Contents

1.1Normal Random Variable Generator **1-1**

1.1.1 Box-Muller 1-1

1.1.2 Summation of Uniform 1-1

1.1.3 Key points 1-1

2.1Header File **2-1**

2.1.1 Key points 2-1

3.1Vector Accumulation **3-1**

3.1.1 Key Points 3-1

4.1Big Number Problem **4-1**

4.1.1 Key Points 4-1

5.1Number to String **5-1**

5.1.1 Key Points 5-1

5.2Integer to Number **5-1**

5.2.1 Header $< sstream >$ 5-1

5.2.2 C library 5-1

5.2.3 Key Point 5-1

6.1Pre-fix, Post-fix and Type conversion **6-1**

6.1.0.1 Pre-fix 6-1

6.1.0.2 Post-fix 6-1

6.1.0.3 Type conversion 6-1

6.1.0.4 Key Points 6-1

7.1Virtual Destructor **7-1**

7.1.0.5 Example 7-1

7.2Abstract Class **7-1**

7.2.0.6 Example 7-1

8.1Template Function **8-1**

8.1.0.7 Examples 8-1

8.2 Template Class **8-1**

8.2.0.8 Examples 8-1

8.2.1 Inheritance 8-1

8.2.1.1 Examples 8-1

8.3 Key Points **8-1**

9.1 Factorial **9-1**

9.2 Key Points **9-1**

10.1 Associative Container **10-1**

10.1.1 Examples 10-1

10.2 Algorithm **10-1**

10.2.1 Compatibility 10-2

11.1 Tutorial 5 **11-1**

11.1.1 Polynomials 11-1

11.2 Tutorial 6 **11-1**

11.2.1 The usage of ENUM 11-1

11.3 Tutorial 7 **11-2**

11.3.1 Overriding and Overloading 11-2

11.4 Interesting Question **11-2**

C++ Design Patterns and Derivatives Pricing **Cambridge 2011**

Chapter 1: A simple Monte Carlo model

Author: Mark S.Joshi *Scribe: Treeman*

1.1 Normal Random Variable Generator

1.1.1 Box-Muller

Let U_1 and U_2 be two independent random variables with distribution $U(0,1)$–Basic Form and Polar Form:

$$\begin{aligned} Z_1 &= \sqrt{-2\log(U_1)}\cos(2\pi U_2) = \sqrt{-2\log(s)\left(\frac{u}{\sqrt{s}}\right)} \\ Z_2 &= \sqrt{-2\log(U_1)}\sin(2\pi U_2) = \sqrt{-2\log(s)\left(\frac{v}{\sqrt{s}}\right)} \end{aligned} \tag{1.1}$$

1.1.2 Summation of Uniform

Note that the sum of twelve i.i.d uniform random variables subtracted by 6 can be a good approximation of standard normal:

$$Y = \sum_{i=1}^{12} X_i - 6 \sim \text{N(0,1)} \qquad \text{where } X_i \sim \text{U(0,1)}$$

1.1.3 Key points

- Monte Carlo uses the Law of Large Numbers to approximate this risk-neutral expectation
- Procedural programs can be hard to extend and reuse
- Classes allow us to encapsulate concepts which makes reuse and extensibility a lot easier
- Making classes closely model real-world concepts makes them easier to design and to explain
- Classes alow us to separate the design of the interface from the coding of the implementation
- Reuse is as much a social issue as a technical one

C++ Design Patterns and Derivatives Pricing **Cambridge 2011**

Chapter 2: Encapsulation

Author: Mark S.Joshi *Scribe: Treeman*

2.1 Header File

```
#ifndef PAYOFF_H
#define PAYOFF_H

class PayOff
{
public:
    enum OptionType {call, put};
    PayOff(double Strike_,OptionType TheOptionsType_);
    double operator()(double Spot) const;
private:
    double Strike;
    OptionType TheOptionsType;
}
#endif
```

