

LẬP TRÌNH C/C++ NÂNG CAO

Yêu cầu trước khi đọc: học xong Lập trình C/C++ căn bản

BÀI 1: NHẮC LẠI VỀ C/C++

Nhập xuất cơ bản

CODE

```
#define max(a,b) (a>b)?a:b      //khai báo macro
typedef unsigned int byte;       //định nghĩa kiểu dữ liệu
const float PI=3.14;            //khai báo hằng số
char c;char s[20];
```

Cách của C

CODE

```
//không dùng scan nếu muốn nhập khoảng trắng
gets(s);           //có thẻ nhập khoảng trắng
puts(s);
fflush(stdin);    //xóa bộ đệm nhập
c=getchar();
putchar@;
```

Cách của C++

CODE

```
//không dùng cin>> nếu muốn nhập khoảng trắng
cin.getline(a,21); //có thẻ nhập khoảng trắng
cout<<a;
cin.get();         //xóa bộ đệm nhập
```

Con trỏ cơ bản

CODE

```
int a=5,*p;
//p=3;          //khong hop ve vi khong the gan gia tri kieu int cho bien kieu int*
//&p=3;         //khong hop le vi dia chi cua p la co dinh
p=&a;          //hop le, gan dia chi ma p tro den
*p=3;          //hop le, gan gia tri tai dia chi ma p tro den
cout<<p<<endl; //cai gi do bat ki, dia chi cua a
cout<<&p<<endl; //cai gi do bat ki, dia chi cua p
cout<<*p<<endl; //3,dau * luc nay mang y nghia "gia tri tai dia chi cua"
```

Truyền giá trị cho hàm

Trong C có khái niệm con trỏ (pointer) Trong C++ có thêm khái niệm tham chiếu (reference)

CODE

```
int a;
int& b=a;
```

Lúc này biến a có một cái nickname là b

Như vậy có tất cả 3 cách viết hàm và truyền tham số

Cách 1:

CODE

```
void add10(int a)
{
    a=a+10;
}
```

```
gọi:  
add10 (n);
```

Không hiệu quả, a vẫn giữ nguyên giá trị

Cách 2:

CODE

```
void add10(int *a)  
{  
    *a=*a+10;  
}  
gọi:  
add10 (&n);
```

Hiệu quả.

Cách 3:

CODE

```
void add10(int &a)  
{  
    a=a+10;  
}  
gọi:  
add10 (n);
```

Hiệu quả, tiện hơn cách 2.

Nhập xuất dữ liệu với kiểu mảng số nguyên

CODE

```
int a[3];
```

Truyền dữ liệu trực tiếp theo kiểu C, cách 1

CODE

```
for(int i=0;i<3;++i) scanf("%d",&(*(a+i)));  
for(int i=0;i<3;++i) printf("%d",*(a+i));
```

Truyền dữ liệu trực tiếp theo kiểu C, cách 2

CODE

```
for(int i=0;i<3;++i) scanf("%d",&a[i]);  
for(int i=0;i<3;++i) printf("%d",a[i]);
```

Truyền dữ liệu trực tiếp theo kiểu C++, cách 1

CODE

```
for(int i=0;i<3;++i) cin>>*(a+i);  
for(int i=0;i<3;++i) cout<<*(a+i);
```

Truyền dữ liệu trực tiếp theo kiểu C++, cách 2

CODE

```
for(int i=0;i<3;++i) cin>>a[i];  
for(int i=0;i<3;++i) cout<<a[i];
```

Nhập xuất dữ liệu bằng hàm với kiểu mảng số nguyên

Nhập xuất dữ liệu bằng hàm với kiểu mảng số nguyên theo kiểu C, cách 1

CODE

```
void input(int[]);
input(a);
void input(int *a)
{
    for(int i=0;i<3;++i)
        scanf("%d",&(*(a+i)));
}
void output(int[]);
output(a);
void output(int *a)
{
    for(int i=0;i<3;++i)
        printf("%d",*(a+i));
}
```

Nhập xuất dữ liệu bằng hàm với kiểu mảng số nguyên theo kiểu C, cách 2

CODE

```
void input(int[]);
input(a);
void input(int a[])
{
    for(int i=0;i<3;++i)
        scanf("%d",&a[i]);
}
void output(int[]);
output(a);
void output(int a[])
{
    for(int i=0;i<3;++i)
        printf("%d",a[i]);
}
```

Nhập xuất dữ liệu bằng hàm với kiểu mảng số nguyên theo kiểu C++, cách 1

CODE

```
void input(int[]);
input(a);
void input(int *a)
{
    for(int i=0;i<3;++i)
        cin>>*(a+i);
}
void output(int[]);
output(a);
void output(int *a)
{
    for(int i=0;i<3;++i)
        cout<<*(a+i);
}
```

Nhập xuất dữ liệu bằng hàm với kiểu mảng số nguyên theo kiểu C++, cách 2

CODE

```
void input(int[]);
input(a);
void input(int *a)
{
    for(int i=0;i<3;++i)
        cin>>*(a+i);
}
void output(int[]);
output(a);
void output(int *a)
{
    for(int i=0;i<3;++i)
        cout<<*(a+i);
}
```

```
input(a);
void input(int a[])
{
    for(int i=0;i<3;++i)
        cin>>a[i];
}
void output(int[]);
output(a);
void output(int a[])
{
    for(int i=0;i<3;++i)
        cout<<a[i];
}
```

Nhập xuất dữ liệu với kiểu mảng số thực

Cách dùng biến tạm

CODE

```
float a[2][3],temp;
for(int i=0;i<2;++i)
    for(int j=0;j<3;++j)
    {
        scanf("%f \n",&temp);
        a[i][j]=temp;
    }
```

Cách dùng con trỏ

CODE

```
float a[2][3];float *p;
p=(float*)a;
for(int i=0;i<2*3;++i)
    scanf("%f", (p+i));
```

Nhập mảng số thực 2 chiều bằng cách dùng ép kiểu

CODE

```
float a[3][2];float *p;p=(float*)a;
for(int i=0;i<3;i++)
    for(int j=0;j<2;j++)
        scanf("%f", ((float*)p+i*2+j));
```

Xuất mảng số thực 2 chiều bằng cách dùng ép kiểu

CODE

```
float a[3][2];float *p;p=(float*)a;
for(int i=0;i<3;i++)
    for(int j=0;j<2;j++)
        printf("%f\n", *(p+i*2+j));
```

Nhập mảng số thực 2 chiều bằng cách dùng malloc

CODE

```
float** p;p=(float**)malloc(2);
for(int i=0;i<3;i++)
    for(int j=0;j<2;j++)
        scanf("%f", (p+i*2+j));
```

Xuất mảng số thực 2 chiều bằng cách dùng malloc

CODE

```
float** p;p=(float**)malloc(2);
for(int i=0;i<3;i++)
    for(int j=0;j<2;j++)
        printf("%f\n",*(p+i*2+j));
```

Bài này chỉ có giá trị tham khảo, tổng hợp kiến thức.

BÀI 2: NHẮC LẠI VỀ C/C++ (TIẾP THEO)

Cấu trúc (struct)

Con trỏ cấu trúc (struct pointer)

CODE

```
struct Student
{
    int id;
};

Student *s;
Student m;
s=&m;
s->id=3;      //means (*s).id
cout<<m.id;
```

Sao chép cấu trúc

CODE

```
struct Student
{
    int id;
    char *name;          //một con trỏ, không phải một mảng
};

Student a;
char temp[20];
cin>>temp;
a.name=new char[strlen(temp)+1];
strcpy(a.name,temp);        //phải dùng biến tạm
Student b=a;
strcpy(b.name,a.name);      //phải dùng strcpy, nếu không sẽ sao chép địa chỉ bộ nhớ
```

Gọi hàm với cấu trúc

CODE

```
struct Student{
    char name[10];
    int id;
};

Student m[3],a;
m[0]=(Student){"Pete",1};
add(m[0].name,&m[0].id);
```

Có 4 cách để thêm dữ liệu vào cấu trúc.

Cách 1

CODE

```
void add(char name[],int *place)
{
    cin>>name;
    cin.get();
    cin>>*place;
}
add(a.name,&a.id);
```

Cách 2

CODE

```
void add(Student &s)
{
    cin>>s.name;
    cin.get();
    cin>>s.id;
}
add10(a);
```

Cách 3

CODE

```
void add(Student *s)
{
    cin>>(*s).name;
    cin.get();
    cin>>(*s).id;
}
add(&a);
```

Cách 4

CODE

```
void add(Student *s)
{
    cin>>s->name;
    cin.get();
    cin>>s->id;
}
add(&a);
```

Toán tử sizeof với struct

CODE

```
struct Hello
{
    char c;
    double d;
};
```

sizeof(Mystruct)=12; vì c lấy một 32-bit word (4 byte, không phải 1 byte)

Con trỏ (pointer)

Con trỏ trỏ đến một con trỏ khác

CODE

```
char a='z';      //a='z' và giá trị địa chỉ của a=8277
char *p=&a;      //p=8277 và giá trị địa chỉ của p=6194
char **p2=&p;    //p2=6194 và địa chỉ của p2 sẽ là một cái gì đó
```

Con trỏ void (void pointer)

Con trỏ void dùng để trỏ đến bất cứ một kiểu dữ liệu nào

CODE

```
void increase(void* data,int dataType)
{
    switch(dataType)
    {
        case sizeof(char):
```

```

(*((char*)data))++;break;
case sizeof(int):
    (*((int*)data))++;break;
}
}
int main()
{
    char c=66;int a=-4;
    increase(&c,sizeof(char));
    increase(&a,sizeof(int));
}

```

Con trỏ hàm (function pointer)

Con trỏ hàm dùng để trỏ đến một hàm

CODE

```

int addition(int a,int b)
{
    return a+b;
}
int subtraction(int a,int b)
{
    return a-b;
}
int (*minuse)(int,int) = subtraction;
int primi(int a,int b,int(*functocall)(int,int))
{
    return (*functocall)(a,b);
}
int main()
{
    int m=primi(7,5,&addition);
    int n=primi(20,m,minuse);
    cout<<m<<endl;cout<<n<<endl;
    return 0;
}

```

Hàm nội tuyến (inline function)

Hàm khai báo với từ khóa inline, trình biên dịch sẽ chèn toàn bộ thân hàm mỗi nơi mà hàm đó được sử dụng. Với cách này, các hàm inline có tốc độ thực thi cực nhanh, nên sử dụng với các hàm thường xuyên phải sử dụng trong chương trình.

CODE

```

inline void display(char *s)
{
    cout<<s<<endl;
}
int main()
{
    display("Hello");return 0;
}

```

Nhập xuất với tập tin

CODE

```

#include <iostream>
#include <iomanip>

```

```
int number;
ifstream inf;ofstream outf;
inf.open("input.txt");
outf.open("output.txt");
while(in>>number)
outf<<"Next is"<<setw(4)<<number<<endl;
inf.close();
outf.close();
```

Mở một file dùng cho cả nhập và xuất

CODE

```
fstream f;
f.open("st.txt",ios :: in | ios :: out);
một số chế độ hay dùng
ios :: in nghĩa là nhập vào
ios::out nghĩa là xuất ra tập tin từ đầu tập tin
ios::app nghĩa là thêm dữ liệu vào tập tin (appending)
```

Tập tin header

Tạo một tập tin header có tên là myfile.h

```
#ifndef MYFILE_H
#define MYFILE_H
.....
#endif
```

trong tập tin cpp thêm vào dòng

```
#include "myfile.h"
```

BÀI 3: NHẮC LẠI VỀ LỚP

Cơ bản về lớp

CODE

```
class Date{  
    int day;  
public:  
    Date(int, int a=1);  
    int month;  
    void setDate(int);  
    void output();  
};  
int main(){  
    Date d(6);  
    d.month=3;  
    d.setDate(25);  
    d.output();  
    return 0;  
}  
Date::Date(int day,int month){  
    this->day=day;  
    this->month=month;  
}  
void Date::setDay(int day){  
    this->day=day;  
}  
void Date::output(){  
    cout<<day<<"/"<<month;  
}
```

Hàm khởi tạo

Chúng ta có thể viết một hàm khởi tạo như thế này

CODE

```
class Student  
{  
    string name; int age;  
public:  
    Student(string name,int n):name(name),age(n)  
    {  
    }  
};
```

Nó tương đương với

CODE

```
class Student  
{  
    string name; int age;  
public:  
    Student(string name,int n)  
    {  
        (*this).name = name;  
        this->age = n;  
    }  
};
```

Hàm bạn (friend function)

CODE

```
class Student{
public:
    int id;
    friend bool equal(const Student&, const Student&);
};

int main(){
    Student s1;s1.id=2;
    Student s2;s2.id=3;
    cout<<equal(s1,s2);
}

bool equal(const Student& s1,const Student& s2) {
    return (s1.id==s2.id);
}
```

Overload toán tử (operator overload)

Ví dụ dưới đây sẽ overload toán tử ==

CODE

```
class Student{
public:
    int id;
    friend bool operator==(const Student&,const Student&);
};

int main(){
    Student s1;s1.id=2;
    Student s2;s2.id=3;
    cout<<((s1==s2)?"equal":"unequal");
}

bool operator==(const Student& s1,const Student& s2) {
    return (s1.id==s2.id);
}
```

Overload toán tử nhập và xuất (input >> và output <<)

Mọi người đều biết cin>>a là gọi toán tử nhập cin.operator>>(a) hoặc operator>>(cin,a) Overload 2 toán tử nhập và xuất này hết sức quan trọng về sau. Nhân tiện mỗi khi cấp phát bộ nhớ, dùng xong phải luôn hủy đi để thu hồi lại bộ nhớ đã cấp phát. Vì về sau game cái ưu tiên hàng đầu là bộ nhớ, đừng để lại rác.

CODE

```
class Date{
public:
    int day,int month;
    friend istream& operator>>(istream&,Date&);
    friend ostream& operator<<(ostream&,const Date&);
};

istream& operator>>(istream& ins,Date& d) {
    ins>>d.day;
    ins>>d.month;
    ins.get(); //phải xóa bộ đệm
    return ins;
}

ostream& operator<<(ostream& outs,const Date& d) {
    outs<<d.day<<" / "<<d.month;
```

```

        return outs;
}
int main(){
    Date d;
cin>>d;cout<<d;
    Date *dt=new Date;      //phải tạo object pointer, cấp phát bộ nhớ
    cin>>*dt;cout<<*dt;
    delete dt;           //phải hủy object pointer
}

```

Hàm hủy (destructor)

CODE

```

class myclass{
public:
    int *p;
    myclass();
    ~myclass();
};

int main(){
    myclass m;
    return 0;
}

myclass::myclass(){
    p=new int;      //phải cấp phát bộ nhớ để tránh segmentation fault
}

myclass::~myclass(){
    delete p;
}

```

Hàm khởi tạo sao chép (copy constructor)

CODE

```

class Date{
public:
    int day,int month,char *special;
    Date(int,int,char*);
    Date(const Date&);

    ~Date(){
        delete [] special;    //bởi vì chúng ta cấp phát bộ nhớ cho nó
    }
};

Date::Date(int day,int month,char *special){
    this->day=day;this->month=month;this->special=special;
}

Date::Date(const Date& d){
    this->day=d.day;this->month=d.month;
    this->special=new char[strlen(d.special)+1];          //cấp phát bộ nhớ cho nó
    strcpy(this->special,d.special);           //phải dùng strcpy với char array
}

int main(){
    Date d1(29,8,"birthday");
    Date d2(d1);
    cout<<d2.special;
    return 0;
}

```

Chú ý về cấp phát bộ nhớ

Điều gì sẽ xảy ra khi chúng ta không thể cấp phát bộ nhớ ? Ví dụ chúng ta viết 1 game RTS mà mỗi phe tham chiến có 10 tì quân ?

Giải quyết khi không thể cấp phát bộ nhớ thành công

Chúng ta vẫn thường cấp phát bộ nhớ như sau

CODE

```
char *p;int i;  
cout<<"number of element u want:";  
cin>>i;  
p=new char[i+1];  
delete [] p;
```

Nếu chúng ta không thể cấp phát bộ nhớ ? CPP sẽ ném (throw) ra một ngoại lệ. Có 2 cách để xử lý chuyện này

Cách một là dùng từ khóa nothrow. Vì thế CPP vẫn tạo ra một pointer nhưng là 0

CODE

```
p=new (nothrow) char[i+1];  
if(p==0) cout<<"Can't allocate memory";
```

Cách hai là bắt cái ngoại lệ ấy, Đó là ngoại lệ std::bad_alloc

CODE

```
try{  
    p=new char[i+1];  
}catch(std::bad_alloc &mae){  
    cerr<<"failed to allocate memory"<<mae.what();  
    exit(1);  
}
```

Cấp phát bộ nhớ trong C

Đừng có chỉ mê new và delete không thôi, cấp phát với cách của C vẫn phải dùng về sau đây

CODE

```
char *p;int i;  
printf("number of element u want:");  
scanf("%d",&i);  
p=(char*)malloc(i+1);  
if(p==NULL) exit(1);  
free(p);  
hoặc chúng ta có thể dùng calloc  
p=(char*)calloc(i,sizeof(char));
```

Toán tử gán (assignment operator)

CODE

```
class Base{  
public:  
    Base& operator=(const Base&);  
    friend bool operator!=(const Base&,const Base&);  
private:  
    char* c;  
};  
Base& Base::operator=(const Base& src){  
    if(*this!=src){           //to avoid self-assignment  
        delete [] c;
```

```

c = new char[strlen(src.c)+1];
strcpy(this->c,src.c);
}
return *this;
}
bool operator!=(const Base& b1,const Base& b2) {
    return(strcmp(b1.c,b2.c));
}
Và chúng ta có thể gọi toán tử này
Base s2=s1;

```

Thừa kế (inheritance)

Trong C có thể sinh ra bug, trong C++ chúng sẽ được thừa kế.

