國立台灣海洋大學資訊工程系 C++ 程式設計 期末考試題

姓名：_____

系級：_____

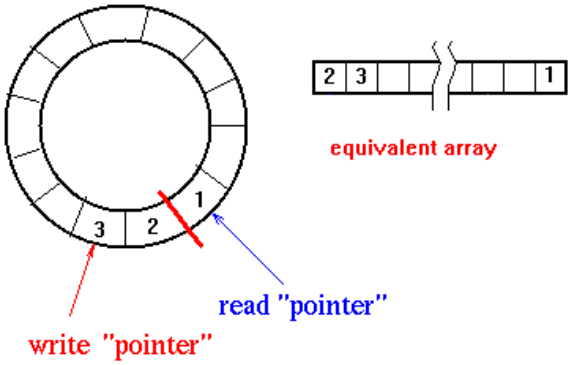
學號：_____

99/06/22

考試時間：09:30 – 12:00

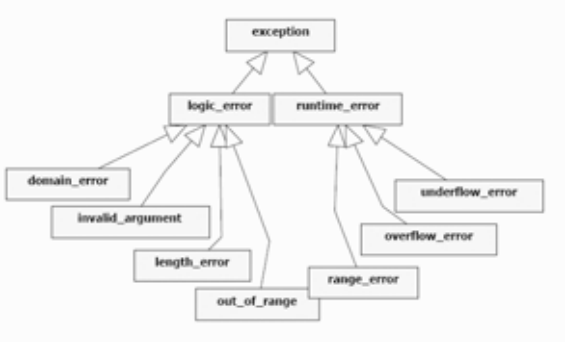
請看清楚題目問什麼,看清楚每一題所佔的分數再回答

考試規則：1. 不可以翻閱參考書、作業及程式
2. 不可以使用任何形式的電腦 (包含計算機)
3. 不可以左顧右盼、不可以交談、不可以交換任何資料、試卷題目有任何疑問請舉手發問 (看不懂題目不見得是你的問題,有可能是中英文名詞的問題)、最重要的是隔壁的答案可能比你的還差，白卷通常比錯得和隔壁一模一樣要好
4. 提早繳卷同學請直接離開教室，不可以逗留喧嘩
5. 違反上述任何一點之同學以作弊論，一律送請校方處理
6. 繳卷時請***繳交簽名過之試題卷及答案卷***



期中考時我們寫過如下的一個 CRingBuf 類別:
在串流媒體或是網路的資料處理時，常常會需要使用 ring buffer 來存放資料，如右圖的 ring buffer 中，藉由一塊記憶體空間來存放資料，概念上在 write pointer 所指的位置的下一個空格內寫入資料，寫入後 write pointer 順時鐘移動一格，由 read pointer 所指的位置讀出資料，資料讀出以後 read pointer 順時鐘移動一格，如果讀與寫的速度有一些差異, ring buffer 可以緩衝讀與寫兩端的處理速度差異，不至於發生寫得太快來不及讀導致資料流失的狀況。CRingBuf 類別資料成員包括一個 m_bufSize 整數資料成員來存放目前陣列的大小，一個 m_buf 整數指標資料成員以便記錄動態配置的陣列位址，一個 m_head 整數資料成員記錄目前 read pointer 在整數陣列中的註標，一個 m_dataCount 整數記錄目前 ring buffer 中可供讀取的資料個數, CRingBuf 類別具有下列基本的介面:

a) CRingBuf(int size): 建構元
b) ~CRingBuf(): 解構元
c) bool get(int &data): 每次由 read pointer 處取出一個整數，回傳失敗與否
d) bool put(int data): 每次於 write pointer 處將傳入的 data 記錄下來，回傳失敗與否
e) int dataCount(): 回傳目前 ring buffer 物件中有幾筆可讀出的資料
f) CRingBuf(CRingBuf &src): 拷貝建構元
g) CRingBuf & operator=(CRingBuf &rhs): 設定運算子
h) int write(int dataArray[], int dataCount);
i) bool resize(int newSize);



1. 上面這個 CRingBuf 類別中存放的資料為整數，如果希望製作一個異質容器類別 CExceptionRingBuf 其中可以存放<stdexcept>中定義的各種 exception 衍生類別之物件 (如右圖，例如: logic_error, runtime_error, domain_error, underflow_error 等等)，請運用繼承及多型的概念:

a) [5] 比照 CRingBuf 之界面定義 CExceptionRingBuf 類別之界面 (不需要實作)

