;
        std::cout << i << std::endl;
        return 0;
}
```

```
>>./a.out
10
>>
```

References

- C++ also allows us to pass arguments by *reference*
  - We saw this before:
    - The getline function takes a **reference** to a string
    - This allows the method to set line with the value of the next line of text read in from the stream

main.cpp

```
#include <iostream>
#include <cctype>
#include <string>
int main( void )
{
	std::string line;
	while( std::getline( std::cin , line ) )
		std::cout << line << std::endl;
	return 0;
}
```

```
>> echo "the quick brown fox" | ./a.out
the quick brown fox
>>
```

References

- C++ also allows us to pass arguments by *reference*
  - As with pointers, this allows a function to affect multiple output values

main.cpp

```
#include <iostream>
void Set2And3( int &a , int &b )
{
	a=2 , b=3;
}
int main( void )
{
	int i1 , i2;
	Set2And3( i1 , i2 );
	std::cout << i1 << " " << i2 << std::endl;
	return 0;
}
```

```
>> ./a.out
2 3
>>
```

References

- C++ also allows us to pass arguments by *reference*
  - We can also return a reference

main.cpp

```
#include <iostream>
using namespace std;
int &minref( int &a , int &b )
{
	if( a<b ) return a;
	else      return b;
}
int main( void )
{
	int a = 5, b = 10;
	int& min = minref( a , b );
	min = 12;
	cout << "a=" << a << ", b=" << b << ", min=" << min << endl;
}
```

```
>> ./a.out
a=12, b=10, min=12
>>
```

References

- C++ also allows us to pass arguments by *reference*
  - We can also return a reference
    - The object receiving the reference must be declared on the same line as the function call

main.cpp

```
#include <iostream>
using namespace std;
int &minref( int &a , int &b )
{
	if( a<b ) return a;
	else      return b;
}
int main( void )
{
	int a = 5, b = 10;
	int &min = minref( a , b );
	min = 12;
	cout << "a=" << a << ", b=" << b << ", min=" << min << endl;
}
```

```
>> ./a.out
a=12, b=10, min=12
>>
```

References

- C++ also allows us to pass arguments by *reference*
  - We can also return a reference
    - The object receiving the reference must be declared on the same line as the function call
    - The function's arguments have to be references themselves!
      - Otherwise we would be returning a reference to minref's stack variable that was no longer in existence

main.cpp

```
#include <iostream>
using namespace std;
int &minref( int a , int b )
{
	if( a<b ) return a;
	else      return b;
}
int main( void )
{
	int a = 5, b = 10;
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In function int& minref(int, int) :
main.cpp:3:18: warning: reference to local variable a returned [-Wreturn-local-addr]
 int& minref( int a , int b )
                  ^
>>
```

References

- C++ also allows us to pass arguments by *reference*
  - If a reference is declared `const`, its value cannot be changed
    - We could have protected the value using pass-by-value
      but that would duplicate the contents of the object

References

- C

main.cpp

```
#include <iostream>
#include <map>
#include <string>

void print( const std::map< int , std::string >& map )
{
        for( std::map< int , std::string >::const_iterator it=map.cbegin() ; it!=map.cend() ; ++it )
                std::cout << it->first << “: “ << it->second << std::endl;
}
int main( void )
{
        int id;
        std::string name;
        std::map< int , std::string > id2name;
        while( std::cin >> id >> name ){ id2name[ id ] = name; }
        print( id2name );
        return 0;
}
```

Outline

- Exercise 25
- References
- Dynamic Memory Allocation
- Review Questions

Dynamic Memory Allocation

- In C, we allocate memory on the heap using malloc:

```
void * malloc( size_t size );
```

  - This function does not need to know the type of the data
  - Just the size of the memory we were requesting

- Similarly, we deallocate memory from the heap using free:

```
void free( void * ptr );
```

  - This function does not need to know the type of the data
  - Just the location that we are freeing

Dynamic Memory Allocation

- In C++, we need to know the data-type to invoke the constructor*
- We do this using the **new** operator:

  <DataType> * new DataType(<ConstructorParams>);

  - This allocates memory for a single object and invokes the constructor
    - Though primitive types (e.g. ints, chars, etc.) don't have constructors, we can use new to allocate them on the heap (they will not be initialized)

main.cpp

```
#include <iostream>
#include <string>
using std::string;
int main( void )
{
	string * strPtr = new string( "Hello" );
	std::cout << *strPtr << std::endl;
	...
	return 0;
}
```

Note that **new** is not a function, i.e. it does not take arguments in parentheses.

Dynamic Memory Allocation

- In C++, we need to know the data-type to invoke the destructor*
- We do this using the delete operator:

delete <DataType>*;

  - This invokes the destructor of the object
    - Though primitive types (e.g. ints, chars, etc.) don't have destructors, we can use delete to deallocate them from the heap
  - And deallocates its memory

Note that delete is not a function, i.e. it does not take arguments in parentheses.

main.cpp

