C++11新特性——语法篇

- 允许用 `nullptr` 代替 0 或者 `NULL`
- **Automatic Type Deduction with `auto`** 用`auto`类型可以不需要指定变量类型，可由编译器自动进行推断， eg `auto i = 42` (推荐在type很长或者很发杂的时候用 auto， 比如迭代器)
- **Uniform Initialization** 任何初始化都可以用一种 **one common syntax** 来进行初始化 (使用 **大括号**) eg: `int values[]{1,2,3}` `vector<int> v{2,3,4,5}` `complex<double> c{4.0,3.0}` `int* p{}` 这时候q会被初始化指向 nullptr 实现的原因是背后有 `initializer_list<T>` 和 `array<T,n>`的支持
- **initializer_list** 采用这个可以实现函数参数个数不定，传给参数是 initializer_list 的也必须是一个 `initializer_list` 或者是`{...}`的形式，initializer_list是一个容器
- **explicit** for ctors taking more than one argument。 在 c++ 11 之前 **explicit** 关键字用在构造函数上可以让编译器强制不进行隐式转换(让用户必须显示地调用构造函数)，在c++11之前一般只有一个参数的构造函数才会进行隐式转换。但在c++11之后允许对有多个参数的构造函数也加上 **explicit** 关键词 使得其不进行隐式转换。（**explicit**用的比较少，这里可能表述的不是很准确）
- **rang-based for statement**, 允许使用 `for( decl : coll)`语法， `coll`是一个容器，比如`vector` 或者是 `{1,2,3}`
- **=default, =delete** 原本在c++中 如果我们自行定义了一个`ctor`(构造函数),那么编译器就不会再给我们一个`default ctor`，但是现在如果加上了 `=default` 就可以重新获得编译器的默认构造函数 eg: `Foo()=default`。 `=default` 用在 **big three**函数上（构造 复制构造 析构) `=delete` 表示不想要这个函数，可以用在任何函数上。比如可以用 `=delete`函数放弃编译器的默认复制构造函数（这样的好处是我们显示地拒绝了编译器的默认复制构造函数，不准让别人复制，可以用在**单例设计模式中**）
- **Alias Template(template typedef)** 别名模板，可以给模板指定别名. 看起来很简单，但是可以在实际操作中起到意想不到的效果（详情看实战）
- **Type Alias** 类型别名，可以用替换 typedef
- **noexcept** ,`void foo() noexcept`这个关键字可以保证不丢异常。同时在**noexcept**内可以放置**bool**表达式，表明在什么时候保证不抛出异常，eg:`void foo() noexcept(true);`
- **override** ,这个关键字可以确保在子类的函数中，该函数一定是重写父类的函数。(如果不小心写错了，编译器会报错)
- **final** 关键字，1 写类的时候如果加上了**final**关键字，那么这个类不能被继承，2 如果用**final**修饰虚函数，那么该虚函数不能被**override**
- **decltype** 通过使用**decltype** 这个关键字，可以让编译器找到一个表达式的type。这个关键字事实上类似于**typeof**这类函数的作用，只是现在**decltype**已经被写入标准里了

> By using the new decltype keyword,you can let the compiler find out the type of an expression. This is the realizaton of the other requested typeof feature. However, the existing typeof implementations were inconsistent and incomplete, so c++11 intorduced a new key word.

- **Lambdas** 允许定义**inline的函数，可以被作为参数或者局部对象来使用，**Lambdas** 改变了C++标准库的使用方法。
- **Variadic Templates** 可变模板参数，能够实现模板参数数目可变，类型可变
- **RValue reference** 右值引用，有了右值引用，出现了移动复制构造函数，和移动赋值函数，可以极大地提升stl容器的效率

1. Uniform Initialization

当编译器看到 {t1,t2...tn}的时候会将其构造成为一个 `initializer_list<T>` 然后将其关联置一个 `array<T,n>` 调用函数 （比如构造函数）时这个array内的元素可以被编译器逐一分解传递给函数，但如果函数的参数是一个 `initializer_list<T>`,调用者不能给予数个T参数，然后认为他们会被自动转化为一个 `initializer_list<T>`传入。（如果函数构造函数的参数有一个参数是 `initializer_list<T>` 那么需要将一个`initializer_list<T>`传给它,这里涉及到`initializer_list<T>`的机制)