CODE

```

class Base{
protected:
    int id;
    Base(int id){
        this->id=id;
    }
};
class Sub:public Base{
public:
    int code;
    Sub(int code,int id):Base(id){
        this->code=code;
    }
};

```

Hàm ảo (virtual function)

Hàm Play trong lớp MusicPlayer là một hàm ảo (virtual function)

CODE

```

class MusicPlayer{
public:
    virtual void Play() {
        cout<<"Play on what ?"<<endl;
    }
};
class DVD:public MusicPlayer{
public:
    void Play() {
        cout<<"Play on DVD"<<endl;
    }
};
int main(){
    MusicPlayer m;m.Play();
    DVD d(2);d.Play();
}

```

Bây giờ chúng ta sẽ làm hàm Play trong lớp MusicPlayer là một hàm thuần ảo (pure virtual function), đồng thời làm lớp MusicPlayer trở thành một lớp trừu tượng (abstract class), chúng ta sẽ không thể tạo instance của nó được nữa

CODE

```

class MusicPlayer{

```

```

public:
    virtual void Play() = 0;
};

class DVD:public MusicPlayer{
public:
    void Play(){
        cout<<"Play on DVD"<<endl;
    }
};

int main(){
    DVD d(2);d.Play();
}

```

Chúng ta tạo con trỏ để trỏ đến các subclass của nó

CODE

```
MusicPlayer *m=new DVD(5);m->play();
```

Chúng ta cũng có thể tạo mảng các con trỏ của một lớp trừu tượng

CODE

```

class MusicPlayer... là một lớp trừu tượng
class DVD:public MusicPlayer...
class CD:public MusicPlayer...
MusicPlayer *m[2];
m[0]=new DVD(5);m[0]->play();
m[1]=new CD("Sony");m[1]->play();

```

Nhắc lại một chút về mảng các kí tự (char array)

CODE

```

char destArray[10];char srcArray[]{"panther"};
strcpy(destArray, srcArray);
strcpy(destArray, srcArray,strlen(srcArray));
strcat(s1,s2);           //thêm (append) s2 vào s1
strncat(s1,s2,n);       //thêm (append) n kí tự đầu tiên của s2 vào s1
strlen(char *s);         //độ dài (length) của char array, không bao gồm "end of char array marker"
char *a;char b[];strcmp(a,b);           //trả về 0 nếu bằng,-1 nếu a<b,1 nếu a>b
atoi, atof, atoll convert một char array thành integer, float hay long, 3 hàm này trong stdlib.h
char *s = "123.45";
int i=atoi(s);
float f=atof(s);

```

Nhắc lại một chút về chuỗi (string)

CODE

```

using std::string;
*khai tạo (constructor)
string s1;string s2("Hello boy");string s3(s2);
string s4(s2,3,4);      //sao chép từ kí tự thứ 3, sao chép 4 kí tự
string s5(8,'*');      //khai tạo chuỗi gồm toàn dấu *
*toán tử gán (assignment)
string s4=s2;string s5.assign(s3);
*so sánh chuỗi (compare string)
if(s1==s2)              //bây giờ có thể dùng == rồi
if(s1.compare(s2))
*cộng chuỗi

```

```
string s1,s2;s1+=s2;s1+='o';
s1.append(s2);           //y nhu s1+=s2
s1.append(s2,3,string::npos);      //thêm vào s1 từ kí tự thứ 3 đến hết s2
s1.insert(7,s2);        //thêm s2 vào sau kí tự thứ 7 của s1
*kích cỡ (capacity)
s.capacity() trả về kích cỡ tối đa
if s.size()==15, s.capacity()==16 (16-byte)
if s.size()==17, s.capacity()==32 (two 16-byte)
*truy xuất chuỗi
#include <stdexcept>
try{
    cout<<s.at(100);
}catch(out_of_range& e){
    cout<<"invalid index";
}
```

BÀI 4: TEMPLATE

Hàm template

Giả sử chúng ta cần viết một hàm trả về số nguyên lớn nhất giữa 2 số

CODE

```
int maximum(int a,int b)
{
    return (a>b)?a:b;
}
```

Rồi đến số thực chúng ta cũng làm như vậy

CODE

```
double maximum(double a,double b)
{
    return (a>b)?a:b;
}
```

div, id: post-25916, class: postcolor

Rồi giả sử như với lớp Person chúng ta cũng phải làm như vậy (toán tử > đã được overload)

CODE

```
Person maximum(Person a,Person b)
{
    return (a>b)?a:b;
}
```

C++ cung cấp một giải pháp cho vấn đề này, đó là template

CODE

```
template<class T>T maximum(T a,T b)
{
    return (a>b)?a:b;
}
int main()
{
    int a=7;int b=5;
    cout<<maximum(a,b);
return 0
}
```

template với nhiều hơn một kiểu dữ liệu

CODE

```
template<class T,typename U>void func(T a,U b);
```

Dùng template với mảng

CODE

```
template<class T,int size>void print(T (&a)[size])
{
    for(int i=0;i<size;i++) cout<<a[i]<<endl;
}
```

Lớp template (template class)

CODE

```
template<class T>class pair
{
    T values[2];
public:
    pair(T first,T second)
    {
        values[0]=first; values[1]=second;
    }
    T getmaximum();
};
template<class T>T pair<T>::getmaximum()
{
    return (values[0]> values[1])? values[0]: values[1];
}
```

Trong hàm main

CODE

```
pair<int> myobject(155,36);
myobject.getmaximum();
```

Thật tuyệt, đúng không ?

Vấn đề không đơn giản như vậy.

Đau đầu

Xem lại hàm template dưới đây

CODE

```
template<class T>T maximum(T a,T b)
{
    return (a>b)?a:b;
}
```

Ví dụ dưới đây thực ra là đang so sánh địa chỉ bộ nhớ (memory address) của 2 biến a và b

CODE

```
char* a = "hello";char* b = "world";
cout<<maximum(a,b);
```

Ví dụ dưới đây cũng là đang so sánh địa chỉ bộ nhớ (memory address) của 2 biến a và b

CODE

```
int a[3],b[3];
cout<<maximum(a,b);
```

Vậy phải làm sao ?

(Trong lập trình, những vấn đề tưởng như nhỏ nhặt thế này thực ra gây đau đầu lắm đó, nhất là khi phải làm dự án từ 1000 words trở lên. Mà đặc biệt riêng lập trình game đụng những chuyện đau đầu này thường xuyên hơn các phần ngành IT khác. Biển dịch thành công, mà tại sao nó ... kì cục vậy nè ?)

Cứu tinh xuất hiện, đó là một tham chiếu mà tham chiếu đến một con trỏ (a reference which refers to a pointer). Đây là dạng đau đầu nhất của tham chiếu.

A reference which refers to a pointer**CODE**

```
int* p; //một con trỏ p bình thường
int*& r = p; //tham chiếu r là nickname mới của p
r = new int; //tương đương với p = new int
*r = 5; //tương đương với *p = 5
cout<<*p; //tương đương với cout<<*r
```

Và như vậy, vấn đề khó khăn với dữ liệu kiểu mảng đã được giải quyết.

CODE

```
template<class T>T* maximum(T*& a,T*& b)
{
    return (*a>*b)?a:b;
}
int main()
{
    char* a="bb";
    char* b="aa";
    cout<<maximum(a,b);
    return 0;
}
```

Lưu ý là chỉ có "một tham chiếu mà tham chiếu đến một con trỏ" và "một con trỏ mà trỏ đến một con trỏ khác", chứ không thể có những khái niệm như "một tham chiếu mà tham chiếu đến một tham chiếu khác" hay "một con trỏ mà trỏ đến một tham chiếu" đâu nhá.

Hết khó khăn chưa ? Chưa đâu.

BÀI 5: TEMPLATE (TIẾP)

Lại đau đầu

Ta muốn viết một chương trình tìm kiếm phần tử trong một mảng. Ta viết như sau

CODE

```
template<class T>int search(T a[],int n,T key)
{
    int index=0;
    while(index<n && a[index] != key) index++;
    if(index == n) return -1;else return index;
}
```

Sau đó trong hàm main ta viết

CODE

```
char *list[]={ "zero", "one", "two" };      //thực ra là mảng 2 chiều thôi
search(list,3,"two");          //ò khong, lại so sánh memory address nữa rồi
```

Nhưng lần này vẫn dễ phức tạp hơn nhiều. Ví dụ nếu là mảng các Person là đụng thêm vấn đề cấp phát bộ nhớ nữa

Giải quyết

Chương trình dưới đây trình bày cách tạo một lớp mảng template, với đủ các chức năng tạo, thêm, truy xuất dữ liệu, toán tử [].

Đặc biệt là giải quyết đau đầu tìm kiếm dữ liệu ở trên vì so sánh memory address. Lưu ý là khi tạo ta phải dùng reference refers to pointer để cấp phát bộ nhớ đó

CODE

```
#include <iostream>
using namespace std;
template<class T>class Array
{
    T* array;int size;
public:
    Array(int n);
    ~Array();
    void setValue(const T&,int n);      //thiết lập dữ liệu
    T& getValue(int n);            //truy xuất dữ liệu
    void makeArray(T *&arr,int n);      //tạo mảng
    T& operator[](int i);           //toán tử [] truy xuất dữ liệu mảng
    int seek(const T& key);        //tìm kiếm trong mảng gọi hàm
    int search(const T* list,int size,const T key);      //tìm kiếm trong mảng có sẵn
};
template<typename T>Array<T>::Array(int n)
{
    size=n;
    array = new T[size];
}
template<typename T>Array<T>::~Array()
{
    delete [] array;
}
template<typename T>void Array<T>::setValue(const T& value,int n)
{
    *(array+n) = value;
}
template<typename T>T& Array<T>::getValue(int n)
{
    return *(array+n);
```

```

} template<typename T>void Array<T>::makeArray(T *&arr,int n)
{
    arr = new T[n];
}
template<typename T>T& Array<T>::operator[](int i)
{
    return *(array+i);
}
template<typename T>int Array<T>::seek(const T& key)
{
    int index=0;
    while((index<size) && *(array+index) !=key) ++index;
    if(index==size) return -1;
    else return index;
}
template<typename T>int Array<T>::search(const T* list,int size,const T key)
{
    int index=0;
    while((index<size) && *(list+index) !=key) ++index;
    if(index==size) return -1;
    else return index;
}
class Person
{
    int age;
public:
    Person(){age=0;}
    Person(int age){this->age=age;}
    int getAge() const{return age;}
    friend bool operator!=(const Person& p1,const Person& p2)
    {
        return p1.getAge()!=p2.getAge();
    }
    friend ostream& operator<<(ostream& os,const Person& p)
    {
        os<<p.getAge()<<endl;
        return os;
    }
};
int main()
{
    Array<Person> a(3);
    a.setValue(Person(5),2);
    cout<<a[2];
    Person* b;
    a.makeArray(b,4);
    for(int i=0;i<4;i++) *(b+i)=Person(i+2);
    cout<<a.seek(Person(5))<<endl;
    cout<<a.search(b,4,Person(4))<<endl;
    return 0;
}

```

Có vẻ đã xong. Hết rắc rối rồi.

Chưa. Vẫn còn 2 rắc rối nữa. Bạn hãy thử viết toán tử output << cho một mảng template class hay so sánh giữa hai mảng

template class như trên thử xem.

Bạn sẽ không viết được đâu nếu không sử dụng cái này: prototype template function (khai báo nguyên mẫu cho hàm template) (Học mấy cái điên đầu này làm gì nhỉ ? Làm gì à ? Hãy thử cho hai cầu thủ trong một game đá banh đối diện nhau. Họ có bao nhiêu hành động có thể làm được lúc đó ? Chuyền bóng ? Lừa bóng ? Đốn ? special Zidane-style skill ? Mike Tyson skill ? Hai mảng các hành động ấy phải đem ra mà chơi lẫn nhau. Bởi thế mang tiếng là "Advance C++" nhưng thực ra trong lập trình game vẫn chỉ là "newbie")

prototype template function

Chuẩn bị một tập tin tên là "array.h"

CODE

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iostream>
using namespace std;
template<class T>class Array;
template<typename T>bool equal(const Array<T>&, const Array<T>&);
template<typename T>ostream& operator<<(ostream&, const Array<T>&);
template<class T>class Array
{
    T* array; int size;
public:
    Array(int n);
    ~Array();
    void setValue(const T&, int n);
    friend bool equal <>(const Array<T>&, const Array<T>&);
    friend ostream& operator<< <>(ostream&, const Array<T>&);
};
#include "array.cpp"
#endif
```

Chuẩn bị một tập tin tên là "array.cpp"

CODE

```
template<typename T>Array<T>::Array(int n)
{
    size=n;
    array = new T[size];
}
template<typename T>Array<T>::~Array()
{
    delete [] array;
}
template<typename T>void Array<T>::setValue(const T& value, int n)
{
    *(array+n) = value;
}
template<typename T>bool equal(const Array<T>& a1, const Array<T>& a2)
{
    return a1.size==a2.size;
}
template<typename T>ostream& operator<<(ostream& os, const Array<T>& a)
{
    for(int i=0;i<a.size;++i) os<<*(a.array+i);
    return os;
}
```

CODE

```
#include "array.h"
class Person
{
    int age;
public:
    Person()
    {
        age=0;
    }
    Person(int age)
    {
        this->age=age;
    }
    int getAge() const
    {
        return age;
    }
    friend bool operator!=(const Person& p1,const Person& p2)
    {
        return p1.getAge() != p2.getAge();
    }
    friend ostream& operator<<(ostream& os,const Person& p)
    {
        os<<p.getAge()<<endl;
        return os;
    }
};
int main()
{
    Array<Person> a(3);
    a.setValue(Person(24),0);
    a.setValue(Person(15),1);
    a.setValue(Person(5),2);
    cout<<a;
    Array<Person> b(3);
    cout<<equal(a,b)<<endl;
    return 0;
}
```

Giải thích: equal và operator<< đều là hai hàm bạn, do đó để hoạt động cần có sẵn lớp Array. Nhưng lớp Array muốn biên dịch được phải cần có hai hàm này. Do đó ta phải khai báo prototype của hai hàm này trước. Nhưng vì đây là 2 template function, nên khi khai báo lại prototype của chúng lần thứ hai trong một class template (ở đây là class Array) ta phải có cái kí hiệu này <>. Khi đó là một prototype template function. Khi đó, thay vì tập tin cpp chứa thân hàm include tập tin header chứa nguyên mẫu của hàm, ta phải làm ngược lại. Kĩ thuật này hiểu và ứng dụng cực kì rắc rối nhưng khổ nỗi lại áp dụng rất nhiều về sau, đặc biệt khi làm các game lớn.

Biên dịch lại mã này với GCC

Không bắt buộc, nhưng nên làm nếu như sau này bạn có định làm việc với game trong môi trường *nix và console. Hãy đem 3 tập tin này (array.h, array.cpp, main.cpp) và thử biên dịch bằng GCC trong Linux thử xem. Nhớ tạo makefile. Trong trường bạn tôi chủ yếu làm việc bằng GCC và VI trong *nix chứ không phải Window. Việc sử dụng các bộ Visual Studio tuy không bị cấm nhưng không được khuyến khích. Và bài tập lần bài thi đều phải submit nguyên project kèm makefile để biên dịch trong môi trường *nix hết.

Viết operator overload và copy constructor

Trong phần trước ta đã xem các ví dụ dùng cách “tham chiếu mà tham chiếu đến con trỏ” Trong phần này chúng ta sẽ overload toán tử = và viết copy constructor cũng sử dụng lại cách này, mà không phải dùng đến prototype template function

CODE

