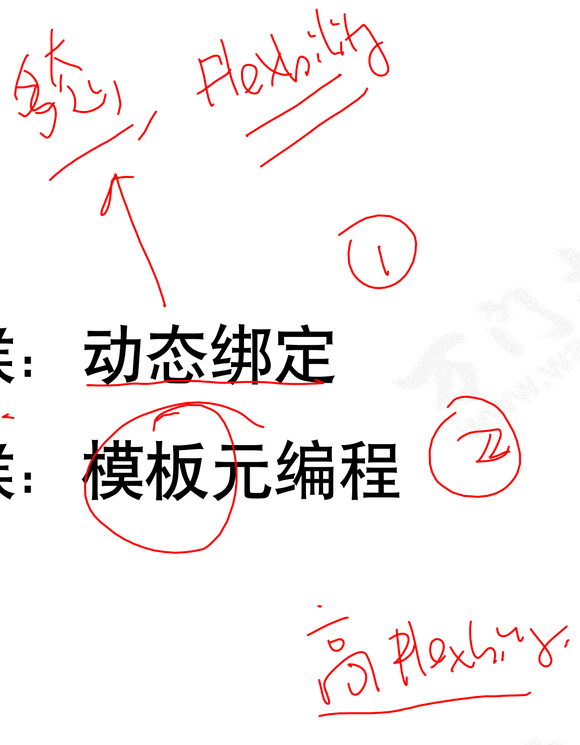
模版和泛型编程

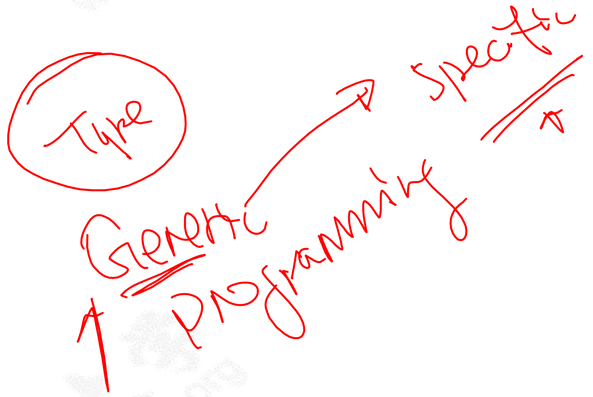
讲师：张嘉星

目标

- 把编译时间的事情放到运行时候：动态绑定
- 把运行时间的事情放在编译时候：模板元编程
- 模板入门



什么是模板



- 模板是泛型编程的基础，泛型编程即以一种独立于任何特定类型的编程
- 模板是创建泛型类或函数的蓝图或公式。库容器，比如迭代器和算法，都是泛型编程的例子，它们都使用了模板的概念
- 每个容器都有一个单一的定义，比如 向量，我们可以定义许多不同类型的向量，比如 vector <int> 或 vector <string>
- 可以使用模板来定义函数和类

STL

Type - Agnostic

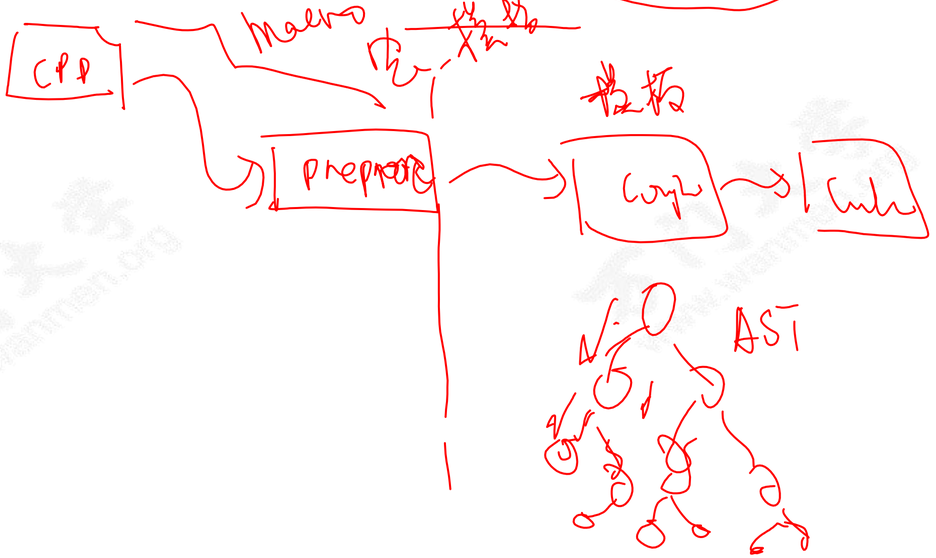
函数模板

```
template <class type>
ret-type func-name(parameter list)
{
...
}
```

```
template <typename T>
inline T const& Max (T const& a, T const& b)
{
  return a < b ? b:a;
}
int main ()
{

  int i = 39;
  int j = 20;
  cout << "Max(i, j): " << Max(i, j) << endl;

  double f1 = 13.5;
  double f2 = 20.7;
  cout << "Max(f1, f2): " << Max(f1, f2) << endl;

  string s1 = "Hello";
  string s2 = "World";
  cout << "Max(s1, s2): " << Max(s1, s2) << endl;

  return 0;
}
```