2. initializer_list< >

example:

```
class P
{
    public :
        p   (int a, int b){
            cout<< "P(int,int), a="<<a<<",b="<< b << endl;
        }   // P1
        P(initializer_list<int> initialist)
        {
            cout<< "P(initializer_list<int>), values= ";
            for(auto i : initialist)
                cout<< i << ' ' ;
            cout<<endl;
        }  //P2
}
P q{1,2}   // 会调用 P2这个函数， 如果没有 P2，编译器会先将 {1,2}组成一个 initializer_list (initializer背后是用array实现的)  然后再将其一个一个分解 最后会调用 P1这个函数
P p{1,2,3}; // 调用 P2

/*
    The initializer_list object refers to the elements of this array without containing them: copy an initializer lis t object produces another object referring o the same underlying elements, not to new copies of them.
*/
```

`initializer_list<T>` 的实现在源码中将一个 指针(迭代器)传递给 `array` 所以 `initializer_list<T>` 可以看成**不含有任何** 元素的 `array`。

`initializer_list` 改变了很多标准库的实现(包括算法库) 所以现在可以这么写了 `max({1,2,3,4}) // 4`

3. range-based for statement

example:

```
// func1
for( int i: {1,2,3,4,5}){
    cout << i <<endl;
}

vector<double> vec;
...
// func2
for (auto elem : vec){
    cout<< elem << endl;
}

// func3
for( auto& elem : vec){
    elem *=3 ;
    // 通过 这种语法+引用可以实现快速赋值,但是部分容器（关联式容器）不能通过迭代器改变其内容, 比如 set \ unordered_set
}

/*
    for( decl : coll){
        statement;
    }
    上述语法等价于
*/
for( auto _pos = coll.begin(), _end = coll.end(); _pos! = _end; ++ _pos){
    decl = *_pos ; // 取出迭代器内容;
    statement;
}
```

4. =default,=delete

在C++中，一个类里面编译器会给类添加默认的 **构造，复制构造，析构** 函数（如果没有自己定义）。

什么类需要自己实现`big three`（构造，复制构造，析构）？当这个类有指针变量的时候。 （涉及深拷贝和浅拷贝）

`=default`: 只能用在 **构造函数，复制构造函数，析构函数** 上，但是注意，如果我们显示定义了一个**复制构造函数** 那么我们不能再使用 `=default` eg:`FOO(const FOO&) = default` 因为这样会导致二义性（编译器不知道用哪一个版本）。 `=default` 可以让我们再自己定义了构造函数之后，还能够拥有编译器提供的默认构造函数（可以省区一些写代码的功夫）

`=delete`: 删除某个函数，可以用来显示地删除 **默认构造函数，默认复制构造函数**, `=delete`可以作用于任何函数。`=delete`可以用在**单例设计模式**中，通过 delete掉默认的复制构造函数，让别人不能够复制（如果在

C++11以前，可能要实现这个功能十分复杂，需要有一个复制构造函数是 **private**的基类，然后让子类继承这个基类来实现，c++中好像有一个 `boost::noncopyable`实现了一个把复制构造函数放在`private`的基类)

5. **Alias Template(template typedef)**

eg:

```
template <typename T>
using Vec = std::vector<T,MyAlloc<T>>;
Vec<int> coll;
// 等价于
std::vector<int,MyAlloc<int>> coll;
```

ps: 使用 **macro**(宏) 和 **typedef** 都无法达到相同的效果 eg: `typedef std::vector<int,MyAlloc<int>> Vec` , 虽然可以直接用Vec但是无法指定参数 eg: `#define Vec<T> template<typename T> std::vector<T,MyAlloc<T>>` `Vec<int>` 等价于 `template<typename int> std::vector<int,MyAlloc<int>>` 很不自然（不像是在定义一个变量）

但是 不能对 **Alias Template** 进行**特化**或者**偏特化**

5.1. alias template 实战

```
//一开始用到了模板的模板这一比较艰涩的c++模板技巧
template<typename T,
         template<class>      // 这里的第二个模板参数表示，这个模板参数也是一个模板，
            class Container       //内层的模板接受一个参数（可省略，在这里就省略了
>
class XCLs
{
    private:
        Container<T> c;
    public:
        XCls(){
            for(long i =0;i<SIZE;++i)
                c.insert(c.end(),T());
            output_static_data(T());
            Container<T> c1(c);
            Container<T> c2(std::move(c));
            c1.swap(c2);
        }
}


XCls<int,vector> c1; //这么调用事实上是错的，因为 vector的模板参数有两个
                     // vector的第二个模板参数有默认值,并且这一个默认值是以第一个模板参数为基础推出来的
                     // 具体可以看一下 vector 的类定义
// 在引入化名模板之后就可以解决上述的问题了
```

```
template<typename T>
ussing Vec = vector<T,allocator<T>>;

// 调用
XCls<int, Vec> c1;
```