```
#include <iostream>
#include <string>
using std::string;
int main( void )
{
    string * strPtr = new string( "Hello" );
    std::cout << *strPtr << std::endl;
    delete strPtr;
    return 0;
}
```

Dynamic Memory Allocation

- We allocate **arrays** of objects using new[]:

  <DataType> * new DataType[<NumElems>];

  - This allocates memory for a NumElems objects
  - And invokes the **default** constructor* for each one

- And we deallocate using delete[]:

  delete[] <DataType>*;

  - This invokes the destructor* for each object
  - And then deallocates the memory for the entire array of objects

Dynamic Memory All

main.cpp

```
#include <iostream>
#include "rectangle.h"

int main( void )
{
    int count;
    std::cout << "Rectangle Count: ";
    std::cin >> count;
    Rectangle** r = new Rectangle*[count];
    for( int i=0 ; i<count ; i++ )
    {
        int w , h;
        std::cout << "Width: " ; std::cin >> w;
        std::cout << "Height: " ; std::cin >> h;
        r[i] = new Rectangle();
        r[i]->width = w;
        r[i]->height = h;
    }
    for( int i=0 ; i<count ; i++ ) r[i]->print();
    for( int i=0 ; i<count ; i++ ) delete r[i];
    delete[] r;
    return 0;
};
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
    double w , h;
    Rectangle( void );
    void print( void ) const;
    double area( void ) const ;
};
#endif // RECTANGLE_INCLUDED
```

Dynamic Memory All

- Memory allocated with **new** must be deallocated with **delete**

main.cpp

```
#include <iostream>
#include “rectangle.h"

int main( void )
{
    int count;
    std::cout << "Rectangle Count: ";
    std::cin >> count;
    Rectangle** r = new Rectangle*[count];
    for( int i=0 ; i<count ; i++ )
    {
        int w , h;
        std::cout << "Width: " ; std::cin >> w;
        std::cout << "Height: " ; std::cin >> h;
        r[i] = new Rectangle();
        r[i]->width = w;
        r[i]->height = h;
    }
    for( int i=0 ; i<count ; i++ ) r[i]->print();
    for( int i=0 ; i<count ; i++ ) delete r[i];
    delete[] r;
    return 0;
};
```

Dynamic Memory All

- Memory allocated with **new** must be deallocated with **delete**
- Memory allocated with **new[]** must be deallocated with **delete[]**
  - We are **delete**ing an array of **pointers** to **Rectangle** objects

  ⇒ The **Rectangle** destructor is not called

main.cpp

```
#include <iostream>
#include "rectangle.h"

int main( void )
{
    int count;
    std::cout << "Rectangle Count: ";
    std::cin >> count;
    Rectangle** r = new Rectangle*[count];
    for( int i=0 ; i<count ; i++ )
    {
        int w , h;
        std::cout << "Width: " ; std::cin >> w;
        std::cout << "Height: " ; std::cin >> h;
        r[i] = new Rectangle();
        r[i]->width = w;
        r[i]->height = h;
    }
    for( int i=0 ; i<count ; i++ ) r[i]->print();
    for( int i=0 ; i<count ; i++ ) delete r[i];
    delete[] r;
    return 0;
};
```

Dynamic Memory All

- Memory allocated with **new** must be deallocated with **delete**

- Memory allocated with **new[]** must be deallocated with **delete[]**

main.cpp

```
#include <iostream>
#include “rectangle.h"

int main( void )
{
        int count;
        std::cout << "Rectangle Count: ";
        std::cin >> count;
        Rectangle** r = new Rectangle*[count];
        for( int i=0 ; i<count ; i++ )
        {
                int w , h;
                std::cout << "Width: " ; std::cin >> w;
                std::cout << "Height: " ; std::cin >> h;
                r[i] = new Rectangle();
                r[i]->width = w;
                r[i]->height = h;
        }
        for( int i=0 ; i<count ; i++ ) r[i]->print();
        for( int i=0 ; i<count ; i++ ) delete r[i];
}
```

Note that since **r[i]** is a pointer to a **Rectangle**, we access its members using the **->** operator

Dynamic Memory All

We could also do this without memory allocation/deallocation using STL vectors

main.cpp

```
#include <iostream>
#include “rectangle.h"

int main( void )
{
	int count;
	std::cout << "Rectangle Count: ";
	std::cin >> count;
	std::vector< Rectangle > r;
	for( int i=0 ; i<count ; i++ )
	{
		int w , h;
		std::cout << "Width: " ; std::cin >> w;
		std::cout << "Height: " ; std::cin >> h;
		r.push_back( Rectangle() );
		r[i].width = w;
		r[i].height = h;
	}
	for( int i=0 ; i<count ; i++ ) r[i].print();
	return 0;
};
```

Outline

- Exercise 25
- References
- Dynamic Memory Allocation
- Review questions

Review questions

1. What is a C++ *reference*?

An alias for an existing variable

Review questions

2. When should you use C++ references?

1. To allow a function to affect multiple outputs (w/o pointers).
2. To pass data to a function without incurring the cost of a copy.

Review questions

3. What is the difference between a pointer and a reference?

Can't be NULL, must be initialized immediately, can't be changed

Review questions

4. How do you dynamically allocate memory in C++?

`new` or `new[]`

Review questions

5. How do you free memory in C++?

`delete` or `delete[]`

Exercise 26

- Website -> Course Materials -> Exercise 26