```
#include <iostream>
#include <string>
using namespace std;
template<typename T>
class Array
{
public:
    int size;
    T* elems;
    Array(int);
    Array(const Array<T>*&);
    void setValue(const T&,int i);
    T& getValue(int n);
    Array<T>& operator=(const Array<T>*&);
    friend bool operator!=(const Array<T>&,const Array<T>&);
};

template<typename T>
Array<T>::Array(int size)
{
    elems = new T[size];
}

template<typename T>
void Array<T>::setValue(const T& value,int i)
{
    *(elems+i) = value;
}

template<typename T>
T& Array<T>::getValue(int i)
{
    return *(elems+i);
}

template<typename T>
Array<T>::Array(const Array<T>*& src)
{
    size=src.size;
    elems = new T[size];
    for(int i=0;i<size;i++)
        *(elems+i) = *(src.elems+i);
}

template<typename T>
Array<T>& Array<T>::operator=(const Array<T>*& src)
{
    if(*this!=src)      //to avoid self-assignment
    {
        size=src.size;
        elems = new T[size];
        for(int i=0;i<size;i++)
            *(elems+i) = *(src.elems+i);
    }
    return *this;
}

template<typename T>
```

```
bool operator!=(const Array<T>& a1,const Array<T>& a2)
{
    if(a1.size!=a2.size) return true;
    else for(int i=0;i<a1.size;i++)
        if(*(a1.elems+i) == *(a2.elems+i)) return false;
    return true;
}
int main()
{
    Array<string> a(2);
    a.setValue("hello",0);
    a.setValue("world",1);
    Array<string> b(3);
    b = a;
    cout<<b.getValue(0)<<endl;
    cout<<b.getValue(1)<<endl;
    return 0;
}
```

BÀI 6: TEMPLATE (TIẾP THEO)

Trình biên dịch và template

Trong bài trước chúng ta thấy một điều hơi lạ, đó là file header array.h có chỉ thị #include file source array.cpp. Tại sao như vậy?

Khi trình biên dịch gặp template, nó kiểm tra cú pháp, nhưng không biên dịch ngay.

Ví dụ nó gặp template<class T> nó không thể biên dịch vì nó không biết kiểu dữ liệu của T.

Khi nó gặp instance đầu tiên của template, ví dụ template<int> nó biên dịch và chúng ta có phiên bản với kiểu dữ liệu int của template.

Khi nó gặp instance thứ hai của template, ví dụ template<double> nó cũng lại biên dịch và chúng ta có phiên bản thứ hai của template, phiên bản với kiểu dữ liệu double. Vân vân.

Thông thường chúng ta viết định nghĩa lớp và nguyên mẫu các hàm của lớp đó ở file header (đuôi .h) rồi mới viết thân cho các hàm đó ở một file source (đuôi .cpp), mà file cpp này include luôn file header đó.

Template phải làm ngược lại. Vì lí do nói trên, cả định nghĩa lớp, nguyên mẫu các hàm lẫn thân của các hàm đó của một lớp template phải được biên dịch cùng nhau. Do đó khi tách rời định nghĩa của một lớp template ra chứa trong một file header riêng, file header đó phải include file source chứa thân các hàm của lớp template đó, rồi một file nào khác muốn dùng template đó phải include cái file header đó.

Ở đây còn một phần nữa về export, tôi đã cắt đi. Có nhiều thứ sau này tôi cũng sẽ cắt đi, nhằm giảm tải cho chương trình xuống đến mức tối thiểu nhất có thể được. Nhưng an tâm là những thứ quan trọng nhất đều có đầy đủ.

Dùng từ khóa nào, class hay typename

Về cơ bản, sự khác biệt giữa chúng là không rõ ràng, cả 2 đều có cùng ý nghĩa và cùng cho kết quả như nhau, bạn muốn dùng từ khóa nào cũng được.

Nhưng có lúc bạn phải dùng từ khóa typename, ví dụ

CODE

```
template<typename T>class Thing {  
    T::SubType *ptr;  
};
```

Chúng ta muốn khai báo 1 con trỏ thuộc kiểu SubType của T, nhưng C++ sẽ hiểu là chúng ta muốn nhân giá trị SubType của kiểu T với ptr Lúc này chúng ta bắt buộc phải dùng từ khóa typename

CODE

```
template<typename T>class Thing{  
    typename T::SubType *ptr;  
};
```

Chuyên môn hóa template (template specialization)

Giả sử ta có một lớp template

```
template<class T>class pair{...}
```

Khi ta tạo một instance bằng cách khai báo cụ thể kiểu của T, ví dụ là int, tức là ta đã chuyên môn hóa (specialization) lớp template đó

```
pair<int> myobject(155,36);
```

Đôi khi ta muốn lớp template tạo ra những instance cụ thể để thực hiện những công việc cụ thể riêng đối với một loại dữ liệu cụ thể nào đó, ta dùng chuyên môn hóa cụ thể (explicit specialization)

Trong ví dụ dưới đây ta muốn riêng đối với kiểu dữ liệu cụ thể là int thì lớp template có một hàm trả về phần dư giữa hai số nguyên, còn với các kiểu dữ liệu khác thì nó trả về 0

CODE

```
template<class T>  
class pair {  
    T value1, value2;
```

```

public:
pair(T first, T second) {
    value1=first; value2=second;
}
T module() {return 0;}
};

//viết lại định nghĩa lớp chuyên môn hóa cho kiểu dữ liệu int
template<>
class pair<int> {
    int value1, value2;
public:
pair(int first, int second) {
    value1=first; value2=second;
}
int module ();
};

//hàm module dành riêng cho lớp chuyên môn hóa
template<>
int pair<int>::module() {
    return value1%value2;
}

int main() {
    pair<int> myints(100,75);
    cout<<myints.module()<<endl;
    pair<float> myfloats(100.0,75.0);
    cout<<myfloats.module()<<endl;
    return 0;
}

```

Ép kiểu dữ liệu (casting) trong C++

Trong C chúng ta ép kiểu dữ liệu như sau

```
int n=(int)45.87;
```

Trong C++ có 1 cách ép kiểu dữ liệu như sau

```
int i = static_cast<int>(45.87);
```

Cho ra kết quả như nhau (tạm chỉ cần biết thế)

Chúng ta sẽ còn quay trở lại với casting trong C++ sau

Diễn dịch đối số (argument deduction)

Xem lại hàm template dưới đây

```
template <typename T> T max(T a, T b)
```

Kiểu dữ liệu của 2 đối số (argument) a và b sẽ được quyết định bởi kiểu dữ liệu của 2 tham số (parameter) truyền vào hàm này.

Và 2 đối số này cùng là kiểu T, nên 2 tham số này phải cùng một kiểu. C++ không có tự động chuyển kiểu ở đây. Ví dụ

```
max(7, 5); //hợp lệ, T lúc này là kiểu int, 2 tham số cùng kiểu int
```

```
max(7, 5.2); //không hợp lệ, T lúc này là kiểu int (kiểu dữ liệu của tham số được truyền trước tiên, nhưng 2 tham số thì một cái kiểu int, một cái kiểu double)
```

Có 2 cách xử lý chuyện này

Cách 1: casting (ép kiểu) tham số đầu tiên

```
max(static_cast<double>(7), 5.2); //lúc này T là kiểu double, 2 đối số đều cùng kiểu double
```

Cách 2: explicit specialization (chuyên môn hóa cụ thể) cho T thành double

```
max<double> (7, 5.2);
```

Đối số của template (template argument)

template thường có các đối số là typename T (với T là kiểu dữ liệu chưa biết) Nhưng thực ra template cũng có các đối số là các kiểu dữ liệu đã biết

Đối số kiểu primitive, ví dụ kiểu int

CODE

```
template<typename T,int size>
class Array{
    T* array;
public:
    Array();
};

template<typename T,int size>Array<T,size>::Array(){
    array = new T[size];
}

int main(){
    Array<string,5> a;
    return 0;
}
```

Đối số là một lớp template khác

CODE

```
#include <iostream>
#include <string>
using namespace std;
template<typename T>
class Array
{
    T* array;
public:
    Array();
};

template<typename T>Array<T>::Array()
{
    array = new T;
}

template<typename T,typename U = Array<typename V>,int size>
class Stack
{
    U* elems;
public:
    Stack();
};

template<typename T,typename U = Array<typename V>,int size>
Stack<T,U,size>::Stack()
{
    elems = new U[size];
}

int main()
{
    Stack<string,Array<double>,5> a;
    return 0;
}
```

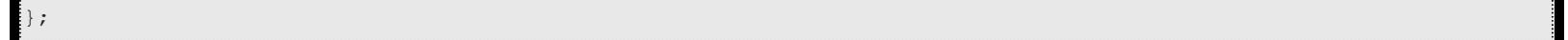
Còn mấy phần nữa, nhưng rất cao và ít dùng về sau trong lập trình game, mà chủ yếu cho lập trình bậc thấp, phần cứng, hệ điều hành, nên tôi bỏ, như thế này đủ nhức đầu và khó nhớ rồi. Các bác học xong template rồi đó, nắm rõ tất cả các kĩ thuật về template để chuẩn bị cho học STL về sau.

Làm cái bài tập chứ nhỉ. Đề đơn giản thôi: lập trình một danh sách liên kết đơn dùng template, đủ các phép thêm, xóa, sửa, truy xuất. Có sẵn cái chương trình mẫu ở dưới này. Chương trình này cực yếu, không có xóa, hủy ... Chương trình cần các bác bổ sung đó.

CODE

```
template<typename T>class Node
{
    T data;Node<T>* next;
public:
    Node<T>(T data) { (*this).data=data; (*this).next=0; }
    T getData(){return data;}
    void setData(T data){(*this).data=data;}
    Node<T>* getNext(){return next;}
    void setNext(Node<T>* next){(*this).next=next;}
};

template<typename T>class List
{
    Node<T>* front;Node<T>* rear;Node<T>* current;
public:
    List<T>(){(*this).front=(*this).rear=(*this).current=0;}
    List<T>(const List<T>& l){
        (*this).front=(*this).rear=(*this).current=0;
        Node<T>* temp=new (nothrow)Node<T>;if(temp==0) exit(1);
        temp=l.front;
        while(temp!=0){
            insertRear(temp->getData());
            temp = temp->getNext();
        }
    }
    ~List<T>(){
        Node<T>* temp=new (nothrow)Node<T>;if(temp==0) exit(1);
        while(front!=0){
            temp=front;
            front=(*front).next;
            delete temp;
        }
    }
    void insertFront(T data){
        Node<T>* temp=new (nothrow)Node<T>;if(temp==0) exit(1);
        (*temp)->setData(data);
        if(front==0) rear=temp;
        else temp->setNext(front);
        front=temp;
    }
    void insertRear(T data){
        Node<T>* temp=new (nothrow)Node<T>;if(temp==0) exit(1);
        (*temp)->setData(data);
        if(rear==0) front=temp;
        else rear->setNext(temp);
        rear=temp;
    }
    void reset(){if(front!=0) current=front;}
    void next(){if(current!=0) current = current->getNext();}
    T getCurrentData(){if(current!=0) return current->getData();}
    bool endOfList(){return(current==0);}
}
```



BÀI 7: CONST, STATIC, EXCEPTION, CASTING (BIÊN HẰNG, BIÊN TĨNH, NGOẠI LỆ, ÉP KIỂU)

CONST

`const int p` và `int const p` là như nhau

`int* const p` nghĩa là một hằng số loại con trỏ mà trỏ đến một biến số kiểu nguyên, nghĩa là bạn không thể thay đổi để con trỏ này trỏ đến một nơi khác được nữa

CODE

```
int a = 3; int b = 5;  
int* const p = &a;  
p = &b; //không hợp lệ
```

`const int* p` nghĩa là một biến số loại con trỏ mà trỏ đến một hằng số kiểu nguyên, nghĩa là bạn có thể thay đổi để con trỏ này trỏ đến một nơi khác, nhưng không thể thay đổi dữ liệu ở nơi nó trỏ đến

CODE

```
int a = 3; int b = 5;  
const int* p = &a;  
p = &b; //hợp lệ  
*p = 6; //không hợp lệ
```

`const int *& p` nghĩa là một tham chiếu mà tham chiếu tới một biến số loại con trỏ mà trỏ đến một hằng số kiểu nguyên

`int const *& p` nghĩa là một tham chiếu mà tham chiếu tới một hằng số loại con trỏ mà trỏ đến một biến số kiểu nguyên

Một biến số const nghĩa là biến đó phải được gán giá trị ngay lúc khai báo và giá trị ấy vĩnh viễn không thay đổi (cái này quá phổ biến rồi)

Một hàm số const nghĩa là hàm số đó sẽ không thay đổi giá trị của bất cứ tham số nào của nó. Một hàm số phải được khai báo const nếu nó có ít nhất một tham số const

CODE

```
class Tree{  
    int age;  
public: int getAge() const{return age;}  
    friend ostream& operator<<(ostream& os, const Tree& t)  
    {os<<t.getAge(); return os;}  
};
```

Hàm số `getAge` phải là một hàm số const để bắt kì hàm số khác dùng tham số const `Tree` có thể sử dụng hàm số `getAge`

Hàm số trả về const

Ví dụ chúng ta viết một hàm như sau

CODE

```
int& hello()  
{  
    int temp=5;  
    return temp;  
}
```

Và một ai đó có thể lợi dụng để gán giá trị cho nó, rất nguy hiểm

CODE

```
hello() = 3;
```

Nên chúng ta phải thay đổi để nó trả về const

CODE

```
const int& hello()
{
    int temp=5;
    return temp;
}
```

Bảo vệ tham số

Hẳn bạn còn nhớ ví dụ này chứ

CODE

```
void add10(int *a) {
    *a=*a+10;
}
add10(&n);
```

Ý nghĩa ở đây là, một hàm hoàn toàn có thể thay đổi giá trị của tham số nếu ta truyền vào nó địa chỉ của biến đó. Mà việc truyền địa chỉ là việc phải làm. Vì đôi khi không thể không truyền địa chỉ, ví dụ kiểu mảng

CODE

```
void dosomething(char* a){...}
char* m = "hello";
dosomething(m);
```

Khi đó ta dùng từ khóa const để ngăn việc hàm thay đổi giá trị của đối số

CODE

```
void dosomething(const char* a){...}
char* m = "hello";
dosomething(m);
```

STATIC

Một biến số const nghĩa là biến đó phải được gán giá trị ngay lúc khai báo và giá trị ấy vĩnh viễn không thay đổi

Một biến số static nghĩa là biến đó phải được gán giá trị ngay trước khi tạo một instance của một lớp và giá trị ấy được thay đổi, nhưng chỉ có duy nhất một biến static ấy tồn tại đối với tất cả instance của lớp đó

CODE

```
class MyClass
{
public:
static int i;
};

int MyClass::i; //tồn tại độc lập, không phụ thuộc instance

int main()
{
    MyClass::i=30;
    MyClass x,y;
    x.i=26;
    cout<<y.i<<endl;
}
```

Một hàm số static nghĩa là hàm số ấy có thể độc hoạt động lặp mà không cần tạo đối tượng của lớp

CODE

```
class MyClass
{
public:
static void print(char* s)
{
cout<<s<<endl;
}
};

int main()
{
    MyClass::p("Hello");
    return 0;
}
```

Ngoại lệ (Exception)

Hàm dưới đây có thể ném ra bất cứ ngoại lệ nào

CODE

```
void in(int a); //chi cần khai báo bình thường
```

Hàm dưới đây không thể ném ra bất cứ ngoại lệ nào

CODE

```
void in(int a) throw();
```

Không như Java, trong C++ bất cứ lớp nào cũng có thể là lớp ngoại lệ (exception class), và cũng không cần phải là lớp con của lớp nào hết

Chúng ta bắt ngoại lệ bằng try catch như Java

CODE

```
class DivideByZero
{
public:
    void msg()
    {
        cout<<"Can not divide by zero"<<endl;
    }
};

int divide(int a,int b) throw(DivideByZero)
{
    if(b==0) throw DivideByZero();
    return a/b;
}

int main()
{
    try
    {
        int a=divide(3,0);
    }
    catch(DivideByZero e)
    {
        e.msg();
    }
    return 0;
}
```

Khi chúng ta bắt được một ngoại lệ, chúng ta có thể ném nó ra để bắt lại một lần nữa bằng throw;

CODE

```
class DivideByZero
{
public:
    void msg()
    {
        cout<<"Can not divide by zero"<<endl;
    }
};

int divide(int a,int b) throw(DivideByZero)
{
    if(b==0) throw DivideByZero();
    return a/b;
}

int main()
{
    try
    {
        try
        {
            int a=divide(3,0);
        }
        catch(DivideByZero e)
        {
            e.msg();throw; //ném ra để bắt lại lần hai
        }
    }
    catch(DivideByZero)
    {
        cout<<"Stopped! That's enough"<<endl;
    }
    return 0;
}
```

Chúng ta có thể ném ra và bắt lại bất cứ cái gì chứ không phải chỉ một lớp

CODE

```
int main()
{
    int s[5];
    try
    {
        for(int i=0;i<6;i++)
        {
            if(i>=5) throw "Out of range";
            s[i] = i;
        }
    }
    catch(char* c)
    {
        cout<<c<<endl;
    }
    return 0;
}
```

Chúng ta có thể ném ra nhiều thứ bằng một chỉ thị try và bắt lại tất cả mọi thứ, mỗi thứ một chỉ thị catch khác nhau

CODE

```
int main()
{
    try
    {
        int size;cin>>size;
        if(size>10) throw "So big";
        else throw size;
    }
    catch(int n)
    {
        cout<<n<<" is not here"<<endl;
    }
    catch(char* c)
    {
        cout<<c<<endl;
    }
    return 0;
}
```

Nếu có ngoại lệ nào không bị bắt, hàm đặc biệt void terminate() sẽ được gọi ngay lập tức để chấm dứt chương trình
Chúng ta có thể bắt tất cả các loại ngoại lệ để xử lý chỉ với một chỉ thị catch(...)

CODE

```
int main()
{
    try
    {
        int size;cin>>size;
        if(size>10) throw "So big";
        else throw size;
    }
    catch(...)
    {
        cout<<"throw exception"<<endl;
    }
    return 0;
}
```

Ép kiểu (Casting)

Ép kiểu theo cách của C

int n=(int)45.87;

static_cast (ép kiểu tĩnh)

Kiểm tra lúc biên dịch (compile time)

int n=static_cast<int>(45.87);

int n=static_cast<int>("hello"); //lúc biên dịch sẽ phát sinh lỗi

dynamic_cast (ép kiểu động)

Kiểm tra lúc thực thi (runtime) Lớp dùng với dynamic_cast phải có hàm ảo.

downcast

Ép kiểu các con trỏ và các đối tượng từ kiểu lớp cha thành kiểu lớp con, được gọi là downcast. Đây là một kiểu casting quan trọng,

dùng nhiều về sau trong RTTI. Có 2 cách thực hiện downcast

bằng static_cast

CODE

```
class Base{};  
class Derived : public Base{};  
Base* b=new Derived;  
Derived* d = static_cast<Derived*>(b);
```

bằng dynamic_cast

CODE

```
class Base{virtual void func(){};}  
class Derived : public Base{};  
Base* b = new Derived;  
Derived* d = dynamic_cast<Derived*>(b);
```

upcast

Ép kiểu các con trỏ và các đối tượng từ kiểu lớp con thành kiểu lớp cha, được gọi là upcast. upcast hiếm dùng hơn downcast. Có 2 cách thực hiện upcast

bằng static_cast

CODE