6. Type Alias

6.1. typedef 关键词的用法

ps: 关于 typedef的很好的文章: https://www.cnblogs.com/charley_yang/archive/2010/12/15/1907384.html

typedef 可以用来:

- 定义一种类型的别名，可以同时声明指针性多个对象

```
typedef char char_t, *char_p, (*fp)(void); // 声明 char_t 为类型 char 之别名
                                           // char_p 为 char* 之别名
                                           // fp 为 char(*)(void) 之别名
```

- 在旧代码中辅助struct进行使用,在旧代码中使用 struct声明新的对象的时候必须要带上struct 比如`struct tagPOINT1 p1`,但是使用 typedef就可以避免这种情况

```
typedef struct tagPOINT1{
    int x;
    int y;
} POINT;

POINT p1 ; // 使用了typedef后可以这么用，当然在c++中是可以不用加struct的
        // 在C++中可以直接 tagPOINT1 p1;
```

- 用 typedef来定义和平台无关的类型。一个典型的例子就是 c++标准库中的 `size_t`，通过 `typedef` 将 `size_t`作为变成的接口暴露出来，统一用`size_t`进行变成。在有的平台上，可能需要的是`int` 有的可能是 `long`, 那么这时候只需要更改 `size_t`的`typedef` 定义就可以轻松地实现兼容性，`typedef`可以提升兼容性和代码的可移植性

- 为复杂的声明定义一个简单的别名。

```
// 原版
int *(*a[5])(int ,int );  //这里的 a表示函数指针的数组
                          //  a中的每一个成员都是一个类型为 int*(*)(int,int)的函
数指针

// 新版
typedef int* (*pfun)(int,int);
pfun a[5];                //这里的 a 和旧版本的 a 是等价的
```

6.2. Type Alias 用法

- 用法1

```
using func = void(*)(int,int);
// 上面的句子等价于 typedef void(*fun)(int,int)
```

- 用法2：

```
template<typename T>
struct Container{
    using value_type = T;  // 等价于 typedef T value_type;
}
```

6.3. C++复习using的可能用法

- **using-directives** for namespace and **using-declarations** for namespace members.`using namespace std;`,`using std::count;`
- `using-declarations` for class members

```
protected:
    using _Base::M_allocate;  // 后面就可以直接使用 M_allocate这个类型?
    // 这块还不是很理解
```

- **type alias** and **alias template**

7. noexcept

```
void foo() noexcept(true);
```

在上面的代码中，**noexcept**表明了`foo`不会抛出异常，但是如果`foo()`抛出异常了，那么程序就会终止，调用`std:terminate()`，`terminate()`这个函数在默认情况下又会调用`std::abort()`(这个函数会结束程序)

ps: **noexcept**可以用在**移动构造函数**上，(待深入研究，先插个桩)

> You need to inform C++(specially std::vector) that **your move constructor and destructor does not throw** Then the move constructor will be called when the vector **grows**. **If the constructor is not noexept, std::vector can't use it** since then it can't ensure the exception guarantees demanded by the standard. (注意 growable containers （会发生 memory reallocation)只有两种，vector和 deque)

8. override

```
struct Base{
    virtual void vfunc(float){};
};
struct Derrived1:Base{
    virtual void vfunc(int){
        /*
        这个函数，按照原意我打算重写父类的虚函数，
        但是我不小心参数写错了，这时候编译器会认为这个函数是一个新的函数
        而没有实现重写，换言之不会报错
        */
    }
}
struct Derived2:Base{
    virtual void vfunc(int ) override{}
    /*
    这里加上了override关键字
    告诉编译器这一定是重写父类的虚函数，
    但是这里参数列表写错了
    编译器会报错
    */
}
```

9. decltype

eg:

```
map<string,float> coll;
// 新写法
decltype(coll)::value_type elem; // 获取容器中元素的类型，来声明一个新的元素
// 旧写法
map<string,float>::value_type elem;

/*新旧写法的差别在于，新写法
可以用 decltype来获得容器的类型
而旧写法必须明确知道容器类型
*/
```

decltype defines a type equivalent to **the type of an expression** realization of the often requested **typeof** feature

9.1. 三大应用:

- to declare return types

  ```
  template  <typename T1,typename T2>
  auto add(T1 x, T2 y)->decltype(x+y); // 里哟个和lambdas表达式类似的语法声明返回类型
  ```

- use decltype **in metaprogramming**

  ps:关于typename 一篇很好的文章： http://feihu.me/blog/2014/the-origin-and-usage-of-typename/

```
template<typename T>
void test18_decltype(T obj)
{
    // 有了 decltype可以这么用
    typedef typename decltype(obj)::iterator iType;
    /*
    这里必须要加 typename
    应为这里使用了泛型编程，
    typename告诉编译器 T::iterator是一个类型，
    详情参考上面给出的文章
    */
    }
```