2.1.1 Key points

- Using a pay-off class allows us to add extra forms of pay-offs without modifying our Monte Carlo routine
- By overloading operator() we can make an object look like a function
- const enforces extra discipline by forcing the coder to be aware of which code is allowed to change things and which code cannot
- The open-closed principle says that code should be open for extension but closed for modification
- Private data helps us to separate interface from implementation

C++ Design Patterns and Derivatives Pricing **Cambridge 2011**

Chapter 3: Inheritance and virtual function

Author: Mark S.Joshi *Scribe: Treeman*

3.1 Vector Accumulation

```
 #include <iostream>
using std::cout;
using std::endl;

#include <algorithm>
#include <numeric>
#include <vector>
#include <iterator>

int main()
{
   std::ostream_iterator< int > output( cout, " " );

   int a2[ 10 ] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
   std::vector< int > v2( a2, a2 + 10 ); // copy of a2
   cout << "\n\nVector v2 contains: ";
   std::copy( v2.begin(), v2.end(), output );

   // calculate sum of elements in v
   cout << "\n\nThe total of the elements in Vector v is: "
      << std::accumulate( v2.begin(), v2.end(), 0 );

   cout << endl;
   // calculate product of elements in v
   cout<< " The product of all elements is ";
   cout<<accumulate(v.begin(),v.end(),1,multiplies<int>())<<endl;
   return 0;
}
```

Output:

```
 Vector v2 contains: 100 2 8 1 50 3 8 8 9 10

The total of the elements in Vector v is: 199
```

3.1.1 Key Points

Big Numbers **Website**

Chapter 4: Deal with Big Factorial

Author: Robert *Scribe: Treeman*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the author.*

4.1 Big Number Problem

```
//calculate nd = the number of digits required
 p = 0.0;
 // p is really log10(n!)
 for(j = 2; j <= n; j++)
 {
   p += log10((double)j);   // cast to double
 }

 nd = (int)p + 1;
 // allocate memory for the char array
 ca = new unsigned char[nd];
 if (!ca)
 {
   cout << "Could not allocate memory!!!";
   return -1;
 }
//initialize char array
for (i = 1; i < nd; i++)
{
  ca[i] = 0;
}
ca[0] = 1;

// put the result into a numeric string using the array of characters
p = 0.0;
for (j = 2; j <= n; j++)
{
  p += log10((double)j);   // cast to double!!!
  nz = (int)p + 1;         // number of digits to put into ca[]
  q = 0;                   // initialize remainder to be 0
  for (i = 0; i <= nz; i++)
  {
    temp = (ca[i] * j) + q;
    q = (temp / 10);
    ca[i] = (char)(temp % 10);
  }
}
```

4.1.1 Key Points

- Calculate the number of digits for each big number
- convert the input string number to the array of number

String and Number **Website**

Chapter 5: Deal with Conversion

Author: Robert *Scribe: Treeman*

5.1 Number to String

Include header $< string >$ and then use to string in the std namespace.

5.1.1 Key Points

5.2 Integer to Number

5.2.1 Header $< sstream >$

```
string Text = "456";
int Number;
if ( ! (istringstream(Text) >> Number) ) Number = 0;
```

5.2.2 C library

The stdlib header contains some functions to convert text and numbers. Notice that some of these functions are not standard!(”atoi”)

```
/* isdigit example */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main ()
{
  char str[]="1776ad";
  int year;
  if (isdigit(str[0]))
  {
    year = atoi (str);
    printf ("The year that followed %d was %d.\n",year,year+1);
  }
  return 0;
}
```

5.2.3 Key Point

++ and – **Website**

Chapter 6: Deal with Overloading

Author: Robert *Scribe: Treeman*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the author.*

6.1 Pre-fix, Post-fix and Type conversion

6.1.0.1 Pre-fix

operator++(): pre-fix;