类模板

- 声明
- 编译展开
- 编译时间
- 可读性难度（readability）




```
template <class type>
class class-name {
.
.
.
}
```

```
// 类模板
template <class T1, class T2>
class A{
  T1 data1;
  T2 data2;
};
```

```
// 函数模板
template <class T>
T max(const T lhs, const T rhs){
  return lhs > rhs ? lhs : rhs;
}
```

全特化

- 通过全特化一个模板，可以对一个特定参数集合自定义当前模板，类模板和函数模板都可以全特化。 全特化的模板参数列表应当是空的，并且应当给出”模板实参”列表：

```
// 全特化类模板
template <>
class A<int, double>{
   int data1;
   double data2;
};

// 函数模板
template <>
int max(const int lhs, const int rhs){
   return lhs > rhs ? lhs : rhs;
}
```

注意类模板的全特化时在类名后给出了”模板实参”，但函数模板的函数名后没有给出”模板实参”。 这是因为编译器根据int max(const int, const int)的函数签名可以推导出来它是T max(const T, const T)的特化。

特化的歧义

- 上述函数模板不需指定”模板实参”是因为编译器可以通过函数签名来推导，但有时这一过程是有歧义的：

```
template <class T>
void f(){ T d; }

template <>
void f(){ int d; }
```

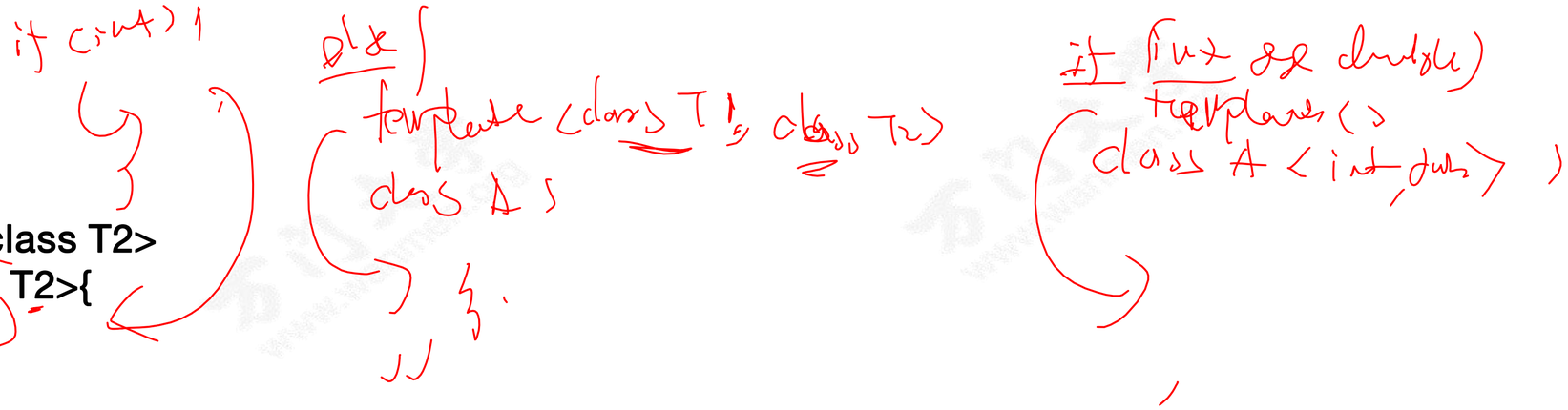
此时编译器不知道f()是从f<T>()特化来的，编译时会有错误：

```
template <class T>
void f(){ T d; }

template <>
void f<int>(){ int d; }
```