```
class Base{};  
class Derived : public Base{};  
Derived* d = new Derived;  
Base* b = static_cast<Base*>(d);
```

bằng dynamic_cast

CODE

```
class Base{virtual void func(){};}  
class Derived : public Base{};  
Derived* d = new Derived;  
Base* b = dynamic_cast<Base*>(d);
```

const_cast (ép kiểu hằng)

const_cast dùng chủ yếu với các con trỏ

const_cast dùng để thay đổi một biến số thành một hằng số (thêm từ khóa const vào)

CODE

```
int a=3;  
int* b=&a;  
*b = 8;//hợp lệ  
const int* c=const_cast<int*>(b);  
*c=5; //không hợp lệ  
cout<<*c;  
cout<<a;
```

const_cast dùng để thay đổi một hằng số thành một biến số (gỡ từ khóa const ra)

CODE

```
const int a=3;  
const int* b=&a;  
*b = 8; //không hợp lệ  
int* c=const_cast<int*>(b);  
*c=5; //hợp lệ  
cout<<*c;  
cout<<a;
```

reinterpret_cast (ép kiểu thông dịch lại)

reinterpret_cast sẽ ép kiểu bất cứ con trỏ hay đối tượng nào mà không hề có sự kiểm tra nào cả. Không khuyến khích dùng và bây giờ ta cũng chưa phải dùng, sẽ học sau.

BÀI 8: STL - SEQUENTIAL CONTAINER

Yêu cầu: học xong môn cấp trúc dữ liệu và giải thuật cơ bản hoặc tương đương để có kiến thức cơ bản về các cấu trúc dữ liệu động như danh sách liên kết (linked list), hàng đợi (queue), ngăn xếp (stack), tập hợp (set), ánh xạ (map) và các giải thuật tìm kiếm, sắp xếp cơ bản.

STL (Standard Template Library) là một bộ thư viện vô cùng hữu dụng của C++ dùng để làm việc với các cấu trúc dữ liệu phổ biến như danh sách, hàng đợi, ngăn xếp và các phép toán chủ yếu với các cấu trúc dữ liệu này như tìm kiếm, sắp xếp, truy xuất, thêm, xóa, sửa.

STL bao gồm

*Các container (các bộ lưu trữ dữ liệu) là các cấu trúc dữ liệu phổ biến đã template hóa dùng để lưu trữ các kiểu dữ liệu khác nhau. Các container chia làm 2 loại:

-sequential container (các bộ lưu trữ dữ liệu tuần tự) bao gồm list, vector và deque

-associative container (các bộ lưu trữ dữ liệu liên kết) bao gồm map, multimap, set và multiset

*Các iterator (các con trỏ dữ liệu) là các con trỏ để trỏ đến các phần tử trong các bộ lưu trữ

*Các algorithm (các thuật toán lưu trữ dữ liệu) là các hàm phổ biến để làm việc với các bộ lưu trữ như thêm, xóa, sửa, truy xuất, tìm kiếm, sắp xếp

*Các function object (các đối tượng hàm) là các hàm và phép toán phổ biến để làm việc với các phần tử được lưu trữ cũng như các bộ lưu trữ và các thuật toán lưu trữ như cộng, trừ, nhân, chia, so sánh

*Các adapter (các bộ tương thích) Các adapter chia làm 3 loại

-container adapter (các bộ tương thích lưu trữ) bao gồm stack, queue và priority_queue

-iterator adapter (các bộ tương thích con trỏ)

-function adapter (các bộ tương thích hàm)

Trước tiên ta học về các container.

LIST

CODE

```
#include <list>
```

list rong STL là danh sách liên kết đôi, không hỗ trợ random access (truy xuất dữ liệu bất kì) Nghĩa là nếu bạn muốn truy xuất một phần tử bất kì trong list thì bạn phải truy xuất từ phần tử đầu tiên hoặc phần tử cuối cùng của list rồi truy xuất dần đến phần tử đó

Khởi tạo sao chép

list có thể khởi tạo sao chép từ mảng, từ list khác hoặc từ các container khác

CODE

```
int a[10];
list<int> list1(a+2,a+7);
list<int> list2(list1.begin()++,--list1.end());
```

Các hàm thường dùng của list

CODE

```
void push_front(T element): đưa một phần tử vào đầu list
void push_end(T element): đưa một phần tử vào cuối list
void pop_front(): gỡ phần tử đầu list ra
void pop_end(): gỡ phần tử cuối list ra
iterator begin(): trả về iterator trỏ đến phần tử đầu list
iterator end(): trả về iterator trỏ đến phần tử cuối list
```

Ví dụ dưới chúng ta tạo một list, đưa phần tử vào và truy xuất phần tử

CODE

```
list<string> list1;
list1.push_back("Zebra");list1.push_back("Penguin");list1.push_front("Lion");
list<string>::iterator i;
for(i=list1.begin();i!=list1.end();++i) cout<<*i<<endl;
```

Toán tử * được dùng để lấy giá trị của iterator

*i: trả về giá trị được trả tới bởi iterator i, ở đây là các phần tử của list1

Nếu chúng ta muốn một list chứa nhiều list, ta chỉ cần khai báo

CODE

```
list<list<string> > listOfList;
```

Nếu list có khai báo const, chúng ta phải dùng const_iterator thay vì iterator

CODE

```
const list<string> list1;
list<string>::const_iterator i = list1.begin();
```

Các hàm thường dùng khác của list

CODE

```
int n=list1.size(); //trả về số phần tử của list
bool b=list1.empty(); //kiểm tra list, nếu rỗng (không có phần tử) thì trả về true, ngược lại trả về false
list1.insert(list1.begin(),"Seadog"); //chèn phần tử "Seagon" vào vị trí đầu list
list1.insert(++list1.begin(),2,"Seadog"); //chèn phần tử "Seagon" vào một vị trí cụ thể
list1.erase(list1.begin()); //xóa một phần tử ở một vị trí cụ thể
list1.erase(++list1.begin(),3); //xóa 3 phần tử bắt đầu từ một vị trí cụ thể
list1.clear(); //xóa tất cả các phần tử
list1.remove("Zebra"); //tim kiêm và xóa phần tử "Zebra"
list1.sort(); //sắp xếp tăng dần (ascending)
list1.reverse(); //sắp xếp giảm dần (descending)
list1.resize(int); //thiết lập số phần tử mới của list
iterator i=list1.find(++list1.begin(),--list1.end(),"Penguin"); //tìm kiếm phần tử "Penguin", bắt đầu ở một vị trí cụ thể kết thúc ở một vị trí cụ thể khác, trả về iterator trỏ đến phần tử này. Nếu không tìm thấy, hàm này trả về vị trí kết thúc, ở đây là --list1.end()
```

Các hàm với hai list

CODE

```
list<string> list1;
list<string> list2;
list1.splice(--list1.end(),list2,list2.begin());
//splice(cut và paste) một phần tử từ vị trí list2.begin() của list2 đến vị trí --list1.end() của list1
list1.splice(--list1.end(),list2);
//splice(cut và paste) tất cả phần tử của list2 đến vị trí --list1.end() của list1
list1.merge(list2); //merge 2 list, nghĩa là list1 = list1 + list2;
list2.swap(list1); //swap 2 list, nghĩa là temp = list2; list2 = list1; list1 = temp;
```

VECTOR

CODE

```
#include <vector>
```

vector giống list ngoại trừ

- cho phép random access, với operator[], nghĩa là v[5], v[6], etc như mảng
- được tối ưu hóa với các phép toán ở phía đuôi (rear operations)
- không có sẵn các hàm push_front, pop_front, splice, sort, merge, reserve

CODE

```
vector<int> v;  
v.push_back(5);  
cout<<v[0];
```

DEQUE

CODE

```
#include <deque>
```

deque giống list ngoại trừ

- cho phép random access, với operator[], nghĩa là d[5], d[6], etc như mảng
- được tối ưu hóa với các phép toán ở phía đầu (front operations)
- không có sẵn các hàm splice, sort, merge, reserve

CODE

```
deque<int> d;  
d.push_front(5);  
cout<<d[0];
```

ALGORITHM

CODE

```
#include <algorithm>
```

Do vector và deque không có sẵn các hàm splice, sort, merge, reserve nên nếu ta muốn sử dụng các chức năng này với vector và deque ta phải dùng các hàm trong thư viện algorithm

Sắp xếp

CODE

```
#include <algorithm>  
vector<int> v;  
sort(v.begin(), v.end()); //sắp xếp tăng dần (ascending)  
reverse(v.begin(), v.end()); //sắp xếp giảm dần (descending)
```

Sao chép

CODE

```
int a[]={1,2,3,4};  
vector<int> v;  
v.resize(3);  
copy (a,a+3,v.begin()); //sao chép 3 phần tử từ mảng a vào vector v
```

Merge và swap

CODE

```
vector<int> v, v2;
merge(v1.begin(), v1.begin() + 5, ++v2.begin(), v2.end(), v1.begin()); // hợp 2 phần dữ liệu, phần môt từ v1.begin()
đến v1.begin() + 5, phần hai từ ++v2.begin() đến v2.end(), sau đó chép tất cả vào v1 bắt đầu từ v1.begin()
swap(v1, v2);
```

CÁC PHẦN TỬ LÀ CON TRỎ

Giả sử ta có lớp Person với hàm khởi tạo Person(char* name)

CODE

Để đưa một phần tử Person vào vector:

```
vector<Person> v; v.push_back(Person("C"));
```

Để đưa một phần tử con trỏ Person vào vector, ta phải dùng new để cấp phát bộ nhớ:

```
vector<Person*> vp; vp.push_back(new Person("M"));
```

Để truy xuất phần tử

```
vector<Person>::iterator i;
```

```
for(i=v.begin(); i!=v.end(); ++i) cout << *i << endl;
```

Để truy xuất phần tử con trỏ

```
vector<Person*>::iterator ip;
```

```
for(ip=vp.begin(); ip!=vp.end(); ++ip) cout << **ip << endl;
```

*i: trả về giá trị được trả tới bởi iterator i, ở đây là các phần tử của vector v

vector<Person>::iterator i; (*i) trả về Person

**ip: trả về giá trị của các con trỏ được trả tới bởi iterator i, ở đây là các phần tử con trỏ của vector vp

vector<Person*>::iterator ip; (**ip) trả về Person*

nhiều vậy (**ip) trả về *(Person*)

BÀI 9:FUNCTION OBJECT (ĐỐI TƯỢNG HÀM)

Function object

Một function object (đối tượng hàm) là một object (đối tượng) được sử dụng như một function (hàm). Một function object là một instance của một lớp mà lớp đó phải có ít nhất một hàm thửa

- quyền truy xuất phải là public
- phải là một hàm thành viên, không phải là một hàm friend
- không phải là một hàm static
- có khai báo operator()

Ví dụ ta viết một hàm bình thường như sau

CODE

```
void iprintf(int i) const
{
    cout<<i<<endl;
}
```

Bây giờ ta sẽ viết một lớp như sau

CODE

```
class iprintf
{
public:
    void operator()(int i) const
    {
        cout<<i<<endl;
    }
};
```

Instance của lớp này là một object được gọi là function object, là một object được sử dụng như một function. Sử dụng như thế nào ?

CODE

```
iprintf x;
x(5);
```

hoặc

CODE

```
iprintf()(5);
```

Khi ta gọi iprintf()(5) nghĩa là chúng ta đang gọi đến operator() của lớp iprintf function object còn được gọi là một functor hay một functional. Từ đây khi đề cập đến function object sẽ dùng functor.

Ví dụ dưới đây là một lớp có nhiều hơn một operator()

CODE

```
class iprintf
{
int i;
public:iprintf(int i):i(i){}
public:
    void operator()() const
    {
        cout<<i<<endl;
    }
    void operator()(int i) const
```

```

{
    cout<<"Integer:"<<i<<endl;
}
void operator() (float f) const
{
    cout<<"Float:"<<f<<endl;
}
};

int main(int argc,char** argv)
{
    iprintf x(20);
    x();
    x(5);      //giả sử không có operator()(int i), câu này sẽ gọi operator()(float f)
    x(2.3);    //giả sử không có operator()(float f), câu này sẽ gọi operator()(int i) với i = 2
    x("something"); //lỗi
    return 0;
}

```

Tương tự thay vì iprintf(5); x(7); chúng ta cũng có thể gọi iprintf(5)(7);

Có một điều chú ý ở ví dụ trên là nếu cùng tồn tại operator()(int i) và operator()(float f) thì câu lệnh x(2.3); sẽ báo lỗi ambiguous (nhập nhằng) giữa hai hàm. Có một cách đơn giản là viết lại thành x((float)2.3);

Predicate

Predicate có một định nghĩa khác phức tạp hơn. Ở đây chỉ nêu điều cần thiết nhất có liên can đến chương trình.

Một predicate được đề cập đến ở đây là một function hoặc một functor có điều kiện giá trị trả về đúng hoặc sai hoặc một giá trị có thể chuyển kiểu thành đúng hoặc sai. Trong C/C++, đúng có nghĩa là khác 0 và sai có nghĩa là bằng 0

Ví dụ hàm sau đây là một predicate

CODE

```

double truefalse(double n)
{
    return n;
}

```

Một số hàm thường dùng trong algorithm

Hàm find

CODE

```

vector<int> v;
v.push_back(4);v.push_back(3);v.push_back(2);
vector<int>::iterator i = find (v.begin(),v.end(),3);
if(i!=v.end()) cout<<*i;

```

Hàm find tìm từ phần tử v.begin() đến phần tử v.end() và trả về iterator trả đến phần tử có giá trị là 3, nếu không tìm thấy sẽ trả về v.end()

Hàm find_if

CODE

```

int IsOdd(int n)
{
    return n%2;
}
int main()
{
    list<int> l;
    l.push_back(4);l.push_back(5);l.push_back(2);

```

```

list<int>::iterator i=find_if(l.begin(),l.end(),IsOdd);
if(i!=l.end()) cout<<*i;
}

```

Hàm `find_if` tìm từ phần tử `v.begin()` đến phần tử `v.end()` và trả về iterator trả đến phần tử có giá trị thỏa predicate, nếu không tìm thấy sẽ trả về `v.end()`

Lưu ý, lúc này `IsOdd` đóng vai trò là một predicate, xác định xem phần tử của list có là số lẻ hay không (tức là khi đưa vào làm tham số của hàm `IsOdd` có trả về một số khác 0 hay không)

Chúng ta viết lại predicate này bằng cách dùng functor

CODE

```

class IsOdd
{
public:
    bool operator()(int n) const
    {
        return n%2;
    }
};

int main()
{
    list<int> l;
    l.push_back(4);l.push_back(5);l.push_back(2);
    list<int>::iterator i=find_if(l.begin(),l.end(),IsOdd());
    if(i!=l.end()) cout<<*i;
}

```

Hàm `equal`

Ở trên chúng ta mới xét các ví dụ với predicate có một đối số, ta xét một hàm khác của algorithm dùng predicate nhiều hơn một đối số, hàm `equal`

CODE

```

class compare
{
public:
    bool operator()(int i,int j) const
    {
        return i==j;
    }
};

int main()
{
    compare c;
    int a[] = {1, 2, 3, 4, 5};
    list<int> l(a,a+3);      //list ít phần tử hơn mảng
    cout<<equal(l.begin(),l.end(),a,c)<<endl;
    a[2] = 6;
    cout<<equal(l.begin(),l.end(),a,c)<<endl;
    return 0;
}

```

Hàm `equal` so sánh từng phần tử của list từ phần tử `l.begin()` đến phần tử `l.end()` với từng phần tử tương ứng của mảng `a` sao cho mỗi cặp phần tử đều thỏa predicate là `c`, trả về là `true` nếu từng cặp phần tử so sánh với nhau đều cho giá trị `true` (không cần quan tâm đến số lượng phần tử có tương ứng không) Nhưng chỉ cần một cặp trả về `false` thì hàm sẽ trả về `false`

Hàm search

Hàm search tìm vị trí của một chuỗi con trong một chuỗi lớn hơn, nếu tìm thấy thì trả về iterator trỏ đến vị trí của chuỗi con đó trong chuỗi lớn. Hàm này có hai "phiên bản"

CODE

```
int a[] = {3,4,5};
vector<int> v;
for(int i = 0;i<10;i++) v.push_back(i);
vector<int>::iterator iv = search(v.begin(),--v.end(),a,a+2);
if(iv!=v.end()) cout<<iv-v.begin();
```

Phiên bản thứ nhất tìm vị trí của chuỗi con từ phần tử có vị trí $a+0$ đến phần tử có vị trí $a+2$ trong chuỗi lớn hơn từ phần tử có vị trí $v.begin()$ đến phần tử có vị trí $--v.end()$. Nếu không tìm thấy thì trả về vị trí $--v.end()$.

Trong đoạn mã trên có một điều đáng chú ý là $iv-v.begin()$. Lưu ý là hàm search trả về một iterator, iterator này là $iv = v.begin() +$ vị trí của chuỗi con. Do đó đơn giản vị trí của chuỗi con = $iv-v.begin()$

CODE