Sol:

```
#include <stdexcept>
using namespace std;
class CExceptionRingBuf
{
public:
    CExceptionRingBuf(int size);
    ~CExceptionRingBuf();
    bool get(exception* &data);
    bool put(exception* data);
    int dataCount();
    CExceptionRingBuf(CExceptionRingBuf &src);
    CExceptionRingBuf & operator=(CExceptionRingBuf &rhs);
    int write(exception* dataArray[], int dataCount);
    bool resize(int newSize);
private:
    int m_bufSize;
    exception **m_buf;
    int m_head;
    int m_dataCount;
};
```

interface

b) [10] 請撰寫一個 strbuf_overflow_error 類別，繼承 overflow_error 類別，實作其建構元 (exception 類別的建構元定義為 exception(const char *))，初始化父類別物件，並且 override virtual const char* exception::what() const; 成員函式，回傳 "strbuf_overflow_error::" 字串以及 exception::what() 字串之串接字串

Sol:

```
#include <stdexcept>
using namespace std;
class strbuf_overflow_error: public overflow_error
{
public:
    strbuf_overflow_error(const char *);
    virtual const char* what() const;
};
-----
strbuf_overflow_error::strbuf_overflow_error(const char *str): overflow_error(str)
{
}
```

```
#include <string>
class strbuf_overflow_error : public overflow_error
{
        …
private:
        string m_str;
};
const char* strbuf_overflow_error::what() const
{
        m_str = "strbuf_overflow_error::";
        m_str += overflow_error::what();
        return m_str.c_str();
}
```

or

```
class strbuf_overflose_error: public overflow_error
{
        …
private:
        char *m_str;
};
strbuf_overflow_error::~ strbuf_overflow_error()
{
        delete[] m_str;
}
#include <string.h>
const char* strbuf_overflow_error::what() const
{
        char str1[] = "strbuf_overflow_error::";
        char *str2 = overflow_error::what();
        delete[] m_str;
        m_str = new char[strlen(str1)+strlen(str2)+1];
        strcpy(m_str, str1);
        strcat(m_str, str2);
        return m_str;
}
```

c) [8] 替 CExceptionRingBuf 類別製作一個 display(ostream &os) 成員函式，運用多型指標呼叫 virtual const char* exception::what() const; 成員函式將 ring buffer 中所有儲存的 exception 物件的錯誤原因字串列印到傳入的輸出串流 os 中

Sol:

```
#include <stdexcept>
```

```
#include <iostream>
using namespace std;
void CExceptionRingBuf::display(ostream &os)
{
    int i;
    for (i=m_head; i<m_head+m_dataCount; i++)
        os << m_buf[i%m_bufSize]->what() << endl;
}
```

d) [7] 請問上題中運用多型指標呼叫時編譯器採取的繫結方式為何？有什麼特性？

Sol:

動態繫結 (dynamic binding)

在編譯時 compiler 以 virtual function table 中虛擬函式的函式指標來翻譯 what() 函式的呼叫，如此透過函式指標的呼叫可以在程式執行時才根據多型指標 m_buf[i%m_bufSize] 所指到的物件來決定執行哪一個類別的 what() 函式。

2. [20] 請問 C++ 中多型 (polymorphism) 分為哪三種類型？請舉例說明？

Sol:

靜態多型 (static polymorphism):

主要就是函式多載 (function overloading) 以及運算子多載 (operator overloading)

同樣函式的名稱但是函式的 signature 不一樣，編譯器在編譯時就可以根據前後文型態來決定究竟要呼叫哪一個函式

動態多型 (dynamic polymorphism):

透過虛擬函式 (virtual function) 以及多型指標/參考 (polymorphic pointer/reference) 來達成，程式編譯時不能夠決定呼叫哪一個函式，必須要在程式執行時才能夠根據多型指標所指到的物件來決定呼叫哪一個函式

參數化多型 (parametric polymorphism):

C++ 中以樣板 (template) 來達成參數化的多型，包括樣板函式以及樣板類別

```
template <class T>
swap(T *x, T *y)
{
    T tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
template <class T>
class Vector
{
….
private:
```

```
    T *m_data;
};
```