偏特化

- 类似于全特化，偏特化也是为了给自定义一个参数集合的模板，但偏特化后的模板需要进一步的实例化才能形成确定的签名。 值得注意的是函数模板不允许偏特化。偏特化也是以template来声明的，需要给出剩余的”模板形参”和必要的”模板实参”。例如：

```
template <class T2>
class A<int, T2>{
    ...
};
```



偏特化

- 函数模板是不允许偏特化的，下面的声明会编译错：

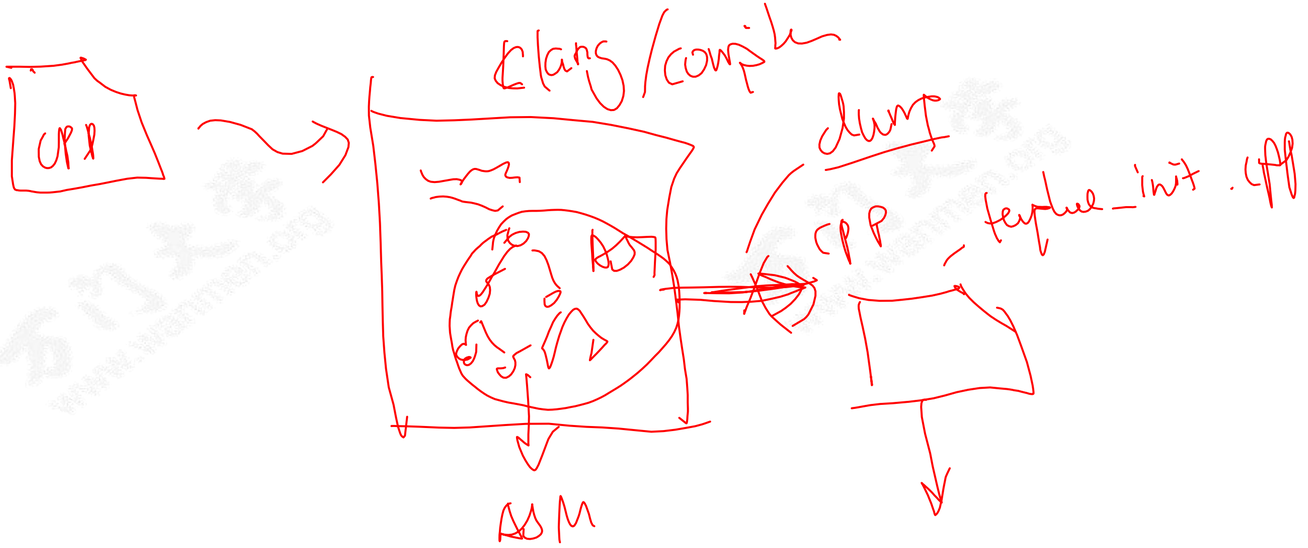
```
template <class T1, class T2>
void f(){}

template <class T2>
void f<int, T2>(){}
```

但函数允许重载，声明另一个函数模板即可替代偏特化的需要：

```
template <class T2>
void f(){}                  // 注意：这里没有"模板实参"
```

偏特化

- 多数情况下函数模板重载就可以完成函数偏特化的需要，一个例外便是std命名空间。std是一个特殊的命名空间，用户可以特化其中的模板，但不允许添加模板



元编程

- 这就是一个接受类型（Template-Args）并返回类型（Type）的函数
- 在模板元编程中我们称呼这个东西为“元函数”
- 别忘了：偏特化（分支语句）和递归（循环语句），我们可以随意自如的处理类型

```
template<typename T>
class Vector {
public:
    using value_type = T;    // A convention
    // ...
};

template<typename C>
using Element_type = typename C::value_type;  // the type of C's elements

template<typename Container>
void algo(Container& c)
{
    Vector<Element_type<Container>> vec;    // keep results here
    // ...
}
```

模板相关：alias：using

- 定义新的模板

```
template<typename Key, typename Value>
class Map {
    // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m;     // m is a Map<string,int>
```