6.1.0.2 Post-fix

operator++(int): post-fix; (Remark: return type no reference)

```
complex complex{
   double re,im;
public:
   ...
   complex& operator++(){++re;++im;return *this;}
   complex operator++(int){
       complex old(re,im);
       ++re;++im;
       return old;
   }
};
```

6.1.0.3 Type conversion

Remark:No return type.

```
class complex{
   ...
   operator double() {return sqrt(re*re+im*im);}
   ...
};
```

6.1.0.4 Key Points

Constructor/Destructor **Website**

Chapter 7: Deal with Virtual

Author: Robert *Scribe: Treeman*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the author.*

7.1 Virtual Destructor

So when should I declare a destructor *virtual* ?

Whenever the class has at least one virtual function. Having virtual functions indicate that a class is meant to act as an interface to derived classes, and when it is, an object of a derived class may be destroyed through a pointer to the base.

7.1.0.5 Example

```
new_number *p = new fraction [100];
delete []p;
```

7.2 Abstract Class

Abstract class contains at least one pure virtual member function and never define an object of an abstract class. Note that if the inheriting class does not provide overiding definition for pure virtual function can be an abstract class.

7.2.0.6 Example

```
class base{
public:
   virtual void foo()=0;
};
class sub : public base{};//sub is also an abstract class
```

Template **Website**

Chapter 8: Deal with Template

Author: Robert *Scribe: Treeman*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the author.*

8.1 Template Function

8.1.0.7 Examples

```
template <typename T1, typename T2>
T absolute (const T1 &x, const T2 &y) {if(x-y<0) return y-x;else return x-y;}
```

8.2 Template Class

8.2.0.8 Examples

```
template <typename T>
class my_pair{
    T i,j;
public:
    my_pair (const T &a, const T &b): i(a), j(b) {}
}; // my_pair<int> z(1,2);
```

8.2.1 Inheritance

8.2.1.1 Examples

```
template <typename T>
class my_four: public my_pair<T>{
    T k,l;
public:
    my_four(const T &a,... ,const T &d): my_pair<T>(a,b),k(c),l(d){}
    void print(){
        cout<< my_pair<T>::i<<mypair<T>::j<<k<<l<<endl;
    }
}
```

8.3 Key Points

- type independent coding
- line-by-line the same statements
- complie-time polymorphism ↔ run-time polymorphism
- The template keyword is needed for both declaration and implementation

Overflow **Website**

Chapter 9: Deal with Range of Basic Types

Author: Robert *Scribe: Treeman*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the author.*

9.1 Factorial

Complie-Time computation and Recursive version:

```
#include <iostream>
using namespace std;

unsigned long long factorial(int N){
  if(N==1){return 1;}
  else{return N*factorial(N-1);}
}

template <long long N>
class Factorial{
public:
  enum {value=N*Factorial<N-1>::value};
};

template <>
class Factorial<0>{
public:
  enum {value=1};
};
int main(){
  int input;
  cin>>input;
  cout<<factorial(input)<<endl;
  cout<<Factorial<20>::value<<endl;
  return 0;
}
```

9.2 Key Points

- long long int(8 Bytes) will not be out of range until 20!(19 digits) while int(4 Bytes) 13!

Standard Template Library **Website**

Chapter 10: Deal with STL

Author: Robert *Scribe: Treeman*

10.1 Associative Container

They include set, multiset, map and multimap.A map, for example, is a sorted associative container that holds pairs of keys and data, where each entry's key is unique. Hence it is ideally suited to implement dictionaries, for instance.