3. 樣板 (template) 類別之設計

i. [20] 請繼承標準 C++ 函式庫中的 vector<class type> 樣板類別設計一個 sortedVector 樣板類別，並且替此衍生類別增加成員函式: (請標明 sortedVector 類別之檔案名稱)

```
void sort();                    // 以排序演算法將 sortedVector 內存放的資料由小到大排序
```

提示: 請參考下圖之 swap(int&, int&) 及 bubbleSort 函式

```
01 void swap(int &x, int &y)
02 {
03         int tmp;
04         tmp = x;
05         x = y;
06         y = tmp;
07 }
08
09 void bubbleSort(int array[], int size)
10 {
11         for (int i = 0; i < size; i++)
12                 for (int j = 0; j < (size - i - 1); j++)
13                         if (array[j] > array[j+1])
14                                 swap (array[j], array[j+1]);
15 }
```

下圖為測試程式

```
01 #include "sortedVector.h"
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05
06 #include "Student.h"
07
08 void main()
09 {
10         int i;
11         sortedVector<int> x(20);
12
13         srand(0);
14
15         for (i=0; i<20; i++)
16                 x[i] = rand()%100;
17
18         x.sort();
19
20         for (i=0; i<20; i++)
21                 cout << x[i] << ' ';
22         cout << endl;
23
24         sortedVector<Student> y;
25
26         for (i=0; i<20; i++)
27                 y.push_back(Student(i, rand()%40+60, rand()%40+60));
28
29         y.sort();
30
31         for (i=0; i<20; i++)
32                 cout << y[i];
33         cout << endl;
34 }
```

Sol:

```
---- sortedVector.h ----
#ifndef _SORTEDVECTOR_H
#define _SOTRTEDVECTOR_H

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

template <class type>
```

```
class sortedVector : public vector<type>
{
public:
	sortedVector();
	sortedVector(int);
	virtual ~sortedVector();
	void sort();
private:
	void swap(type &x, type &y);
};

template <class type>
sortedVector<type>::sortedVector()
{
}

template <class type>
sortedVector<type>::sortedVector(int size): vector<type>(size)
{
}

template <class type>
sortedVector<type>::~sortedVector()
{
}

template <class type>
void sortedVector<type>::swap(type &x, type &y)
{
	type tmp = x;
	x = y;
	y = tmp;
}

template <class type>
void sortedVector<type>::sort()
{
	for (int i = 0; i < size(); i++)
		for (int j = 0; j < (size() - i - 1); j++)
			if ((*this)[j] > (*this)[j+1])
				swap((*this)[j], (*this)[j+1]);
```

```
}

#endif    // _SOTRTEDVECTOR_H
```

ii) [30] 請實作在測試程式中可配合 sortedVector 運作之 Student 類別 (至少需要 overload operator>() 及 operator<<())

Sol:

```
---- student.h ----
#ifndef _STUDENT_H
#define _STUDENT_H

#include <iostream>
using namespace std;

class Student
{
public:
        Student(int id, int mathScore, int physicScore);
        virtual ~Student();
        int operator>(Student &rhs);
        void print(ostream &os);
private:
        int m_id;
        int m_mathScore;
        int m_physicScore;
};

ostream& operator<<(ostream &os, Student &student);

#endif // _STUDENT_H

---- student.cpp ----

Student::Student(int id, int mathScore, int physicScore)
        : m_id(id), m_mathScore(mathScore), m_physicScore(physicScore)
{
}

Student::~Student()
{
```

```
}

int Student::operator>(Student &rhs)
{
	double weightedScore_lhs, weightedScore_rhs;
	weightedScore_lhs = (m_mathScore*2 + m_physicScore) / 3.0;
	weightedScore_rhs = (rhs.m_mathScore*2 + rhs.m_physicScore) / 3.0;
	return weightedScore_lhs > weightedScore_rhs;
}

void Student::print(ostream &os)
{
	cout << "Student id: " << m_id << endl;
	cout << "		math score: " << m_mathScore << endl;
	cout << "		physic score: " << m_physicScore << endl;
}

ostream& operator<<(ostream &os, Student &student)
{
	student.print(os);
	return os;
}
```

or using friend function to implement

```
class Student
{
	….
	friend ostream& operator<<(ostream &os, Student &student);
	….
};
ostream& operator<<(ostream &os, Student &student)
{
	cout << "Student id: " << student.m_id << endl;
	cout << "		math score: " << student.m_mathScore << endl;
	cout << "		physic score: " << student.m_physicScore << endl;

	return os;
}
```