```
class compare
{
public:
    bool operator()(int i,int j) const
    {
        return i==j+1;
    }
};

int main()
{
    int a[] = {3,4,5};
    vector<int> v;
    for(int i = 0;i<10;i++) v.push_back(i);
    vector<int>::iterator iv = search(v.begin(),v.end(),a,a+2,compare());
    if(iv!=v.end()) cout<<iv-v.begin();
    return 0;
}
```

Phiên bản thứ hai sẽ phải cần đến một predicate có hai đối số giống như hàm equal. Phiên bản thứ hai tìm vị trí của chuỗi con từ phần tử có vị trí $a+0$ đến phần tử có vị trí $a+2$ trong chuỗi lớn hơn từ phần tử có vị trí $v.begin()$ đến phần tử có vị trí $--v.end()$ sao cho từng cặp phần tử thỏa compare() (ở đây là điều kiện phần tử của v = phần tử của $a + 1$). Nếu không tìm thấy thì trả về vị trí $--v.end()$.

Tương tự như hàm search này là hàm find_end, nhưng thay vì trả về vị trí đầu tiên của chuỗi con xuất hiện trong chuỗi lớn thì lại trả về vị trí cuối cùng của chuỗi con xuất hiện trong chuỗi lớn.

Hàm for_each

Hàm for_each dùng để duyệt từng phần tử trong một chuỗi các phần tử cho trước

Dùng for_each để in ra các phần tử, ví dụ

CODE

```
void display(const string& s){cout<<s<<endl;}
list<string> l;l.push_back("hello");l.push_back("world");
for_each(l.begin(),l.end(),display);
```

Tương tự dùng với một functor

CODE

```
template<typename T>class Output
```

```

{
public:
    void operator() (const T& t) {cout<<t<<endl;}
};

int main(int argc, char* argv[])
{
    list<string> l;l.push_back("hello");l.push_back("world");
    for_each(l.begin(),l.end(),Output<string>());
}

```

Hàm count

Hàm count dùng để đếm số lượng phần tử trong một chuỗi các phần tử cho trước

CODE

```

list<string> l;l.push_back("hello");l.push_back("world");
cout<<(int)count(l.begin(),l.end(),"hello")<<endl;

```

Hàm count_if

Hàm count_if dùng để đếm số lượng phần tử thỏa một điều kiện nào đó trong một chuỗi các phần tử cho trước, hàm cần một predicate một đối số

CODE

```

class IsOdd
{
public:
bool operator()(int n) const{return (n%2)==1;}
};

int main(int argc, char* argv[])
{
    list<int> l;for(int i=0;i<10;i++) l.push_back(i);
    cout<<(int)count_if(l.begin(),l.end(),IsOdd())<<endl;
}

```

Toàn bài chúng ta học về predicate và thư viện algorithm, còn một số hàm sẽ học sau. Có một điều đáng lưu ý đó là predicate một đối số và hai đối số. Số lượng đối số của predicate được gọi là **hạng (arity)** của predicate. Biết sơ điều này và chuẩn bị cho bài tiếp theo.

BÀI 10: THƯ VIỆN FUNCTIONAL

CODE

```
#include <functional>
```

Hạng của một predicate

Có nhiều sự mập mờ do từ đồng nghĩa giữa các khái niệm toán học trong cả hai ngôn ngữ tiếng Việt và tiếng Anh, do đó định nghĩa sau chỉ ở mức cõi gắng chính xác nhất có thể được:

Số toán tử (operand) của một phép toán (operator), tương ứng là số đối số (argument) của một hàm (function), được gọi là **hạng (arity)** của phép toán hay hàm đó

Tương tự, số toán tử (operand) của một biểu thức (expression), tương ứng là số đối số (argument) của một đối tượng hàm (functor), được gọi là **hạng (arity)** của biểu thức hay đối tượng hàm đó

Ví dụ

Unary (đơn nguyên, đơn phân, một toán hạng, một ngôi)

$n!$ (giai thừa của n) là một unary operator

$n!$ là một unary expression, chỉ bao gồm một unary operator

`int gaiithua(int n)` là một unary function

một object của class `gaiithua{int operator()(int n)...}` là một unary functor

Binary (nhị nguyên, nhị phân, hai toán hạng, hai ngôi)

$a + b$ là một binary expression, chỉ bao gồm một binary operator

`int addition(int a,int b)` là một binary function

một object của class `addition{int operator()(int a,int b)...}` là một binary functor

Ternary (tam nguyên, tam phân, ba toán hạng, ba ngôi)

$b * b - 4 * a * c$ là một ternary expression, bao gồm một unary operator và ba binary operator

`double delta(double a, double b,double c)` là một ternary function

một object của class `delta{ double operator()(double a, double b,double c)...}` là một ternary functor

n-ary (đa nguyên, đa phân, nhiều toán hạng, nhiều ngôi)

Tương tự như trên, ngoài ra còn có nhiều từ gốc Latin khác như quaternary (bốn toán hạng) quinary (năm toán hạng) ... gọi chung là **nhiều toán hạng**.

Hạng của predicate tức là **hạng** của function hay functor mà đóng vai trò predicate. Như ví dụ ở trên, `addition` là một binary predicate, `delta` là một ternary predicate

Cấu trúc unary_function trong thư viện functional

Trong thư viện functional đã định nghĩa sẵn cấu trúc `unary_function`

CODE

```
template<class Arg, class Result>
struct unary_function
{
    typedef Arg argument_type;
    typedef Result result_type;
};
```

`unary_function` là cấu trúc định nghĩa sẵn cho tất cả unary function và unary functor với `Arg` là kiểu dữ liệu của đối số và `Result` là kiểu trả về của hàm có `operator()`

Chúng ta viết lại lớp `IsOdd`, định nghĩa nó là một `unary_function`

CODE

```
class IsOdd:public unary_function<int,bool>
{
public:
    bool operator()(int n) const
    {
        return n%2;
    }
};
```

Cấu trúc binary_function trong thư viện functional

Tương tự, trong thư viện functional đã định nghĩa sẵn cấu trúc binary_function

CODE

```
template<class Arg1, class Arg2, class Result>
struct binary_function
{
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

binary_function là cấu trúc định nghĩa sẵn cho tất cả binary function và binary functor với Arg1 là kiểu dữ liệu của đối số thứ nhất và Arg2 là kiểu dữ liệu của đối số thứ hai và Result là kiểu trả về của hàm có operator()

Chúng ta viết lại lớp compare, định nghĩa nó là một binary_function

CODE

```
class compare:public binary_function<int,int,bool>
{
public:
    bool operator()(int i,int j) const
    {
        return i==j;
    }
};
```

Tương tự chúng ta có thể tự viết các cấu trúc ternary_function, quaternary_function, vân vân nếu muốn. Ví dụ dưới đây là một cấu trúc ternary_function tự viết và một lớp được định nghĩa là một ternary_function

CODE

```
template<class Arg1, class Arg2, class Arg3, class Result>
struct ternary_function
{
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Arg3 third_argument_type;
    typedef Result result_type;
};

class multiply:public ternary_function<int,float,long,double>
{
public:
    double operator()(int i,float f,long l) const
    {
        return i*f*l;
    }
};
```

Ràng buộc (bind) toán hạng cho predicate

Có nhiều hàm chỉ chấp nhận một đối số, nhưng chúng ta lại cần chuyển vào cho nó các predicate là binary predicate như binary function hay binary functor. Trong trường hợp đó chúng ta cần ràng buộc toán hạng cho binary predicate đó để nó trở thành một unary predicate

Ví dụ chúng ta cần dùng hàm find_if để tìm các phần tử trong một vector thỏa một binary predicate, nhưng find_if lại chỉ chấp nhận unary predicate, khi đó chúng ta cần ràng buộc toán hạng cho binary predicate đó để nó trở thành một unary predicate binary predicate muốn được ràng buộc toán hạng phải được định nghĩa là một binary_function

Hàm bind1st

Hàm bind1st ràng buộc toán hạng thứ nhất của một binary predicate với một giá trị cho trước để nó trở thành một unary predicate

với đối số còn lại của binary predicate ban đầu trở thành đối số của unary predicate kết quả

CODE

```
class compare:public binary_function<int,int,bool>
{
public:
    bool operator()(int i,int j) const
    {
        return i+1==j;
    }
};

int main()
{
    vector<int> v;
    v.push_back(4);v.push_back(0);v.push_back(1);
    vector<int>::iterator i=find_if(v.begin(),v.end(),bind1st(compare(),0));
    if(i!=v.end()) cout<<i-v.begin();
    return 0;
}
```

Trong ví dụ trên, đối số thứ nhất của compare() đã được ràng buộc bằng 0, compare() trở thành một predicate chỉ có một đối số là đối số còn lại của compare() ban đầu, và find_if chỉ việc truyền tham số là iterator trả đến các phần tử của v vào đối số này, quá trình chạy vòng lặp diễn ra giống như sau

compare()(0,4) //phép so sánh $0 + 1 == 4$ trả về false

compare()(0,0) //phép so sánh $0 + 1 == 0$ trả về false

compare()(0,1) //phép so sánh $0 + 1 == 1$ trả về true

Hàm bind2nd

Hàm bind2nd ràng buộc toán hạng thứ hai của một binary predicate với một giá trị cho trước để nó trở thành một unary predicate với đối số còn lại của binary predicate ban đầu trở thành đối số của unary predicate kết quả

CODE

```
class compare:public binary_function<int,int,bool>
{
public:
    bool operator()(int i,int j) const
    {
        return i+1==j;
    }
};

int main()
{
    vector<int> v;
    v.push_back(4);v.push_back(0);v.push_back(1);
    vector<int>::iterator i=find_if(v.begin(),v.end(),bind2nd(compare(),1));
    if(i!=v.end()) cout<<i-v.begin();
    return 0;
}
```

Trong ví dụ trên, đối số thứ hai của compare() đã được ràng buộc bằng 1, compare() trở thành một predicate chỉ có một đối số là đối số còn lại của compare() ban đầu, và find_if chỉ việc truyền tham số là iterator trả đến các phần tử của v vào đối số này, quá trình chạy vòng lặp diễn ra giống như sau

compare()(4,1) //phép so sánh $4 + 1 == 1$ trả về false

compare()(0,1) //phép so sánh $0 + 1 == 1$ trả về true

compare()(1,1) //phép so sánh $1 + 1 == 1$ trả về false (thực ra không có phép so sánh này, hàm đã trả về iterator rồi)

Một số hàm thường dùng của thư viện algorithm

Hàm sort

CODE

```
vector<int> v;
```

Hàm này có 2 phiên bản

Sắp xếp lại một chuỗi phần tử theo thứ tự tăng dần (ascending)

CODE

```
sort (v.begin(),v.end());
```

Sắp xếp lại một chuỗi phần tử thỏa một binary predicate

CODE

```
template<typename T>class Bigger{
public:
    bool operator() (const T& t1,const T& t2){return t1>t2;}
};

template<typename T>class Output{
public:
    void operator() (const T& t){cout<<t<<endl;}
};

int main(int argc, char* argv[]){
    vector<int> v;for(int i=0;i<10;i++) v.push_back(i);
    sort(v.begin(),v.end(),Bigger<int>());
    for_each(v.begin(),v.end(),Output<int>());
    return 0;
}
```

Hàm transform

CODE

```
vector<int> v1;
for(int i=0;i<6;i++) v1.push_back(i);
```

Hàm này có hai phiên bản:

CODE

```
int increase(int i){return ++i;}
vector<int> v2;
v2.resize(v1.size());
transform(v1.begin(),v1.end(),v2.begin(),increase);
```

Phiên bản thứ nhất sẽ lấy tất cả phần tử từ v1.begin() đến v1.end(), transform chúng bằng hàm increase, sau đó chép giá trị đã transform vào bắt đầu từ v2.begin()

CODE

```
int addition(int i,int j){return i+j;}
vector<int> v3;
v3.resize(v1.size());
transform(v1.begin(),v1.end(),v2.begin(),v3.begin(),addition);
```

Phiên bản thứ hai sẽ lấy tất cả phần tử từ v1.begin() đến v1.end(), transform chúng bằng hàm addition với đối số thứ hai là tất cả phần tử từ v2.begin(), sau đó chép giá trị đã transform vào bắt đầu từ v3.begin()

Một số hàm thường dùng của thư viện functional

Các hàm toán học cơ bản

Bao gồm cộng (plus) trừ (minus) nhân (multiplies) chia (divides) chia lấy dư (modulus) đổi dấu (negate) Các hàm này rất đơn giản, ví dụ negate

CODE

```
int a[]={1,-2,3};  
transform(a,a+3,a,negate<int>());  
for_each(a,a+3,Output<int>());
```

Ví dụ plus

CODE

```
int a[]={1,2,3,4,5};  
int b[]={6,7};  
int c[5];  
transform(a,a+5,b,c,plus<int>());
```

Ở bài trên có một điều đáng chú ý, bạn tự tìm xem

Ví dụ modulus

CODE

```
int a[]={1,2,3,4,5};  
int b[]={2,2,2,2,2};  
int c[5];  
transform(a,a+5,b,c,modulus<int>());
```

Cái ví dụ hàm modulus này hơi ... kì kì. Modulus là một binary function, giả sử bây giờ chúng ta muốn các phần tử của a luôn modulus cho 2 thì làm thế nào ? Phải ràng buộc toán hạng cho modulus để nó trở thành một unary function thôi.

CODE

```
int a[]={1,2,3,4,5};  
int b[5];  
transform(a,a+5,b,bind2nd(modulus<int>(),2));
```

Các hàm so sánh

Bao gồm equal_to (==) not_equal_to (!=) greater (>) less (<) greater_equal (>=) less_equal(<=) logical_and (&&) logical_or (||) logical_not (!)

Các hàm này cách dùng y như nhau, lấy một ví dụ hàm greater

CODE

```
int a[]={3,2,5,1,4};  
sort(a,a+5,greater<int>());  
for_each(a,a+5,Output<int>());
```

Giả sử ta muốn dùng hàm count_if với hàm greater, trả về số phần tử nhỏ hơn 3 chẳng hạn. Ta làm thế nào ? greater là một binary function, lại phải ràng buộc toán hạng cho nó với đối số thứ nhất là 3 rồi

CODE

```
int a[]={3,2,5,1,4};  
cout<<(int)count_if(a,a+5,bind1st(greater<int>(),3));  
for_each(a,a+5,Output<int>());
```

BÀI 11 – ASSOCIATIVE CONTAINER (CÁC BỘ LƯU TRỮ LIÊN KẾT)

Bao gồm map (ánh xạ) multimap (đa ánh xạ) set (tập hợp) multiset (đa tập hợp)

Sự khác nhau giữa các associative container và sequential container ở một điểm:

- các sequential container lưu trữ các phần tử (gọi là các value) và các value này được truy xuất tuân tự theo vị trí của chúng trong bộ lưu trữ
- các associative container lưu trữ các phần tử (gọi là các value) và các khóa (gọi là các key) liên kết với các value và các value này được truy xuất theo các key mà chúng có liên kết

Map

CODE

```
#include<map>
map<char*,int> ánh xạ từ một char* đến một int
map<char*,int> mapInt;
mapInt["one"] = 1;
cout<<mapInt["one"];
```

Ánh xạ từ một key đến một value. Ví dụ sau key là lớp string, value là lớp Person

CODE

```
class Person{
public:string name;
    Person(string name):name(name){}
    friend ostream& operator<<(ostream& os,const Person& p)
    {
        os<<p.name;return os;
    }
};

typedef map<string,Person> MP;
typedef MP::iterator MPI;
typedef MP::const_iterator MPCI;
typedef MP::value_type MPVT;
void display(const MP& mp)
{
    for(MPCI i=mp.begin();i!=mp.end();++i) cout<<(*i).first<<" "<<(*i).second<<endl;
}
int main()
{
    MP mapPerson;Person p("Viet");
    mapPerson.insert(MPVT("one",p));
    display(mapPerson);
    return 0;
}
```

Giải thích:

value_type dùng để khởi tạo một cặp (key,value) cho một ánh xạ. Còn một cách khác là dùng lớp pair của thư viện utility. Có 2 cách

Cách một là khởi tạo luôn một instance của lớp pair

CODE

```
#include<utility>
mapPerson.insert(pair<string,Person>("two",Person("Nam")));
```

Cách hai là dùng hàm make_pair

CODE

```
pair<string,Person> pr = make_pair(string("two"),Person("Nam"));
```

```
mapPerson.insert(pr);
```

value_type thực chất cũng là một pair

Comparator

Một functor dùng để so sánh, sắp xếp, etc các phần tử trong một map gọi là một comparator. Khi đó map thay vì có 2 argument như map<key K,value V> thì có 3 argument là map<key K,value V,comparator C>

Dùng comparator để so sánh

CODE

```
class comparePerson
{
public:
    bool operator() (Person p1, Person p2) {
        return p1.name.compare(p2.name);
    }
};

typedef map<Person,int, comparePerson> MAP;
MAP pMap;
Person p=new Person(...);
MAP::iterator i=pMap.find(d);
if(i==pMap.end()) pMap.insert(MAP::value_type(d,1));
```

Dùng comparator để sắp xếp

CODE

```
class comparePerson
{
public:
    bool operator() (const Person& p1, const Person& p2)
    {
        return p1.name.compare(p2.name);
    }
};

typedef map<string,Person,comparePerson> MP;
MP mapPerson;Person p("Viet");
mapPerson.insert(pair<string,Person>("one",Person("Nam")));
mapPerson.insert(pair<string,Person>("two",Person("Viet")));
mapPerson.insert(pair<string,Person>("three",Person("An")));
display(mapPerson);
```

Bạn lưu ý là tất cả các associative container đều có xây dựng sẵn comparator mặc định là less<key> (trong thư viện functional)

Nghĩa là khi bạn khai báo

CODE

```
map<char*,int> mapInt;
```

thực ra là

CODE

```
map<char*,int,less<char*> > mapInt;
```

Ví dụ

CODE

```
typedef map<char*,int> MI;
typedef map<char*,int>::iterator MII;
MI m;m["c"] = 1;m["b"] = 2;m["a"] = 3;
```

```
for(MII i=m.begin();i!=m.end();++i)
    cout<<(*i).first<<" "<<(*i).second<<endl;
```