In header <cstddef>:

```
using size_t = unsigned int;
```

模板相关：const 和 constexpr

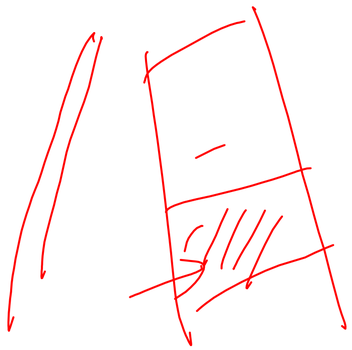
- 编译阶段评估

```
const int dmv = 17;                          // dmv is a named constant
int var = 17;                                // var is not a constant
constexpr double max1 = 1.4*square(dmv);     // OK if square(17) is a constant expression
constexpr double max2 = 1.4*square(var);     // error: var is not a constant expression
const double max3 = 1.4*square(var);         // OK, may be evaluated at runtime
double sum(const vector<double>&);           // sum will not modify its argument
vector<double> v {1.2, 3.4, 4.5};            // v is not a constant
const double s1 = sum(v);                    // OK: evaluated at run time
constexpr double s2 = sum(v);                // error: sum(v) not constant expression

constexpr double square(double x) { return x*x; }
```

模板高级特性

- 没有任何overhead的创建一个模板Buffer类

```
template<typename T, int N>
struct Buffer {
    using value_type = T;
    constexpr int size() { return N; }
    T[N];
    // ...
};

Buffer<char,1024> glob;  // global buffer of characters (statically allocated)

void fct()
{
    Buffer<int,10> buf; // local buffer of integers (on the stack)
    // ...
}
```

