* 姓名：夏学飞
* 学号：SA22218194
* 专业：软件工程
* 学院：先进技术研究院

C++课程第一次作业-头文件代码注解

一、头文件以及防止重复编译

```
#ifndef H112

#define H112 251113L

#include<iostream>
#include<iomanip>
#include<fstream>
#include<sstream>
#include<cmath>
#include<cstdlib>
#include<string>
#include<list>
#include <forward_list>
#include<vector>
#include<unordered_map>
#include<algorithm>
#include <array>
#include <regex>
#include<random>
#include<stdexcept>

//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------

typedef long Unicode;

//-----------------------------------------------------------------------------
```

- **#ifndef H112的使用**
  由于函数的声明和定义一般都是分开存放的，所以都是分开进行编译，那么就有头文件或者声明重复编译的问题,在这里，我们使用ifndef（条件编译）来解决这个问题

```
#ifndef H112

#define H112 251113L

// 若干代码行 C++一些头文件

#endif //H112  群里面发的头文件 这一行语句貌似没有
```

#ifndef H112 先测试H112是否被宏定义过，然后如果在前面没有被宏定义过，则条件指示符的值为真，那么从#ifndef到#endif之间的所有语句都被包含进来进行编译处理，相反的，如果#ifndef指示符的值是假的，那么它与#endif指示符之间的所有代码行编译时都被忽略（在这里包括各种C++头文件），也就是说条件指示符#ifndef的最主要目的就是防止头文件的重复包含和编译

- **引入的头文件**

代码中引入这些头文件，编译器会先把这些头文件iostream中的所有内容copy到#include位置，再进行编译

```
#include<iostream> // C++标准输入输出头文件 input output stream的简写
#include<iomanip>// io表示输入输出，manip是Manipulator的缩写，该头文件主要是对cin,cout进行操作运算子，
#include<fstream>// C++ stl中文件操作的集合，包含所有常用的文件操作 file stream
#include<sstream>// 该标准头文件包含了ostringstream、istringstream、stringstream这三个类，要使用他们
#include<cmath>// C++数值计算的函数
#include<cstdlib>// 用来调用内存动态分配函数和随机数函数
#include<string>// C++的字符容器，内含string类，包含若干字符串处理函数
#include<list>// C++标准库中的序列容器，允许在容器内的任何位置插入、提取元素
#include <forward_list>// forward_list是一个由元素组成的单向链表，提供了O(1)复杂度的元素插入，不支持快
#include<vector>// vector是C++标准模板库中的一种容器，可以存放各种类型的对象，是一种存放任意类型的动态数
#include<unordered_map>// unordered_map是一个将key和value关联起来的容器，可以根据key查找对应的value，
#include<algorithm>// C++标准模板库(STL)头文件之一，提供了大量基于迭代器的非成员模板函数
#include <array>// 该头文件定义了固定大小的数组容器array，除了容器的元素不会保存任何其他的数据，大小固定
#include <regex>// C++11的正则表达式库
#include<random>// 随机数生成器、分布类
#include<stdexcept>// 标准异常库 出现错误的情况下一般都会选择抛出std::exception的派生类对象
```

- **typedef long Unicode;**

使用typedef关键字，对一些变量类型或者声明定义别名，即为指定类型long起别名Unicode

- **using namespace std;**

C++的STL都是定义在std命名空间中，using namespace std语句就是为了提前声明要引用的命名空间，这样在引用命名空间就不需要添加命名空间前缀

二、中间代码解释

2.1 标准输出流转换成字符串输出模板

```
template<class T> string to_string(const T& t)
{
        ostringstream os;
        os << t;
        return os.str();
}
```

template 是C++模板定义的关键词，T是 代表任意类型的变量，在定义好函数模板之后，在编译的时候，编译器会根据传入的to_string函数的参数变量类型，自动生成对应参数变量类型的to_string函数。

这里将标注输出流中的数据转换成字符串类型

2.2 结构体Range_error以及抛出异常

```
struct Range_error : out_of_range {     // enhanced vector range error reporting
        int index;
        Range_error(int i) :out_of_range("Range error: " + to_string(i)), index(i) { }
};
```