- **pass the type of a lambda** 当我们需要把**lambda**组为 **hash function**或者**ordring or sorting criterion** 的时候，我们需要用`decltype`获取lambda的type传递给模板参数，如下

```
auto cmp =[](const Person&p1, const Person& p2){
    return p1.lastname()<p2.lastname();
}
std::set<Person,decltype(cmp)> cool(cmp);
// 不但需要传递cmp对象
// （一定还要传递cmp作为参数否则大概率会报错，因为lambda函数是一个匿名类
//没有默认构造函数），
//还需要在模板指定类型，
//这时候就可以用decltype来获得类型了。
```

10. Lambdas

> A lambda is a definition of functionality that can be defined inside statement and expressions. Thus, you can use as lambda as an **inline finction**，Theminimal lambda function has no parameters and simply does something

eg:

```
auto I = []{
    std::cout<<"hello lambda"<<std::endl;
};
I();
```

10.1. Lambdas的语法

$...multable_{opt}:throwSpec_{opt}:->:retType_{opt}{...}$

ps:带有opt下标表示可选（当然也可不选），如果都没有选可以不写小括号，否则必须写小括号

- $[...]$ **lambda introducer** 可以用来捕获(在lambda函数提内使用)外部变量(**nonstatic outside object**),如果是**Static** (eg:`std::out`)可以直接使用
    - [=] 表示外部变量是通过**传值**的方式传给**lambda**
    - [&] 表示外部变量是通过**传引用**的方式传给**lambda**
    - 传值和传引用的区别只可意会不可言传 `^_^`

    `[=,&y]`表示外部的 y 采用引用方式捕获，其它的所有的外部object采用传值的方式捕获 `[x]` 表示外部的 x 采用传值的方式捕获

```
int id =0 ;
auto f =[id]()mutable{
    std::cout << "id:" << id << std::endl;
    ++id; // OK, 这里只有带了 mutable才能修改 id
}

// 上述代码等价于（不完全等价，辅助理解）
class Functor{
    private:
      int id = 42; // copy of ouside id
    public :
      void operator()(){
          // 重载 ()方法
          std::cout << "id:" << id << std::endl;
           ++id;
      }
};
Functor f;
// 所以 Lambda的 Type相当于一个
// 匿名的函数对象(function object or functor)
```

- $(...)$ **the parantheses for the parameters** 这里面就是像写一般函数一样，放参数
- $multable$ 是否可以修改被捕获的对象,如果是采用**传引用** 捕获的话不用加上**multabe**也可以修改
- $throwSpec$ 异常说明，比如可以加上`noexcept`表明不抛出异常
- $retType$ 指定返回的类型，如果不指定由函数体内的`return`语句自动推导
- ${...}$, 函数体

ps: **c++20**给lambda添加了很多新特性，在**c++20** 中，lambda甚至可以使用模板参数.....待研究，先插个桩 : https://zh.cppreference.com/w/cpp/language/lambda

10.2. Lambda 注意事项

- Lambda 类似与一个函数对象

- Lambda 没有默认构造函数(很多的错误来源，比如在使用很多STL(如set)的时候)
- Lambda 没有赋值操作符
- 在STL中中函数，**Function object** 是一个非常有力的方式来自定义STL算法的部分行为(如比较方式)，但是写**Function object**（详情参照**Lambda 语法**等价于的部分）需要我们写类，有了**Lambda**之后就方便了很多。

11. Variadic Templates

语法范例:

```
// 递归边界，最后一次调用print(args...)
// args...已经没有参数了
void print()
{

}

//  typename...Types 表示一包模板
// 任意个数任意类型的模板
template<typename T, typename...Types>
void print(const T& first Arg,const Types&...args)
{
    cout<<firstArg<<endl;
    print(args...);
}
// 用 variadic templates 可以方便地实现函数递归

/*
    简单地理解记忆:
    ...args 表示封包
    args... 表示解包
    所以上面的参数 const Type& ...args 表示 args是一包
    而 print(args...)表示把
    原本的一包进行解包分成一个参数和
    另外一包
    只是帮助理解
*/
```

ps : 可以用`sizeof...(args)`来获得一包里有几个参数

Variadic Templates, 模板参数可变,模板参数可变有两层含义:

- 参数个数变化(variable number) 利用**参数个数减少的特性**实现递归函数的调用，使用function template完成.
- 参数类型(different type) 利用**参数个数注意减少**导致**参数类型也注意减少**的特性，实现递归继承或者递归复合，用 class template 实现。

ps: 如果参数类型都相同，只想实现参数个数不定的话，可以使用 `initializer_list<T>`

11.1. Variadic Templates 实战

- **example1** 递归函数调用

```
void printX(){};   //递归边界

template<typename T,typename... Types>
void printX(const T& firstArg,const Types& ...args)
{
    cout<<firstArg<<endl;  // 打印第一个
    printX(args...);      // 递归调用 又把 args... 分成 一个 + 剩下的一包。
}
// 调用
printX(1,2,1.5,"hello");
/*
    输出结果:
    1
    2
    1.5
    hello
*/

/*
    对于模板，模板有特化的概念，
    谁更加特化就调用谁
    eg:
    template<typename...Types>
    void printX(const Type&...args){...}
    这个函数可以和上面的函数共存，因为特化的程度不一样
    上面的版本更加特化
*/
```

- **example2** 利用 Variadic Template 模拟 printf

```
//边界条件
void printf(const char* s)
{
    while(*s)
    {
        if(*s =='% && *(++s)!='%)
            thorw std::runtime_error("invalid format string:missing
arguments");
        std::cout<<*s++;
    }
}

template<typename T, typename...Args>
void printf(const char* s, T value, Args ...args)
{
    while(*s){
        if(*s =='%' && *(++s) !='%'){
            std::cout << value;
```

```
                printf(++s,args...) // 这时候 value已经被消耗掉了，args分解成了
一个 value 和 一包 args,如果args没有了会调用上面那个函数
                return ;
            }
            std::cout<<*s++;       //只输出当前字符
        }
        // 字符串扫描完后面还有参数，说明值给的太多了
        throw std::logic_error("extra arguments provided to printf");
    }
```

- **example3** 用 Variadic Templates 实现多参数的 max函数

```
    //http://stackoverflow.com/questions/3634379/variadic-templates
    int maximum(int n)
    {
        return n;
    }

    template<typename...Args>
    int maximum(int n, Args...args )
    {
        return std::max(n,maximum(args...));
    }
```

- **example4** 用不同于一般的方法处理 first,和 last元素

```
    //output operator for tuples
    template<typename...Args>
    ostream& operator<<(ostream& os, const typle<Args...>& t){
        os<<"["
            PRINT_TUPLE<0,sizeof...(Args),Args...>::print(os,t);
        return os <<"]";
    };

    template <int IDX, int MAX ,typename...Args>
    struct PRINT_TYPLE{
        static void print(ostream& os,const tuple<Args...> & t){
            os<< get<IDX>(t) << (IDX+1 == MAX ?"",",");
            PRINT_TUPLE<IDX+1,MAX,Args...>::print(os,t);
        }
    };

    // 边界，模板偏特化,精髓
    template <int MAX,typename ...Args>
    struct PRINT_TUPLE<MAX,MAX,Args...>{
        static void print(std::ostream& os, const tuple<Args...>& t) {}
    };
```

```
/*
一些个人的思考:
模板的递归和普通的递归还是不同的,
模板的递归是在编译时决定的(感觉
递归了几次就会真的编几个函数)因为模板事实上是采用一种liline的方式。
而普通的运行时递归会在运行时压栈。
(回想一下我们是怎么用汇编写递归代码的)
*/
```

ps: 关于模板递归一些个人的思考(不一定对欢迎指正)

- 模板的递归和普通的递归还是不同的，模板的递归是在编译时决定的(感觉递归了几次就会真的编几个函数)因为模板事实上是采用一种liline的方式。
- 而普通的运行时递归会在运行时压栈。 (回想一下我们是怎么用汇编写递归代码的)
- 因此个人认为模板递归的效率应该会比普通递归的效率高。