10.1.1 Examples

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {
map<string, int> freq; // map of words and their frequencies
string word;
while (cin >> word && word != "quit") {
++freq[word];
}
// write the frequency and the word
for (map<string, int>::iterator i = freq.begin(); i != freq.end(); ++i) {
cout << i->second << " " << i->first << endl;
}
return 0;
}
```

10.2 Algorithm

```
int main(){//#include <vector>,<algorithm>,<numeric>,<iostream>//using namespace std;
  vector<int> u(10);
  for(unsigned int i=0;i<u.size();i++) u[i]=i-4;
  copy(u.begin(),u.end(),ostream_iterator<int>(cout," "));
  cout<<endl;
  vector<int>::iterator new_end=remove(u.begin(),u.end(),-2);
  // new_end=remove(u.begin(),u.end(),-2); // new_end=remove(u.begin(),u.end(),-1);
  copy(u.begin(),u.end(),ostream_iterator<int>(cout," "));
  cout<<endl;
  u.erase(new_end,u.end());
  copy(u.begin(),u.end(),ostream_iterator<int>(cout," "));
  cout<<endl;
}
```

Note that Remove delete the specified element and then push back an element which has the same value as the previous v.end(). Besides, *binary_search* should be used after sorted.

10.2.1 Compatibility

The remarkable thing about the STL algorithm is that they work with almost any kind of container, including C style arrays. Here is an example, that first sorts and then outputs a given array.

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;
int main() {
int A[] = {1, 4, 2, 8, 5, 7};
const int N = sizeof(A) / sizeof(int);
sort(A, A + N); // sort
copy(A, A + N, ostream_iterator<int>(cout, " ")); // copy : "1 2 4 5 7 8"
cout << endl;
return 0;
}
```

Tutorials **Website**

Chapter 11: Deal with Pitfalls

Author: Robert *Scribe: Treeman*

11.1 Tutorial 5

11.1.1 Polynomials

```
//const keyword lead to return the right hand value
const double &operator[] (int i) const { return a[i]; }
//no const means it is able to be as left hand of an operator.
double &operator[] (int i) { return a[i]; }

polynomial operator+(const polynomial &l) const {
polynomial res;
const polynomial *p;
if (n > l.n) { res = (*this); p = &l; }
else { res = l; p = this; }
for (int i = 0; i <= p->n; i++) res[i] += (*p)[i];
return res;
}

//operator<< overloading
friend ostream &operator<< (ostream &os, const polynomial &p) {
bool empty = true;
for (int i = p.n; i >= 0; i--) {
if (p.a[i] > 0.0) {
if (!empty) os << " + ";
os << p.a[i] << " * x^" << i;
empty = false;
} else if (p.a[i] < 0.0) {
os << " - " << -p.a[i] << " * x^" << i;
empty = false;
}
}
if (empty) os << "0";
return os;
}
```

11.2 Tutorial 6

11.2.1 The usage of ENUM

```
enum COLOUR {RED, GREEN, YELLOW, ORANGE};
string COLOUR_STRING[] = {"red", "green", "yellow", "orange"};
COLOUR color="red";
cout<<COLOUR_STRING[color]<<endl;
```

11.3 Tutorial 7

11.3.1 Overriding and Overloading

- To provide two (or more) functions that perform similar, closely related things, differentiated by the types and/or number of arguments it accepts. Contrived example:

```
void Log(std::string msg); // logs a message to standard out
void Log(std::string msg, std::ofstream); // logs a message to a file
```

- To provide two (or more) ways to perform the same action. Contrived example:

```
void Plot(Point pt); // plots a point at (pt.x, pt.y)
void Plot(int x, int y); // plots a point at (x, y)
```

- To provide the ability to perform an equivalent action given two (or more) different input types. Contrived example:

```
wchar_t      ToUnicode(char c);
std::wstring ToUnicode(std::string s);
```

In some cases it's worth arguing that a function of a different name is a better choice than an overloaded function. In the case of constructors, overloading is the only choice.

Overriding a function is entirely different, and serves an entirely different purpose. Function overriding is how polymorphism works in C++. You override a function to change the behavior of that function in a derived class. In this way, a base class provides interface, and the derived class provides implementation.

11.4 Focused Questions

- 9.1,9.2,9.3;
- 7.4,7.5,7.6
- 5.3