Chạy thử bạn sẽ thấy các value trong map đã được sắp xếp lại vị trí theo các key của chúng
comparator dùng với các sequential container

CODE

```
class People
{
    public:int age;
    People(int age) { (*this).age=age; };
class AgeSort{
    public:bool operator() (const People*& a,const People*& b)
    {
        return (*a).age>(*b).age;
    }
};
typedef list<const People*> LP;
int main()
{
    const People* p1 = new People(5);const People* p2 = new People(7);
    LP p;p.push_back(p1);p.push_back(p2);
    p.sort(AgeSort());//using sort with comparator
    for(LP::const_iterator i=p.begin();i!=p.end();++i) cout<<(**i).age;
    return 0;
}
```

multimap

Với map thì mỗi key chỉ ánh xạ đến một và chỉ một value. Với multimap thì mỗi key có thể ánh xạ đến nhiều hơn một value, nói cách khác là nhiều value trong multimap có chung một key

CODE

```
#include<map>
typedef multimap<string,Person> MP;
MP multimapPerson;
multimapPerson.insert(MPVT("one",Person("Nam")));
multimapPerson.insert(MPVT("one",Person("Viet")));
display(multimapPerson);
typedef multimap<Person,int,comparePerson> MAP;
```

Cũng chính vì lí do nhiều value trong multimap có thể có chung một key nên multi không có operator[] như map, tức là bạn không thể gọi multimapPerson["one"]

set

set cũng giống map ngoại trừ một điều, key cũng chính là value

CODE

```
#include<set>
set<int> s;
for(int j=0;j<6;j++) s.insert(rand());
for(set<int>::iterator i=s.begin();i!=s.end();++i) cout<<(*i)<<endl;
```

set dùng với comparator (greater đóng vai trò comparator)

CODE

```
set<int,greater<int> > s;
for(int j=0;j<6;j++) s.insert(rand());
for(set<int,greater<int> ::iterator i=s.begin();i!=s.end();++i)
```

```
cout<<(*i)<<endl;
```

set không có operator[]

multiset

multiset cũng giống set ngoại trừ một điều, mỗi key có thể ánh xạ đến nhiều hơn một value, nói cách khác là nhiều value trong multiset có chung một key

Bạn có thể thắc mắc điều này chẳng có ý nghĩa gì, vì trong set thì key cũng chính là value, Không, chúng có khác đấy, thử xem nhé:

CODE

```
#include<set>
set<int> s;
s.insert(1);
s.insert(1);
for(set<int>::iterator i=s.begin();i!=s.end();++i) cout<<(*i)<<endl;
multiset<int> ms;
ms.insert(3);
ms.insert(3);
for(multiset<int>::iterator mi=ms.begin();mi!=ms.end();++mi)
    cout<<(*mi)<<endl;
```

BÀI 12: CÁC BỘ TƯƠNG THÍCH VÀ CÁC THƯ VIỆN KHÁC

container adapter (các bộ tương thích lưu trữ)

Bao gồm stack, queue và priority_queue

Các bộ tương thích lưu trữ, dưới đây gọi là các bộ tương thích, làm các bộ lưu trữ khác trở nên tương thích với nó bằng cách đóng gói (encapsulate) các bộ lưu trữ khác trở thành bộ lưu trữ cơ sở của nó. Ví dụ

CODE

```
stack<int, vector<int>> s;
```

Khi đó vector trở thành bộ lưu trữ cơ sở của bộ tương thích stack

Nếu không khai báo bộ lưu trữ cơ sở, stack và queue mặc định sử dụng deque làm bộ lưu trữ cơ sở, trong khi priority_queue mặc định sử dụng vector làm bộ lưu trữ cơ sở, có nghĩa là khi khai báo

CODE

```
stack<int> s;
```

thực ra là

CODE

```
stack<int, deque<int>> s;
```

stack và queue

stack là LIFO, queue là FIFO, xem thử sự khác biệt qua ví dụ palindrome sau

(lưu ý, palindrome tức là một từ đọc xuôi hay ngược đều như nhau, ví dụ 12321, level, aka)

CODE

```
#include <stack>
#include <queue>
using namespace std;
int main() {
    stack<char> stackInt; queue<char> queueInt;
    char a;//store temp user input
    int n;//no of numbers user intend to input
    cout<<"how many elements:"; cin>>n;
    for(int i=0;i<n;i++) {
        cin>>a;
        stackInt.push(a);
        queueInt.push(a);
    }
    for(int i=0;i<n;i++) {
        if(stackInt.top()!=queueInt.front()) {
            cout<<"not a palindrome"<<endl; break;
        }
        stackInt.pop(); queueInt.pop();
        if(i==n-1) cout<<"a palindrome"<<endl;
    }
}
```

Lưu ý 2 cả stack và queue đều có các hàm sau

void push(T) thêm phần tử vào

void pop(T) gỡ phần tử ra

stack có thêm hàm

T top() truy xuất phần tử tiếp theo

queue có thêm hàm

T front() truy xuất phần tử tiếp theo

T back() truy xuất phần tử cuối cùng của queue

priority_queue

priority_queue là queue trong đó phần tử đầu tiên luôn luôn là phần tử lớn nhất theo một tiêu chuẩn sắp xếp nào đó priority_queue giống như khái niệm heap (đống) mà ta đã biết (heap và giải thuật heapsort trong môn CSDL). Thực ra priority_queue chỉ là queue mặc định có cài sẵn thêm comparator less<T> giống như các associative container thôi. Ta có thể cài lại comparator do ta định nghĩa cho nó (ví dụ bài dưới đây cài greater<T>).

CODE

```
#include <queue>
class Plane{
    int fuel;
public:    Plane(int fuel){(*this).fuel=fuel;}
    friend ostream& operator<<(ostream& os, const Plane& p){
        os<<p.fuel<<endl; return os;}
    bool operator>(const Plane& p) const{
        return fuel>p.fuel;}
};

typedef priority_queue<Plane, vector<Plane>, greater<Plane> > PriorityQueuePlane;
int main(){
    vector<Plane> vP;
    vP.push_back(Plane(4));vP.push_back(Plane(7));
    vP.push_back(Plane(3));vP.push_back(Plane(9));
    PriorityQueuePlane v(vP.begin(),vP.end());
    while(!v.empty()){
        cout<<v.top();v.pop();
    }
    return 0;
}
```

Lưu ý là priority_queue có push, pop và top, không có front và back

iterator adapter (các bộ tương thích con trỏ)

Các bộ tương thích iterator làm các container và iterator khác trở nên tương thích với nó bằng cách đóng gói (encapsulate) các container và iterator khác trở thành container và iterator cơ sở của nó. Chúng có dạng khai báo cơ bản như sau

CODE

```
#include<iterator>
template<class Container, class Iterator>
class IteratorAdapter
{
    //nội dung
};
IteratorAdapter<vector<int>, vector<int>::iterator> vectorIntAdapter;
```

Không học thêm về iterator và iterator adapter

function adapter (các bộ tương thích hàm)

Có 2 bộ tương thích hàm chúng ta đã học trước đó là bind1st và bind2nd. Chúng ta sắp học not1, not2, mem_fun, mem_fun_ref và ptr_fun. Tất cả đều nằm trong thư viện functional

not1

Đổi giá trị trả về của một unary predicate từ false thành true và ngược lại, unary predicate phải được định nghĩa là unary_function

Ví dụ dùng IsOdd tìm các số chẵn (nghĩa là IsOdd trả về not(true))

CODE

```
class IsOdd:public unary_function<int,bool>{
public:bool operator()(const int& n) const{return n%2;}
};

int main(int argc, char* argv[]){
    int a[] = {1,2,3,4,5};
```

```
cout<<count_if(a,a+5,not1(IsOdd()))<<endl;
return 0;
}
```

not2

Đổi giá trị trả về của một binary predicate từ false thành true và ngược lại, binary predicate phải được định nghĩa là binary_function

Ví dụ dùng compare để so sánh 2 mảng với các phần tử không bằng nhau (nghĩa là compare trả về not(true))

CODE

```
class compare:public binary_function<int,int,bool>{
public:bool operator()(int i,int j) const{return i==j;}
};

int main(int argc, char* argv[]){
    int a[] = {1,2,3,4,5};
    int b[] = {6,7,8,9,10};
    cout<<equal(a,a+5,b,not2(compare()))<<endl;
    return 0;
}
```

ptr_fun

Chuyển một con trỏ hàm (function pointer) thành một functor

CODE

```
int addition(int a,int b){return a+b;}
int output(int a){cout<<a<<endl;return 0;}
int(*cong)(int,int) = addition;
int(*xuat)(int) = output;
int main()
{
    int a[] = {1,2,3,4,5};
    int b[] = {6,7,8,9,10};
    int c[5];
    transform(a,a+5,b,c,ptr_fun(cong));
    for_each(c,c+5,ptr_fun(xuat));
    return 0;
}
```

Ở đây chúng ta có binary function là addition và unary function là output, và binary function pointer là cong và unary function pointer là xuat, và ta dùng ptr_fun để chuyển các con trỏ hàm này thành binary functor và unary functor để đóng vai trò predicate dùng trong hai hàm transform và for_each

ptr_fun chỉ dùng cho stand-alone và static member function, với non-static member function phải dùng mem_fun hay mem_fun_ref

mem_fun

Chuyển một hàm thành viên (member function) của một lớp thành một functor và truyền vào functor này các đối số là các con trỏ mà trỏ đến các đối tượng của lớp đó

CODE

```
class Person{
    int age;
public:
    Person(int age):age(age){}
    int display(){cout<<age<<endl;return 0;}
};

int main(){
```

```

list<Person*> l;
l.push_back(new Person(4));
l.push_back(new Person(5));
for_each(l.begin(),l.end(),mem_fun(&Person::display));
return 0;
}

```

mem_fun_ref

Chuyển một hàm thành viên (member function) của một lớp thành một functor và truyền vào functor này các đối số là các tham chiếu mà tham chiếu đến các đối tượng của lớp đó

CODE

```

class Person{
    int age;
public:
    Person(int age):age(age){}
    int display(){cout<<age<<endl;return 0;}
};
int main(){
    list<Person> l;
    l.push_back(Person(4));
    l.push_back(Person(2));
    l.push_back(Person(5));
    for_each(l.begin(),l.end(),mem_fun_ref(&Person::display));
    return 0;
}

```

Mục đích chính là để tăng hiệu suất chương trình, thứ cực kì quan trọng trong lập trình game. Tưởng tượng bạn sẽ phải gọi 1 câu lệnh như thế này

CODE

```
for(list<Person*>::iterator i=l.begin();i!=l.end();i++) (**i).display();
```

gọi tới từng hàm thành viên của từng phần tử của list, giảm hiệu suất kinh khủng

Thay vào đó dùng mem_fun hay mem_fun_ref, chỉ cần truyền vào một con trỏ hay một tham chiếu tới hàm thành viên, tăng hiệu suất rõ rệt.

KHUYẾN CÁO: ptr_fun và mem_fun hay mem_fun_ref, cả 3 hàm này đều trả lại functor, được sử dụng rất nhiều không chỉ trong lập trình game vì tăng tốc độ và hiệu suất chương trình. So sánh giữa các ngôn ngữ với nhau, nhờ vào những đặc điểm như con trỏ, etc, cùng với những hàm tiện ích đặc biệt trong STL nhất là 3 hàm này, để cùng đạt được một mục đích thì dùng C++ đạt được tốc độ và hiệu suất hơn bất kì ngôn ngữ bậc cao nào khác. Do đó bạn nên hiểu và sử dụng nhuần nhuyễn thư viện STL, nhất là 3 hàm này. Đây cũng là phần trung tâm chính của cả môn học C/C++ nâng cao

Thư viện numeric

Trong thư viện này có một hàm cần chú ý, hàm accumulate

CODE

```

#include<numeric>
double acc(double total, double elements){
    return total+elements;
}
int main(){
    multiset<double> s;
    for(int i=0;i<6;i++) s.insert(0.3);
    double sum = accumulate(s.begin(),s.end(),0.0,ptr_fun(acc));
}

```

Hàm accumulate truyền vào functor acc (do ptr_fun chuyển từ function thành functor) tham số là total = 0.0 và lần lượt là các phần tử của set, sau đó acc tính tổng total và các element rồi trả về để accumulate tích lũy và cuối cùng trả giá trị ra biến sum

Thư viện bitset

bitset có cấu trúc giống như một mảng, nhưng mỗi phần tử chỉ chiếm một bit (nên nhớ kiểu dữ liệu char mỗi phần tử chiếm 8 bit)

Ví dụ sau ta khởi tạo một bitset 7 phần tử với 5 phần tử đầu là 1,1,0,1,0

CODE

```
#include<bitset>
bitset<7> b(string("01011"));
for(int i=0;i<7;i++) cout<<b[i]<<endl;
```

Thư viện valarray

valarray giống như là một mảng lưu trữ các phần tử. Nó đáng chú ý vì nó có thể làm việc được với các hàm toán học thường dùng trong thư viện <cmath> và cũng như nhiều phép toán thường dùng khác

CODE

```
#include<valarray>
#include<cmath>
int i=0;
int a1[] = {3,2,6,4,5};
valarray<int> v1(a1,5);
v1 <<= 3;//phép toán << (dịch trái bit)
for(i=0;i<4;i++) cout<<v1[i]<<endl;
double a2[] = {2.4,6.8,0.2};
valarray<double> v2(a2,3);
v2 = sin(v2);//hàm sin
for(i=0;i<3;i++) cout<<v2[i]<<endl;
```

STL đến đây là kết thúc. Môn C/C++ nâng cao còn độ vài bài nữa thôi là xong

Những phần sau đã được cắt bớt, không post để giảm nhẹ chương trình, ai thích có thể tự tìm hiểu thêm: iterator, iterator adapter và allocator

BÀI 13: RTTI, I/O, EXTERN VÀ PREPROCESSOR DIRECTIVE

RTTI (Runtime type identification)

Trong Java, để biết một object có phải là một instance của một class hay không, ta dùng instanceof
if(os instanceof ostream)

Trong C++ ta dùng hàm typeid
if(typeid(os)==typeid(ostream))

Trong C++, nếu ta muốn overload toán tử xuất << (output) 2 lần cùng với ostream và ofstream để vừa có thể xuất ra màn hình và tập tin trong cùng một chương trình, chương trình thực ra sẽ làm việc không thành công như ta mong muốn, dù không báo lỗi gì cả. Đó là vì ofstream là lớp con của ostream, do đó toán tử xuất của nó bị khai báo trùng hợp với toán tử xuất của cha nó. Điều này cũng tương tự như khi ta muốn overload toán tử nhập >> (input) 2 lần cùng với istream và ifstream, vì ifstream là lớp con của istream.

Khi phát triển những game thương mại lớn nếu để "lọt số" những lỗi ngầm khó phát hiện như vậy thì khi có "chuyện gì" xảy ra, với số lượng kinh hoàng các lớp và các toán tử đã được phát triển thì thời gian đi tìm và sửa lỗi sẽ cũng rất ... kinh hoàng. Do đó, để đảm bảo an toàn, khi phải overload cùng một toán tử cho 2 lớp cha và con, phải sử dụng RTTI

Ta sử dụng RTTI bằng cách dùng typeid và downcast bằng dynamic_cast. RTTI (Runtime type identification) (xác định kiểu dữ liệu lúc thực thi) Lúc thực thi, chương trình sẽ xác định kiểu dữ liệu của object chính xác là instance của cha hay con. Trước hết, ta viết riêng hàm cho con trước. Nếu xác định là instance của con, ta ép kiểu của object xuống thành kiểu của con rồi cho thực hiện hàm ta viết riêng cho con. Nếu không phải là vẫn **div, id: post-26368, class: postcolor** thường. Lớp cha phải có hàm ào (istream và ostream đều thỏa điều này)

Ví dụ dưới đây ta viết 2 hàm printToFile và readFromFile dành cho con (ofstream và ifstream) trước rồi dùng typeid và downcast

CODE

```
#include<iostream>
#include<fstream>
using namespace std;
class Person{
    char* name;
public:
    Person(){}
    Person(char* name):name(name){}
    void setName(char* name){
        (*this).name = new char[strlen(name)+1];
        strcpy((*this).name,name);
    }
    char* getName() const{return name;}
    void printToFile(ofstream& os) const{os<<*this;}
    void readFromFile(ifstream& is){is>>*this;}
    friend ostream& operator<<(ostream& os,const Person& p){
        if(typeid(os)==typeid(ofstream))
            p.printToFile(dynamic_cast<ofstream&>(os));//downcast
        else os<<p.getName()<<endl;
        return os;
    }
    friend ofstream& operator<<(ofstream& ofs,const Person& p){
        ofs<<p.getName()<<endl;
        return ofs;
    }
    friend istream& operator>>(istream& is,Person& p){
        if(typeid(is)==typeid(ifstream))
            p.readFromFile(dynamic_cast<ifstream&>(is));//downcast
        else{
            char* temp = new char[20];
            is.getline(temp,21);
            fflush(stdin);
            p.setName(temp);
        }
        return is;
    }
    friend ifstream& operator>>(ifstream& ifs,Person& p){
        char* temp = new char[20];
        ifs>>temp;
        p.setName(temp);
        return ifs;
    }
};
int main(){
    Person a;
    cin>>a;
    ofstream ofs("a.txt");
    ofs<<a;
    ofs.close();
    cout<<a;
    Person b;
    ifstream ifs("a.txt");
    ifs>>b;
    ofs.open("b.txt");
    ofs<<b;
    cout<<b;
    ofs.close();
    return 0;
}
```

I/O LIBRARY (THƯ VIỆN NHẬP XUẤT)

Ta đã học qua bộ thư viện này, chủ yếu ios, iostream, fstream. Ta không đi sâu chi tiết thư viện này mà chỉ chú ý thêm vài thứ sau đây

filebuf

Đọc toàn bộ tập tin vào một chuỗi, sử dụng filebuf trong <fstream>

filebuf (file buffer) bộ đệm tập tin

CODE

```
ifstream fin;fin.open("data.dat");//mở file, đưa vào stream
filebuf *buf = fin.rdbuf();//đọc toàn bộ stream vào buffer
long size=(*buf).pubseekoff(0,ios::end,ios::in);//kích thước của buffer
(*buf).pubseekpos(0,ios::in);//vị trí tìm kiếm
char* temp = new char[size];//tạo mảng kí tự
```

```
(*buf).sgetn(temp,size);//chuyển từ buffer vào mảng kí tự  
cout.write(temp,size);//viết mảng kí tự vào luồng xuất ra màn hình  
string s(temp);//chuyển mảng kí tự ra chuỗi
```

Thư viện <sstream>

Có 2 lớp phải chú ý là ostringstream và istream

Những đối tượng được đưa vào ostringstream vẫn giữ nguyên kiểu dữ liệu của nó chứ không hề chuyển kiểu thành string, ví dụ

CODE

```
string s="Hi there ";double d=45.67;int n=2;  
ostringstream output;output<<s<<d<<n;
```

Muốn xuất ra những gì đã đưa vào, ta dùng istream

CODE

```
string input="Hi there 45.67 2";string s1,s2;double d;int n;  
istream values(input);values>>s1>>s2>>d>>n;
```

Bây giờ ta có thể xuất toàn bộ dữ liệu trong một file ra dùng filebuf

CODE

```
filebuf *buf = fin.rdbuf();  
...  
string s(temp);  
istringstream values(s);values>>s1>>s2>>s3>>...;
```

Từ khóa extern

Từ khóa extern thông báo với trình biên dịch là một phần của chương trình đã được liên kết với một ngôn ngữ khác hoặc đã được khai báo theo một qui ước khác hoặc trong một phần chương trình khác.