定义结构体Range_error,动态数组vector范围溢出的时候进行报错，out_of_range异常的含义表示一个参数值不在允许的范围之内

2.3 初始化Vector以及重载Vector数组下标运算符[]

```
// trivially range-checked vector (no iterator checking):
template< class T> struct Vector : public std::vector<T> {
        using size_type = typename std::vector<T>::size_type;

#ifdef _MSC_VER
        // microsoft doesn't yet support C++11 inheriting constructors
        Vector() { }
        explicit Vector(size_type n) :std::vector<T>(n) {}
        Vector(size_type n, const T& v) :std::vector<T>(n, v) {}

        template <class I>
        Vector(I first, I last) : std::vector<T>(first, last) {}
        Vector(initializer_list<T> list) : std::vector<T>(list) {}
#else
        using std::vector<T>::vector;   // inheriting constructor
#endif

        T& operator[](unsigned int i) // rather than return at(i);
        {
                if (i<0 || this->size() <= i) throw Range_error(i);
                return std::vector<T>::operator[](i);
        }
        const T& operator[](unsigned int i) const
        {
                if (i<0 || this->size() <= i) throw Range_error(i);
                return std::vector<T>::operator[](i);
        }
};
```

首先使用typename重定义vector::size_type为size_type,接着判断_MSC_VER是否被编译过，没有编译过，执行Vector()的各种重载构造函数，C++中的explicit关键字只能用于修饰只有一个参数的类构造函数, 它的作用是表明该构造函数是显示的, 而非隐式的。Vector的各种构造函数参数提供了不同的构造方法。接着重载数组下标运算符[],让他可以选取指定类型的vector元素。

2.4 重载[]运算符

```
// disgusting macro hack to get a range checked vector:
#define vector Vector

// trivially range-checked string (no iterator checking):
struct String : std::string {
        using size_type = std::string::size_type;
        //      using string::string;

        char& operator[](unsigned int i) // rather than return at(i);
        {
                if (i<0 || size() <= i) throw Range_error(i);
                return std::string::operator[](i);
        }

        const char& operator[](unsigned int i) const
        {
                if (i<0 || size() <= i) throw Range_error(i);
                return std::string::operator[](i);
        }
};
```

这里重载了[]数组下标运算符，首先判断下标是否合法，不合法抛出异常，合法的话，返回指定下标的字符串

2.5 hash函数能处理string参数

```
namespace std {

        template<> struct hash<String>
        {
                size_t operator()(const String& s) const
                {
                        return hash<std::string>()(s);
                }
        };

} // of namespace std
```

hash函数支持char,int long,这里可以处理string参数

2.6 退出结构体 抛出异常 运行超时

```
struct Exit : runtime_error {
        Exit() : runtime_error("Exit") {}
};
```

定义Exit退出结构体，runtime_error表示难以被预先检测的异常

2.7 内联函数以及error函数重载

```
// error() simply disguises throws:
inline void error(const string& s)
{
        throw runtime_error(s);
}

inline void error(const string& s, const string& s2)
{
        error(s + s2);
}

inline void error(const string& s, int i)
{
        ostringstream os;
        os << s << ": " << i;
        error(os.str());
}
```

C++函数重载通常用一个函数名，命名一组功能相似的函数，这样做减少了函数的数量，要求：函数名称必须相同，参数列表必须不同，函数的返回类型可以相同也可以不相同，仅仅返回类型不同不足以成为函数的重载。

这里三个error函数都是同样的函数名，但是参数类型不同，函数体也不同，第一个error该函数抛出一个runtime_error异常，第二个error函数先拼接两个字符串，然后报错，第三个error函数使用标准输出流输出异常。

使用inline来声明三个重载的error函数为内联函数，当编译器处理调用内联函数的语句的时候，不会将该语句编译成函数调用的指令，而是将内联函数的代码直接插入调用语句处，所以执行的时候顺序执行，不会进行跳转执行

2.8 static_cast强制转换类型

```
template<class T> char* as_bytes(T& i)  // needed for binary I/O
{
        void* addr = &i;        // get the address of the first byte
                                                // of memory used to store the object
        return static_cast<char*>(addr); // treat that memory as bytes
}
```

这里定义了一个函数模板，函数返回的是char* 类型，参数选择引用的方式进行传参，先将i的地址取出来 传入一个指针addr，由于采用函数模板的方式，参数的类型不是很确定，所以，先定义成void *指针，然后采用static_cast函数将addr强制转换成char *类型指针