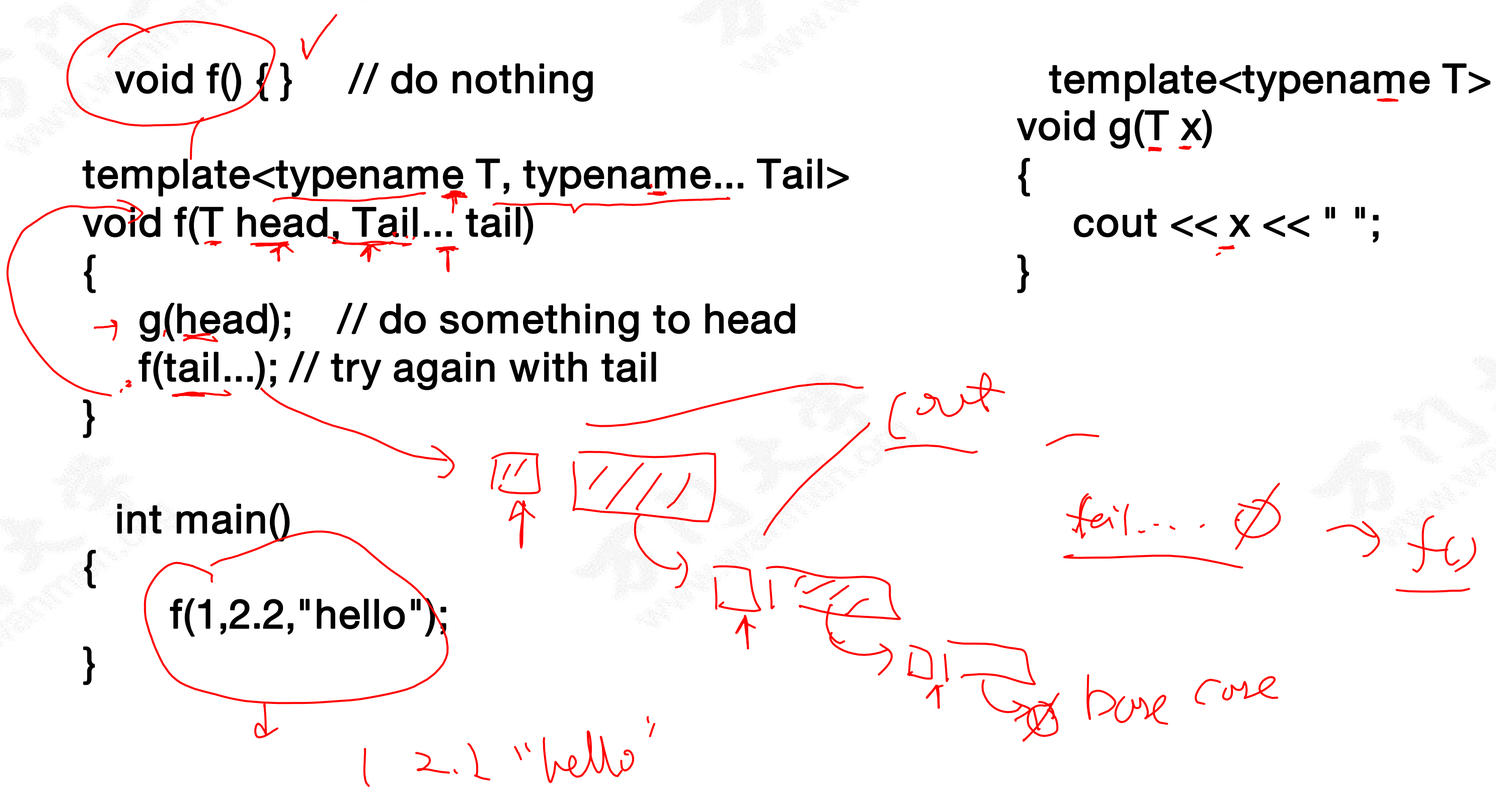


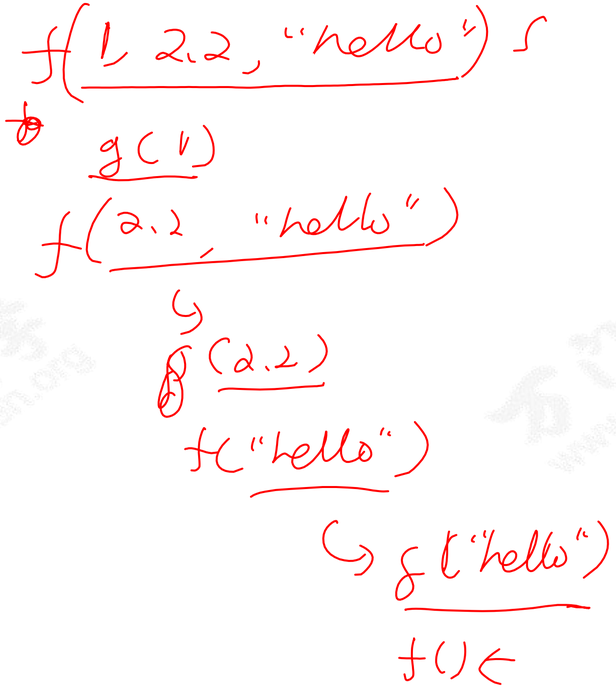
模板高级特性：Variadic Templates

- 可以接受任何长度的输入
- 递归展开

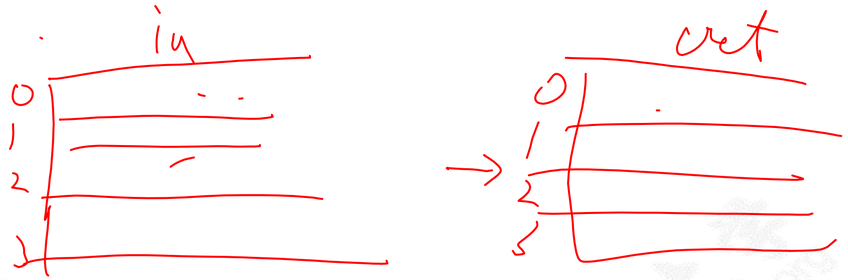
```
void f() { }    // do nothing

template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
    g(head);    // do something to head
    f(tail...); // try again with tail
}

int main()
{
    f(1,2.2,"hello");
}
```

```
template<typename T>
void g(T x)
{
    cout << x << " ";
}
```





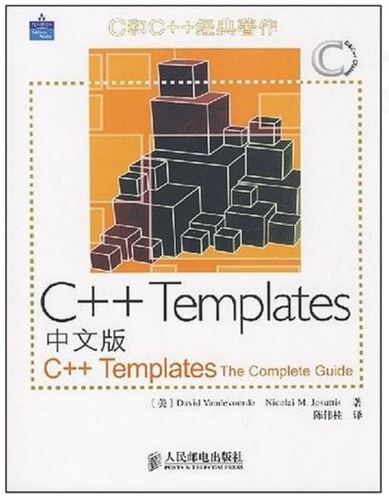
模板应用的例子




```
//float** in; float** out;
for( size_t ch=0 ; ch<channelNum ; ++ch ) {
    for( size_t i=0; i<length ; ++i ) {
        out[ch][i]=in[ch][i];
    }
}
```

模板主要问题

- 编译时间
- 阅读难度
- 调试难度
- 使用难度（C++里的黑魔法）



- 深入学习：https://book.douban.com/subject/2378124/
- 奇妙的工具：
- clang++-7 -Xclang -ast-print -fsyntax-only test.cpp

作业

- 使用模板来编写Fibonacci

```
// 编写模板

int main(){
    std::cout << "Fibonacci<500>:" << Fibonacci<500>::value << std::endl;
}
```

输出：Fibonacci<500>:2171430676560690477

谢谢观看

更多好课，请关注万门大学APP