Trường hợp thứ nhất: ta có một tập tin c.obj chứa mã nhị phân của hàm dosomething viết bằng C. Bây giờ ta muốn viết một chương trình C++ sử dụng thư viện ấy. Ta khai báo trong main.cpp

CODE

```
extern "C" {  
    void dosomething(int i);  
}  
int main(int argc,char** argv) {  
    dosomething(5);  
}
```

Trường hợp thứ hai: ta có một thư viện đồ họa viết bằng C là graphics.lib và tập tin header của nó là graphics.h. Bây giờ ta muốn viết một chương trình C++ sử dụng thư viện ấy. Ta khai báo trong main.cpp

CODE

```
extern "C" {  
    #include "graphics.h"  
}
```

Trường hợp thứ ba: ta có một dự án có 2 tập tin 1.cpp và 2.cpp trong đó biến a và hàm in đã khai báo ở tập tin 1.cpp như sau

CODE

```
int a=7;  
void in(int a){cout<<a;}
```

thế thì ở tập tin 2.cpp ta khai báo

CODE

```
extern int a;cout<<a;  
extern void in(int a);in(25);
```

Preprocessor directive (chỉ thị tiền xử lí)

preprocessor (bộ tiền xử lí)

Trước khi biên dịch một chương trình, bộ biên dịch (compiler) chuyển các file mã nguồn qua bộ tiền xử lí (preprocessor). Nhiệm vụ của preprocessor là xử lí các file mã nguồn qua việc xử lí các chỉ thị tiền xử lí (preprocessor directive) cho ra các file mã nguồn tương đương với các file mã nguồn ban đầu. Việc xử lí này bao gồm gỡ bỏ các comment (chú thích) thi hành các chỉ thị tiền xử lí như #include, #define hoặc tương đương

vẫn vẫn và hoàn toàn chỉ ở mức text level, do đó việc kiểm tra lỗi rất hạn chế

Ví dụ

--file header.h--

CODE

```
void add(int);  
#include "function.cpp"
```

--file function.cpp--

CODE

```
void add(int a) {  
    return ++a;  
}
```

--file main.cpp--

CODE

```
#include "header.h"  
//dung macro  
#define abs(x) ((x>0)?x:-x)  
#define two 2  
int main() {  
    add(two*abs(-5));  
    return 0;  
}
```

Sau khi qua tiền xử lí sẽ trở thành một file như sau

CODE

```
void add(int);  
void add(int a) {  
    return ++a;  
}  
int main() {
```

```
add(2*(-5>0)?-(-5));
return 0;
}
```

Các chỉ thị tiền xử lý (preprocessor directive)

#define: định nghĩa một macro (quá dễ rồi)

#include: bao gồm một tập tin hay macro vào chương trình (quá dễ rồi)

#undef: hủy bỏ định nghĩa một macro, macro đó có thể định nghĩa lại bằng #define, ví dụ

CODE

```
#define max(a,b) ((a>b)?a:b)
#undef max
#define max(a,b) ((a>b)?2*a:3*b)
```

#error: định nghĩa câu thông báo khi gặp lỗi, ví dụ

CODE

```
#error bi loi roi
int main(){
    int a = 10/0;
}
```

Câu thông báo lỗi sẽ là câu ta đã định nghĩa

#pragma: các tùy chọn chỉ thị biên dịch (tùy thuộc vào trình biên dịch)

Các chỉ thị điều kiện

Bao gồm #if (nghĩa là if) #elif (nghĩa là else if) #else (nghĩa là else) #endif (nghĩa là end if) ví dụ đoạn mã sau

CODE

```
#if MAX_WIDTH>10
#undef MAX_WIDTH
#define MAX_WIDTH 10
#elif MAX_WIDTH<1
#undef MAX_WIDTH
#define MAX_WIDTH 1
#else
#undef MAX_WIDTH
#define MAX_WIDTH 5
#endif
```

có thể viết lại giống như sau

CODE

```
if(max_width>10)
{
    #undef max_width;
    max_width = 10;
}
else
{
    if(max_width<1)
    {
        #undef max_width;
        max_width = 1;
    }
    else
    {
        #undef max_width;
        max_width = 5;
    }
}
```

ngoài ra còn có

#ifdef có nghĩa là "nếu đã định nghĩa"

tương tự như nó là #if defined

#ifndef có nghĩa là "nếu chưa định nghĩa"

tương tự như nó là #if !defined

CODE

```
#ifdef MYDEF_H
#define MYLIB_H
#endif
#ifndef MYHEADER_H
#include "myheader.h"
#endif
```

Nếu đã định nghĩa MYDEF_H thì định nghĩa thêm MYLIB_H

Nếu chưa định nghĩa MYHEADER_H thì bao gồm tập tin "myheader.h" vào mã nguồn

Viết lại dùng defined

CODE

```
#if defined(MYDEF_H)
#define MYLIB_H
#endif
#if !defined(MYHEADER_H)
#include "myheader.h"
#endif
```

Chỉ thị #line

FILE là một macro đã định nghĩa sẵn, trả về đường dẫn của tập tin gọi macro

LINE là một macro đã định nghĩa sẵn, trả về thứ tự của dòng lệnh gọi macro

CODE

```
#include<iostream>
using namespace std;
int main()
```

```
{  
    cout<<__FILE__<<endl;  
    cout<<__LINE__<<endl;//dòng thứ 6  
    return 0;  
}
```

chỉ thị #line định nghĩa lại thứ tự của dòng tiếp theo nó

CODE

```
#include<iostream>  
using namespace std;  
#line 46  
int main()//dòng thứ 46  
{  
    cout<<__FILE__<<endl;  
    cout<<__LINE__<<endl;//dòng thứ 46+3  
    return 0;  
}
```

chỉ thị #line còn định nghĩa lại đường dẫn tập tin

CODE

```
#include<iostream>  
using namespace std;  
#line 46 "c:\\main.cpp"  
int main()  
{  
    cout<<__FILE__<<endl;//đường dẫn mới  
    cout<<__LINE__<<endl;  
    return 0;  
}
```

#line và __FILE__ và __LINE__ quan trọng trong việc dùng để debug

Chỉ thị toán tử #

gọi là stringizing operator directive, chỉ dùng với tham số của macro Nó sẽ chuỗi hóa (stringize) tham số này đơn giản chỉ bằng cách bọc tham số trong cặp nháy kép, ví dụ

CODE

```
#define str(x) cout<<#x  
int main(){  
    str(dau bung);  
}
```

Chỉ thị toán tử #@

gọi là charizing operator directive, chỉ dùng với tham số của macro Nó sẽ kí tự hóa (charize) tham số này đơn giản chỉ bằng cách bọc tham số trong cặp nháy đơn, ví dụ

CODE

```
#define chr(x) #@x  
int main(){  
    char c=chr(K);  
}
```

Chỉ thị toán tử ##

gọi là merging operator directive, chỉ dùng với tham số của macro Nó sẽ hợp (merge) tham số với chuỗi macro đã định nghĩa với nó, ví dụ

CODE

```
#define merging(n) cout<<a##n  
int main(){  
    int a3 = 1;  
    merging(3);  
}
```

BÀI 14: DESTRUCTOR, CONSTRUCTOR, CONVERSION VÀ DEBUG

Hàm hủy ảo (virtual destructor)

Trong ví dụ sau, hàm hủy của Derived sẽ không được gọi

CODE

```
class Base{  
    public:Base(){};~Base(){};  
};  
class Derived:public Base{  
    public:Derived(){};~Derived(){};  
};  
int main(){  
    Base* b = new Derived(); delete b;  
}
```

Trong trường hợp này, ta cần khai báo hàm hủy của Base là hàm hủy ảo (tuyệt đối không được là pure virtual destructor)

CODE

```
class Base{  
    public:Base(){};virtual ~Base(){};
```

Hàm khởi tạo chuyển kiểu (conversion constructor)

Bất kì một constructor một đối số nào đều có thể trở thành một conversion constructor

CODE

```
class Thing{  
    int num;  
public:  
    Thing(int num) {(*this).num=num;}  
    friend ostream& operator<<(ostream& os, const Thing& t) {os<<t.num;}  
};  
void display(const Thing& t){cout<<t<<endl;}  
int main(){  
    display(7);  
}
```

Hàm display sẽ kiểm tra lớp Thing sau đó dùng Thing(int num) làm conversion constructor để gọi Thing t(7);display(t); Nếu chúng ta muốn constructor không bị dùng làm conversion constructor nữa thì chỉ việc thêm từ khóa explicit vào khai báo nguyên mẫu của nó

CODE

```
explicit Thing(int num) {....};
```

Thứ tự khởi tạo đối số của constructor

constructor sẽ chỉ khởi tạo giá trị cho những đối số của nó theo đúng thứ tự chúng được khai báo trong lớp, chứ không phải theo thứ tự đối số chúng được khai báo trong nguyên mẫu hàm. Ví dụ

CODE

```
class Thing{  
    int total,part1,part2;  
    Thing(int a,int b):part1(a),part2(b),total(part1+part2){}  
};
```

total sẽ là biến được constructor khởi tạo giá trị trước tiên (vì trong lớp nó được khai báo trước tiên) và lúc này part1 và part2 đều chưa có giá trị (vì chưa được khởi tạo) nên phép cộng part1+part2 lập tức sẽ tạo ra lỗi chương trình

Hàm chuyển kiểu (conversion function)

Hàm chuyển kiểu là loại hàm một lớp có chức năng tự động chuyển một object của lớp đó thành một kiểu dữ liệu

-phải là hàm thành viên không có tham số

-kiểu trả về không nên là void (không có ý nghĩa gì nữa)

CODE

```

class ViDu{
    char* s;
    int x,y;
public:
    ViDu(char* s,int x,int y):s(s),x(x),y(y){}
    operator char*(){return s;}
    operator int(){return x*y;}
    operator ViDu(){}
    int operator()(int i){return i;}
};

int main(){
    ViDu v("hello",5,3);
    cout<<v<<endl;//v trả về char*
    int i = 5;
    i += v;//v trả về int
    i += v(4);//v là functor
    cout<<i<<endl;
    return 0;
}

```

Các kĩ thuật debug

Debug là một phần quan trọng của lập trình game. Game phát hành mà có bug thì chẳng ai mua, uy tín công ty sẽ sa sút và bạn sẽ bị sa thải.

Các trình biên dịch thường hỗ trợ 2 chế độ Debug và Release. Bài này yêu cầu bạn sử dụng một trình biên dịch có hỗ trợ cả 2 chế độ và biên dịch các ví dụ với cả 2 chế độ Debug và Release

Sử dụng __FILE__ và __LINE__

Ví dụ giả sử chương trình dưới đây sẽ có bug, ta dùng __FILE__ và __LINE__ để in ra thông báo tập tin và dòng mã có lỗi

CODE

```

int main(){
    cout<<"Bugs in "<<__FILE__<<" at line "<<__LINE__<<endl;
}

```

Ta khai báo một macro cho việc debug. Macro của ta chỉ chạy dưới chế độ Debug. Nếu biên dịch dưới chế độ Debug, macro BUG sẽ in ra thông báo tập tin và dòng mã có lỗi, nếu không thì macro BUG không có ý nghĩa gì cả. Chế độ Debug được định nghĩa sẵn với macro DEBUG hoặc _DEBUG tùy trình biên dịch

CODE

```

#ifndef DEBUG
#define BUG cout<<"Bugs in "<<__FILE__<<" at line "<<__LINE__<<endl
#endif
#ifndef DEBUG
#define BUG
#endif
int main()
{
    BUG;
    return 0;
}

```

Có 2 vấn đề đau đầu mà lập trình viên C++ hay gặp phải là null pointer và leak memory

Phát hiện null pointer

null pointer

null pointer là một nguy hiểm chết người ta hay gặp phải

Con trỏ mà trỏ tới một null pointer

CODE

```
int* p = NULL;  
int** pp = &p; cout<<**pp;
```

Một cách phát hiện

CODE

```
int* p = NULL;  
if(p!=NULL) {int** pp = &p; cout<<**pp; }
```

Tham chiếu mà tham chiếu tới một null pointer

CODE

```
int* p = NULL;  
int& r = *p; cout<<r;
```

Một cách phát hiện

CODE

```
int* p = NULL;  
if(p!=NULL) {int& r = *p; cout<<r; }
```

Sử dụng hàm assert

Ta debug cái lỗi chết người này bằng hàm assert trong thư viện <cassert>

void assert(int test); //nếu test=0 thì hàm sẽ xuất lỗi ra stderr và gọi hàm abort hủy bỏ chương trình

CODE

```
#include<cassert>  
int* p = NULL;  
assert(p); //p=0 nên assert fail  
int& r = *p; cout<<r;
```

Và bạn có thể dùng assert để tránh null pointer ví dụ như sau (điều gì xảy ra nếu một trong 2 con trỏ là null pointer bạn cũng biết rồi)

CODE

```
void swap(int* a, int* b) {  
    assert(a);  
    assert(b);  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Phát hiện leak memory

leak memory (rò rỉ bộ nhớ)

leak memory (rò rỉ bộ nhớ) là lỗi xảy ra khi lập trình viên

-quên hủy bỏ vùng nhớ đã cấp phát

-cấp phát vùng nhớ bằng new[] nhưng hủy bỏ vùng nhớ bằng delete

CODE

```
int* p = new int[5];  
delete p;
```

-vì lí do nào đấy lệnh delete đã không được gọi, ví dụ chương trình ném ra ngoại lệ hoặc thoát bất ngờ

CODE

```
try{
    int* a = new int[5];
    cout<<a[6]<<endl;
    delete [] a;
}catch(std::out_of_range& ex) {
    cerr<<ex.what();
    exit(EXIT_FAILURE);
}
```

Phát hiện rò rỉ bộ nhớ với malloc và free

Có nhiều cách viết mã để phát hiện rò rỉ bộ nhớ nhưng nói chung đều tuân theo qui luật chung: nếu vùng nhớ đã cấp phát bằng new/malloc mà không giải phóng bằng delete/free thì vùng nhớ đó đã bị rò rỉ

Chương trình sau phát hiện rò rỉ bộ nhớ bằng cách dùng một list lưu thông tin về những vùng nhớ đã cấp phát. Nếu cấp phát bằng hàm xmalloc thì lưu thông tin vùng nhớ đó vào trong list, nếu giải phóng vùng nhớ đó bằng xfree thì xóa thông tin vùng nhớ đó ra khỏi list

CODE

```
#include<iostream>
#include<list>
#include<malloc.h>
using namespace std;
struct MEM_INFO
{
    void* address;
    int size;
    char file[256];
    int line;
};
list<MEM_INFO> lmi;
list<MEM_INFO>::iterator lmii;
void* xmalloc(int size,char* file,int line)
{
    void* ptr = malloc(size);
    if(ptr!=NULL)
    {
        MEM_INFO memInfo;
        memset(&memInfo,0,sizeof(MEM_INFO));
        memInfo.address = ptr;
        memInfo.size = size;
        strcpy(memInfo.file,file);
        memInfo.line = line;
        lmi.push_back(memInfo);
    }
    return ptr;
}
void xfree(void* mem_ref)
{
    free(mem_ref);
    for(lmii=lmi.begin();lmii!=lmi.end();++lmii)
    {
        if((*lmii).address==mem_ref) {lmi.erase(lmii);break;}
    }
}
int main()
```