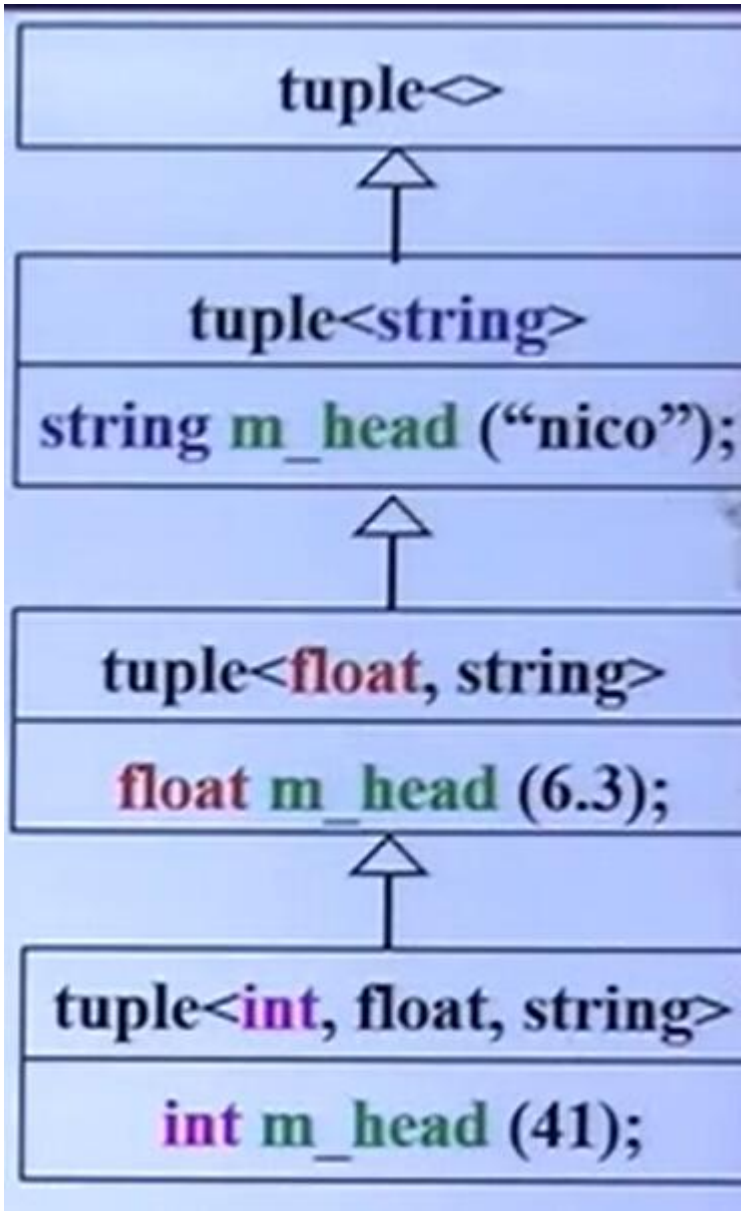
- **example5** 用 Variadic Templates 实现递归继承,tuple实现

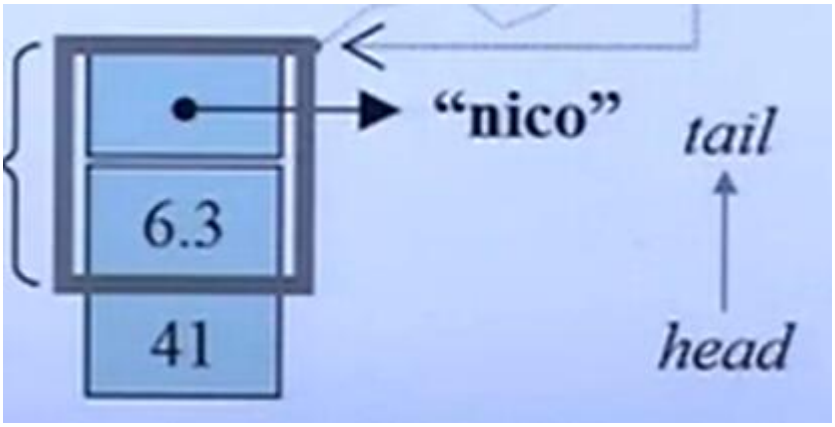
```
#include<iostream>
using namespace std;

template<typename...Args>
class Tuple; //类声明

// 类特化
template<>
class Tuple<> {};

//类定义 + 偏特化
template<typename Head, typename...Tail>
class Tuple<Head, Tail...>: private Tuple<Tail...>
{
    typedef Tuple<Tail...> inherited;
    public:
        Tuple(Head v, Tail...vtail):m_head(v),inherited(vtail...){};//这里
的 inherited,表示调用父类的构造函数
        Head head(){return m_head;};
        inherited& tail(){return *this;};
    protected:
        Head          m_head;
};

int main(){
    Tuple<int,float,string> t = Tuple<int,float,string>(41,6.3,"nico");
    printf("
(%d,%f,%s)\n",t.head(),t.tail().head(),t.tail().tail().head());
}
```

运行结果:

```
chonepieceyb@chonepieceyb-VirtualBox:~/文档/c++learning/c++11/variadicTemplates$ ./test
(1,2.200000,h)
```

上述代码的继承结构:



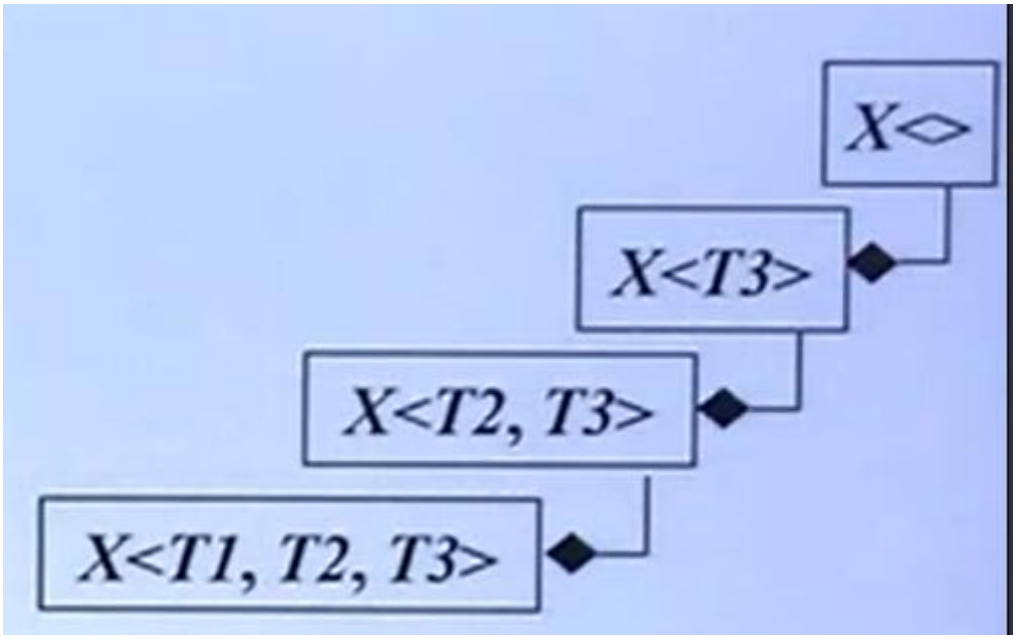
内存空间: 

对于 t , `t.head()`指向41， `t.tail()`指向方框框起来的区域,这里的 `inherited& tail(){return *this; }`挺精髓的。

- example7 用 Variadic Templates 进行递归复合

```
template<typename...Values> class tup; //类声明
template<> class tup<>{};//边界类，特化

// 类定义，偏特化
template<typename Head, typename...Tail>
class tup<Head,...Tail>
{
    typedef tup<Tail...> composited;
    protected:
        composited m_tail; //组合变量
        Head m_head;
    public:
        tup(){}
        tup(Head v, Tail...tail):
        m_tail(vtail),m_head(v){}

        Head head(){return m_head;}
        composited& tail(){return m_tail;} //这里一定要用引用返回
}
//使用方法和例子6相同
```

组合关系:



Rvalue reference 右值引用

> Rvalue references are a **new reference** type introduced in C++0x that help solve the porblem of **unnecessary copying** and enable **pefect forwarding**. When the **right-hand side** of an assignment is an **rvalue**, the the left-hand side object can **steal** resources from the right-hand side object **rather than** performing a separate allocation, thus enable **move semantics**

右值引用可以减少不必要的copy。当赋值操作的右手边是一个**右值**，可以**偷**右手边资源，而不需要非必要的拷贝。

11.2. 左值和右值

- **左值 Lvalue**: 可以出现在 **operator=** 的左边，也就是变量(也可以放在右边)
- **右值 Rvalue**: 只能出现在 **operator=** 的右侧。也就是**临时对象**，临时变量没有名字。

eg 1:

```
int a =9;
int b =4;
a = b; //ok
a+b = 42 // error ,a+b是右值
```

eg2 :

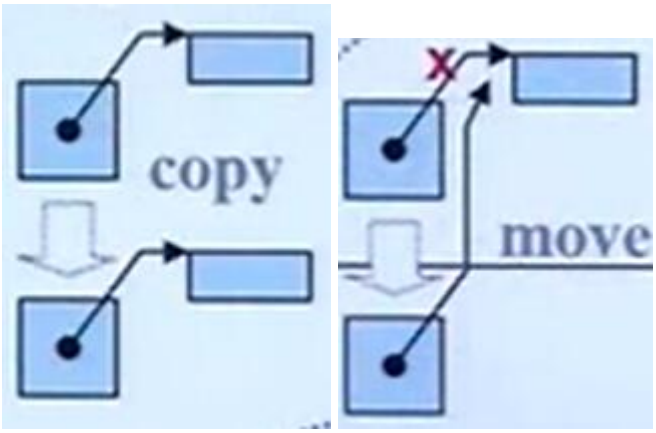
```
int foo(){return 5;}
...
int x = foo() ; // ok x是左值
int* p = &foo(); //Error, 之前对右值无法取地址
fool() =7;         // error ,右值无法赋值
```

11.3. 右值引用

右值引用可以减少不必要的copy。当赋值操作的右手边是一个**右值**，可以**偷**右手边资源，而不需要非必要的拷贝。

所谓的**偷**指的是指针的**浅拷贝**,直接进行指针赋值，进行**move**.

copy 操作 vs move 操作



在这种情况下为什么浅拷贝是被允许的?