2.9 内联函数keep_window_open重载

```
inline void keep_window_open()
{
        cin.clear();
        cout << "Please enter a character to exit\n";
        char ch;
        cin >> ch;
        return;
}

inline void keep_window_open(string s)
{
        if (s == "") return;
        cin.clear();
        cin.ignore(120, '\n');
        for (;;) {
                cout << "Please enter " << s << " to exit\n";
                string ss;
                while (cin >> ss && ss != s)
                        cout << "Please enter " << s << " to exit\n";
                return;
        }
}
```

第一个内联函数keep_window_open()没有参数，首先使用标准输入流进行清屏，然后输出一条语句，之后输入一个字符，结束

第二个内联函数keep_window_open(string s)传入参数 string s，首先判断字符串是不是空，然后不是空，直接清屏，cin.ignore(120,'\n')表示用来清除以回车结束的输入缓冲区的内容，消除上一次输入对下一次输入的影响。然后使用for循环，不停的输入字符串ss 同时将ss和参数s进行比较，直到相等的时候退出。

2.10 内联函数simple_error输出错误

```
// error function to be used (only) until error() is introduced in Chapter 5:
inline void simple_error(string s)      // write ``error: s and exit program
{
        cerr << "error: " << s << '\n';
        keep_window_open();             // for some Windows environments
        exit(1);
}

// make std::min() and std::max() accessible on systems with antisocial macros:
#undef min
#undef max
```

使用内联函数simple_error输出错误，同时调用keep_window_open()函数,#undef min和#undef max是为了取消宏定义，防止max和min的宏定义导致max和min的参数模板之间发生冲突。

2.11 函数模板-判断是否精度丢失

```
// run-time checked narrowing cast (type conversion). See ???.
template<class R, class A> R narrow_cast(const A& a)
{
        R r = R(a);
        if (A(r) != a) error(string("info loss"));
        return r;
}
```

定义一个函数模板，参数有两个R,A, 函数名narrow_cast，函数返回类型R，先将A类型的参数a强制转换成类型R，然后再强制转换成类型A，和原来的传入参数进行比较，如果不相等说明发生了精度的丢失

2.12 获取随机数——内联函数重载randint

```
// random number generators. See 24.7.

inline int randint(int min, int max) {
    static default_random_engine ran;
    return uniform_int_distribution<>{min, max}(ran);
}

inline int randint(int max) { return randint(0, max); }

//inline double sqrt(int x) { return sqrt(double(x)); } // to match C++0x
```

重载两个内联函数randint,第一个randint传入两个参数max,min,这里使用了静态变量default_random_engine类创建引擎，然后使用uniform_int_distribution获取min和max之间的随机数

第二个randint只是单纯的获取0和max之间的随机数

2.13 函数模板-重载排序函数和查找函数

```
// container algorithms. See 21.9.

template<typename C>
using Value_type = typename C::value_type;

template<typename C>
using Iterator = typename C::iterator;

template<typename C>
// requires Container<C>()
void sort(C& c)
{
        std::sort(c.begin(), c.end());
}

template<typename C, typename Pred>
// requires Container<C>() && Binary_Predicate<Value_type<C>>()
void sort(C& c, Pred p)
{
        std::sort(c.begin(), c.end(), p);
}

template<typename C, typename Val>
// requires Container<C>() && Equality_comparable<C,Val>()
Iterator<C> find(C& c, Val v)
{
        return std::find(c.begin(), c.end(), v);
}

template<typename C, typename Pred>
// requires Container<C>() && Predicate<Pred,Value_type<C>>()
Iterator<C> find_if(C& c, Pred p)
{
        return std::find_if(c.begin(), c.end(), p);
}
```

关键字typename被用来做型别之前的标识符号，Value_type = typename C::value_type;这段代码指出value_type是类C中定义的一个型别，因此该语句就是将value_type声明为C::value_type的别名，同时template中的class也可以被替换成typename；using Iterator = typename C::iterator;用法同上

void sort(C& c);该函数调用sort函数进行排序。void sort(C& c, Pred p)使用Predicate 定义的p容器进行排序

Iterator find(C& c, Val v);在容器中查找指定元素，find_if(C& c, Pred p)使用Predicate 定义的p容器进行查找元素