```

{
    int* a = (int*)xmalloc(2, __FILE__, __LINE__);
    char* b = (char*)xmalloc(4, __FILE__, __LINE__);
    double* c = (double*)xmalloc(8, __FILE__, __LINE__);
    xfree(a);
    xfree(c);
    for(lmii=lmi.begin(); lmii!=lmi.end(); ++lmii)
    {
        MEM_INFO memInfo=*lmii;
        cout<<"Address:"<<memInfo.address<<endl;
        cout<<"Size:"<<memInfo.size<<endl;
        cout<<"File:"<<memInfo.file<<endl;
        cout<<"Line:"<<memInfo.line<<endl;
    }
    return 0;
}

```

Tương tự bạn có thể viết cho calloc và realloc

Phát hiện rò rỉ bộ nhớ với new và delete

Với vấn đề phát hiện leak memory, chúng ta nên dùng malloc/free với primitive và dùng new/delete với object, vì delete còn gọi destructor của object. Lý do nữa là overload toán tử new và delete ở global scope rất phức tạp và không phải trình biên dịch nào cũng hỗ trợ overload hai toán tử này. Thêm nữa những ràng buộc chặt chẽ với new/delete khiến đôi khi việc này trở nên không thể. Đây là một trong những lí do vì sao malloc/free vẫn còn hữu dụng cho dù đã có new/delete

Ví dụ sau chúng ta sẽ viết một lớp, overload các toán tử new, delete, new[] và delete[] cho lớp đó. Bạn sẽ thấy là phiên bản overload của các toán tử này đã được gọi

CODE

```

#include<iostream>
#include<malloc.h>
using namespace std;
class MyClass
{
public:
    int data;
    MyClass():data(0){}
    MyClass(int data):data(data){}
    void* operator new(unsigned int size)
    {
        void* ptr = malloc(size);
        cout<<"new"<<endl;
        return ptr;
    }
    void operator delete(void* p)
    {
        cout<<"delete"<<endl;
        free(p);
    }
    void* operator new[](unsigned int size)
    {
        void* ptr = malloc(size);
        cout<<"new[]"<<endl;
        return ptr;
    }
    void operator delete[](void* p)
    {

```

```

        cout<<"delete[]"<<endl;
        free(p);
    }
};

int main()
{
    MyClass* a = new MyClass;
    *a = MyClass(2);
    cout<<(*a).data<,<endl;
    delete a;

    MyClass* b = new MyClass[5];
    for(int i=0;i<5;i++) b[i] = MyClass(i);
    for(int i=0;i<5;i++) cout<<b[i].data;
    delete [] b;
    return 0;
}

```

Còn ví dụ dưới chúng ta sẽ xác định rõ rỉ bộ nhớ nếu khởi tạo bằng new mà không hủy bỏ bằng delete. Với khởi tạo bằng new[] mà hủy bỏ bằng delete ta sẽ gọi delete []

CODE

```

#include<iostream>
#include<list>
#include<malloc.h>
using namespace std;
struct MEM_INFO
{
    void* address;
    int size;
    char file[256];
    int line;
};
list<MEM_INFO> lmi;
list<MEM_INFO>::iterator lmii;
class MyClass
{
public:
    void* operator new(unsigned int size,char* file,int line)
    {
        void* ptr = malloc(size);
        if(ptr!=NULL)
        {
            MEM_INFO memInfo;
            memset(&memInfo,0,sizeof(MEM_INFO));
            memInfo.address = ptr;
            memInfo.size = size;
            strcpy(memInfo.file,file);
            memInfo.line = line;
            lmi.push_back(memInfo);
        }
        return ptr;
    }
    void operator delete(void* p)
    {
        delete [] p;
    }
}

```

```

for(lmii=lmi.begin();lmii!=lmi.end();++lmii)
{
    if((*lmii).address==p) {lmi.erase(lmii);break;}
}
}

void* operator new[] (unsigned int size,char* file,int line)
{
    void* ptr = malloc(size);
    if(ptr!=NULL)
    {
        MEM_INFO memInfo;
        memset(&memInfo,0,sizeof(MEM_INFO));
        memInfo.address = ptr;
        memInfo.size = size;
        strcpy(memInfo.file,file);
        memInfo.line = line;
        lmi.push_back(memInfo);
    }
    return ptr;
}

void operator delete[] (void* p)
{
    delete [] p;
    for(lmii=lmi.begin();lmii!=lmi.end();++lmii)
    {
        if((*lmii).address==p) {lmi.erase(lmii);break;}
    }
}
};

int main()
{
    MyClass* a = new(__FILE__,__LINE__) MyClass;
    MyClass* b = new(__FILE__,__LINE__) MyClass[5];
    for(lmii=lmi.begin();lmii!=lmi.end();++lmii)
    {
        MEM_INFO memInfo=*lmii;
        cout<<"Address:"<<memInfo.address<<endl;
        cout<<"Size:"<<memInfo.size<<endl;
        cout<<"File:"<<memInfo.file<<endl;
        cout<<"Line:"<<memInfo.line<<endl;
    }
    return 0;
}

```

Thực ra, ngay cả khi đã biết các kĩ thuật phát hiện lỗi vẫn khó ngăn chặn hoàn toàn các lỗi trong một dự án game lớn vì nó được phát triển bởi một đội ngũ phát triển đông người làm việc liên tục với cường độ cao trong một thời gian dài. Sơ sót xảy ra vẫn là điều khó tránh khỏi. Lập trình viên dành 40% thời gian viết mã 60% thời gian sửa lỗi là vì thế. Do đó kĩ năng debug tùy thuộc rất nhiều vào tích lũy kinh nghiệm thực tế chứ không phải là thứ có thể tích lũy được qua một quyển sách hay bài báo kĩ thuật nào

BÀI 15: AUTO_PTR, MUTABLE, VOLATILE VÀ ĐÁNH GIÁ TỐC ĐỘ CHƯƠNG TRÌNH

auto_ptr

Trong thư viện <memory> có định nghĩa lớp auto_ptr (nghĩa là con trỏ cấp phát và hủy bỏ vùng nhớ tự động) để giải quyết vấn đề rò rỉ bộ nhớ (tuy vậy vẫn có phiền toái, do đó lập trình viên tự cấp phát và giải phóng bộ nhớ vẫn là lựa chọn được khuyến khích hơn)

Trong ví dụ dưới đây, p trỏ đến a (gọi là p sở hữu a) Bạn không cần gọi delete a Khi chương trình kết thúc, destructor của p được gọi, p sẽ bị hủy, nó sẽ tự động huỷ luôn a cho bạn. Đó là mục đích của auto_ptr, bạn không phải lo về leak memory

CODE

```
#include<memory>
class MyClass{
    int data;
public:
    MyClass(int data):data(data){}
    friend ostream& operator<<(ostream& os,const MyClass& p)
        {os<<p.data<<endl;return os;}
};
int main(){
    MyClass* a = new MyClass(5);
    auto_ptr<MyClass> p(a);
    cout<<*p;
    return 0;
}
```

Dùng con trỏ bình thường thì có thể gây leak memory trong ví dụ sau

CODE

```
try{
    Person *p = new Person; (*p).func();delete p;
}catch(...)
```

Dùng auto_ptr thì không lo việc ấy nữa

CODE

```
try{
    auto_ptr<Person> p(new Person); (*p).func();
}catch(...)
```

Quá tuyệt phải không ? Không hẳn thế, bản thân auto_ptr cũng có nhiều rắc rối khác. Cũng cái ví dụ trên, ta sửa lại một chút. Lần này p sẽ chuyển quyền sở hữu a cho p2. Lần này sẽ sinh ra lỗi, vì p2 trỏ đến vùng nhớ, chứ không phải p. Sau khi chuyển a cho p2 sở hữu, lúc này p chẳng sở hữu cái gì cả (khỉ thật, lúc này p đang trỏ đến cái gì ? ai mà biết)

CODE

```
class MyClass{
    int data;
public:
    MyClass(int data):data(data){}
    friend ostream& operator<<(ostream& os,const MyClass& p)
        {os<<p.data<<endl;return os;}
};
int main(){
    MyClass* a = new MyClass(5);
    auto_ptr<MyClass> p(a);
    cout<<*p;
    auto_ptr<MyClass> p2=p;
    cout<<*p;
```

```
    return 0;  
}
```

const auto_ptr không thể chuyển quyền sở hữu được nữa, ví dụ sau là không hợp lệ

CODE

```
MyClass* a = new MyClass(5);  
const auto_ptr<MyClass> p(a);  
auto_ptr<MyClass> p2=p;
```

Rắc rối thứ hai đó là auto_ptr không được dùng với cấu trúc bộ nhớ động, như mảng hay các bộ lưu trữ của STL như vector, list

CODE

```
int* a = new int[5];  
auto_ptr<int> p(a);
```

lí do là vì khi destructor của p được gọi, nó sẽ gọi delete a, chứ không phải delete [] a

Với các bộ lưu trữ của STL như vector, list, còn lí do là khi đưa phần tử vào, các bộ lưu trữ này chỉ sao chép giá trị của phần tử gốc và sau đó làm việc với các bản sao chép. Trong khi với auto_ptr, các bản sao chép này là KHÔNG giống nhau.

Do đó tuyệt đối không bao giờ dùng (dù chẳng thấy báo lỗi gì cả)

CODE

```
vector<auto_ptr<int> > v;
```

auto_ptr có một vài hàm tiện ích

hàm reset

p đã trả đến a rồi, bây giờ ta trả p đến b, thì vùng nhớ do a trả đến sẽ bị phá hủy

CODE

```
MyClass* a = new MyClass(5);  
cout<<*a;  
auto_ptr<MyClass> p(a);  
MyClass* b = new MyClass(7);  
p = new auto_ptr<MyClass>(b);  
cout<<*p;  
cout<<*a;
```

Ta có thể làm tương tự như vậy bằng hàm reset, vùng nhớ do a trả đến cũng sẽ bị phá hủy

CODE

```
MyClass* a = new MyClass(5);  
cout<<*a;  
auto_ptr<MyClass> p(a);  
MyClass* b = new MyClass(7);  
p.reset(b);  
cout<<*p;  
cout<<*a;
```

hàm get

Hàm get trả về vùng nhớ đã do auto_ptr sở hữu

CODE

```
Thay vì cout<<*p bạn có thể dùng cout<<*(p.get())
```

Bạn có thể dùng hàm get để kiểm tra xem vùng nhớ do auto_ptr trả đến có hợp lệ hay không

Tuy vậy không thể dùng hàm get như sau

CODE

```
auto_ptr<Person> p(a);
auto_ptr<Person> p2(p.get());
```

vì p vẫn còn quyền sở hữu a và p2 không thể chiếm lấy a được

hàm release

Hàm release y như hàm get thêm nữa là auto_ptr từ bỏ sẽ quyền sở hữu vùng nhớ

Khi đó vẫn để ở trên với hàm get đã được giải quyết

CODE

```
auto_ptr<Person> p(a);
auto_ptr<Person> p2(p.release());
```

vì p từ bỏ quyền sở hữu a nên p2 có thể chiếm lấy a

mutable

Trong một số trường hợp, chúng ta cần một biến số const có thể thay đổi giá trị.

Ví dụ chúng ta cần thay đổi giá trị của a bằng hàm affect

CODE

```
class MyClass{
public:
    int a;
    MyClass(int a):a(a){}
    int affect() const{
        return a++; //xuat ra roi moi thuc hien phep cong
    }
};
```

Trong trường hợp này const_cast là một giải pháp hết sức tránh, const_cast không đảm bảo nó bỏ đi const với những object được khai báo const, do đó có thể gây ra lỗi không lường được, ví dụ

CODE

```
class MyClass{
public:
    int a;
    MyClass(int a):a(a){}
    int affect() const{
        MyClass* mc = const_cast<MyClass*>(this);
        return (*mc).a++;
    }
};
int main(){
    const MyClass m(6);
    cout<<m.affect()<<endl;
    return 0;
}
```

Trong trường hợp đó, mutable là lựa chọn thích hợp. mutable gần giống như "không thể là const" Một data member của một const object được khai báo mutable có thể thay đổi giá trị

CODE

```
class MyClass{
public:
    mutable int a;
    MyClass(int a):a(a){}
    int affect() const{
        return a++;
    }
};
```

```

};

int main() {
    MyClass m(6);
    cout<<m.affect()<<endl;
    cout<<m.a<<endl;
    const MyClass m2(17);
    cout<<m2.affect()<<endl;
    cout<<m2.a<<endl;
    return 0;
}

```

volatile

Khi bạn lập trình với các game chạy đa luồng, một biến được sử dụng bởi nhiều luồng khác nhau, mà mỗi luồng không thể biết được biến này sẽ được luồng khác thay đổi giá trị như thế nào. Một biến như vậy phải được khai báo là volatile, tức là những biến mà giá trị có thể bị thay đổi bất cứ lúc nào. Trong phần cứng thì thường dùng hơn chúng ta.

Chúng ta không học về volatile lúc này

Đánh giá tốc độ chương trình

Đây là phần quan trọng để xác định thời gian chạy và đánh giá tốc độ chương trình của mình có tốt hay không. Với game thì tốc độ chạy chương trình là một trong những ưu tiên. Chẳng ai thích những game chất lượng chỉ ở mức khá nhưng chạy chậm rì so với những game chất lượng tốt hơn nhưng chạy nhanh hơn trên cùng một hệ thống.

Bạn có thể tính thời gian chạy của những thuật toán xử lý đồ họa, AI, etc bạn viết trong game bằng những hàm trong thư viện <ctime>

Đây là thư viện làm việc liên quan đến thời gian của C/C++

Các hàm với time (thời điểm)

Ví dụ sau sẽ in ra thời điểm hiện tại

CODE

```

#include <ctime>

int main(int argc,char** argv) {
    time_t currenttime;
    time(&currenttime);
    tm* timeinfo;
    timeinfo = localtime(&currenttime);
    char* time = asctime(timeinfo);
    cout<<time<<endl;
    return 0;
}

```

Giải thích:

time_t: (time type) (kiểu thời điểm) là kiểu dữ liệu lưu trữ thời điểm tính theo giây bắt đầu từ 0 giờ 0 phút 0 giây ngày 1 tháng 1 năm 1970

time(): hàm trả về kiểu time_t thời điểm hiện tại (current time)

tm: cấu trúc lưu thời gian, bao gồm giây, phút, giờ, ngày, tháng, năm

localtime(): hàm chuyển kiểu time_t về kiểu tm

asctime(): hàm chuyển kiểu tm về kiểu char*

Một số hàm khác

ctime(): hàm chuyển kiểu time_t về kiểu char*

mktime(): hàm chuyển kiểu time_t về kiểu tm

difftime(): tính sự khác biệt về thời gian theo giây giữa hai time_t, trả về double

Ví dụ sau dùng difftime để tính sự khác biệt về thời gian theo giây với do something là chương trình của bạn

CODE

```

int main(int argc,char** argv) {
    time_t time1, time2;
    time(&time1);

```

```

//do something
time(&time2);
cout<<difftime(time2,time1)<<endl;
return 0;
}

```

Các hàm với clock (thời khắc)

một khắc: một chút thời gian, một tí xíu thời gian (nhỏ hơn một giây) khắc là một khái niệm thời gian không rõ ràng trong ngôn ngữ nên bạn cũng không cần quan tâm đến một khắc bằng một phần mây của giây làm gì

clock_t: (clock type) (kiểu thời khắc) là kiểu dữ liệu lưu trữ thời khắc

clock(): trả về số lượng thời khắc (clock tick) đã qua kể từ khi chương trình chạy

Có một macro gọi là CLOCKS_PER_SEC trả về số lượng khắc trong một giây (số lượng khắc trong một giây tùy thuộc trình biên dịch và ta không cần quan tâm, một số trình biên dịch để là một ngàn, một số là một triệu)

Ví dụ sau ta sẽ viết hàm wait (chờ tính theo giây) bằng cách dùng clock()

CODE

```

void wait(int seconds){
    clock_t waittime;
    waittime=clock()+seconds*CLOCKS_PER_SEC;
    while(clock()<waittime);
}

int main(int argc,char** argv){
    time_t time1, time2;
    time(&time1);
    wait(3);//chờ 3 giây
    time(&time2);
    cout<<difftime(time2,time1)<<endl;
    return 0;
}

```

seconds*CLOCKS_PER_SEC sẽ tính số lượng khắc cần trải qua trong đủ 3 giây và vòng lặp while của bạn chỉ cần chạy trong đủ số lượng khắc đó

Ngoài ra bạn cũng có thể tính số khắc đã trải qua (sự khác biệt về thời gian theo khắc) với do something là chương trình của bạn

CODE

```

int main(int argc,char** argv){
    clock_t begin_clock = clock();
    //do something
    clock_t end_clock = clock();
    cout<<end_clock-begin_clock<<endl;
    return 0;
}

```

Bây giờ bạn đã có thể dùng <ctime> để tính toán thời gian chương trình của bạn thực thi và so sánh thời gian thực hiện những thuật toán của bạn, xem cái nào nhanh cái nào chậm theo giây hoặc theo khắc. Còn một giải pháp khác chính xác hơn là tính toán dựa trên chính tốc độ của CPU nhưng mình sẽ không trình bày vì nó đụng đến hợp ngữ. Giải pháp này tuy không hoàn toàn chính xác vì còn có sai số vì tùy theo nhiều yếu tố khác nữa nhưng như vậy cũng đủ dùng vì sai số không đáng kể. ở mức chấp nhận được.

Những phần sau đã bị cắt: smart pointer, garbage collector và inline assembly. Các bạn có thể tự tìm hiểu thêm nếu muốn.

This post has been edited by **vietgameprogramming**