- 临时变量本身不能够放在赋值符号的右边，所以临时变量被创建之后其内存里的内容不会被更改的，直接用指针指向临时变量的内存区域十分安全。
- 如果我们能够保证一个变量之后不再使用它，我们可以把左值当成右值(将使用移动构造函数).

```
M c1(c);
M c2(std::move(c1));
```

```
    c1.swap(c2);
  // 必须要保证 之后不再继续使用 c1
```

11.4. 右值引用语法

```
iterator
insert(const_iterator __position, const value_type& x); //普通引用

iterator
insert(const_iterator __position, value_type&& __x); // 右值引用的语法， x是一个临时变量 或者 使用了 std::move
```

11.4.1. 右值引用的问题 unperfect forward

```
void process(int & i){
  cout<<"process(int&):"<<i<<endl;
}
void process(int && i){
  cout<<"process(int&&):"<<i<<endl;
}

void forward(int && i){
  cout<<"fowrard(int &&):"<<i<<",";
  process(i);
}

int a =0;
porcess(a);// 调用 process(int &)
process(1)   // 调用 process(int &&)
process(move(a)); //调用 process(int &&)
fworard(2) ; // fworard(int&&):2, process(int&):2
fworard(move(a)) ; // fworard(int&&):0, process(int&):0
```

如上所述，RValue 经过 forward 再调用两外一个函数就变成了 LValue，原因是经过第一次 forward函数之后，原本的RValue有了参数名称，变成了左值(named object)，所以在forward内部调用了左值的版本.

11.4.2. Perfect Forwarding

> **Perfect forwarding** allows you to write a single function template that takes n arbitrary arguments and forwrds them **transparentlly** to **another arbitrary function**. The **nature** of the argument(modifiable,const,lvalue or **rvalue**) is preserved in this forwarding process

通过标准库提供的 std::forward实现 perfect forward

eg:

```
template<typename T1, typename T2>
void function A(T1&& t1, T2 && t2)
{
  functionB(std::forward<T1>(t1), std::forward<T2>(t2));
}
```

11.5. move-aware class

eg:

```
class MyString
{
  private :
    char* _data;
    size_t _len;
    void __init_data(const char*s){
      _data = new char[_len+1];
      memcpy(_data,s,_len);
      _data[_len] ='\0';
    }
  public:
  //default constructor
  MyString():_data(NULL),_len(0){}
  // constructor
  MyString(const char* p):_len(strlen(p)){
    _init_data(p);
  }
  // copy constructor
  MyString(const MyString & str):_len(str._len){
    _init_data(str.data);
  }
  // move constructor 。移动构造函数必须加上 nonexcept 关键字
  MyString(MyString&& str) noexcept
  :_data(str._data), _len(str._len){
    // 上面的指针赋值是一个浅拷贝
    str._len =0;
    str._data = NULL ;
    //一定把原指针设成NULL
    // 否则可能导致临时变量销毁的时候
    //其析构函数把内存空间也销毁了,
    //这不是我们想要的
    //设置成NULL要配合析构函数, 判断
    //指针是不是NULL再delete
  }

  // copy assignment
  MyString& operator= ( const MyString& str){
    if( this != &str){
      if(_data) delete _data; // 不是空指针才 delete
      len = str._len;
```

```
        _init_data(str.data); //COPY
      }else{

      }
      return *this
    }

    //move assignment
    MyString& operator=(MyString&& str) noexcept{
      // 先判断是不是自我赋值
      if(this !=&str){
        // 判断空然后释放原有的空间
        if(_data) delete _data;
        _len = str._len;
        _data = str._data; //MOVE, 浅拷贝
        // 下面部分同 move ctor, 很重要
        str._len =0;
        str._data =NULL; //配合析构函数,重要
      }else{}
      return *this;
    }

    // dtor
    virtual ~MyString(){
      ++Dtor;
      if(_data){
        delete _data;
      }
    }
  }
```

在移动复制构造函数里的NULL，就是下面这张图中的红叉