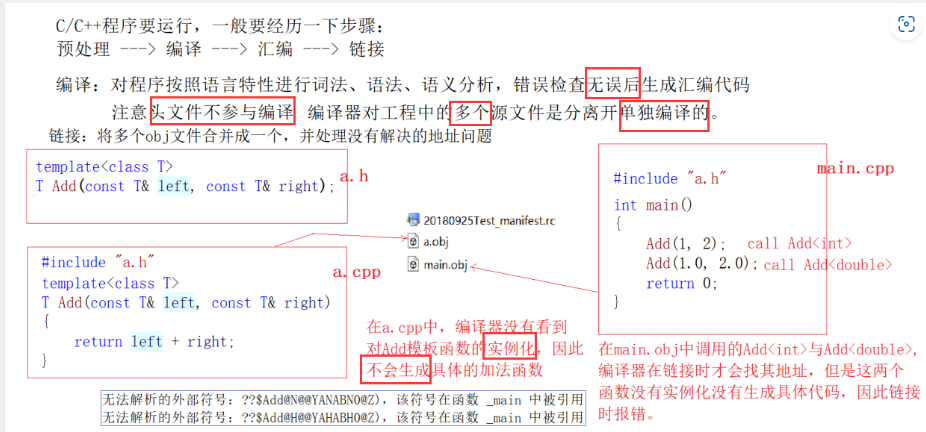
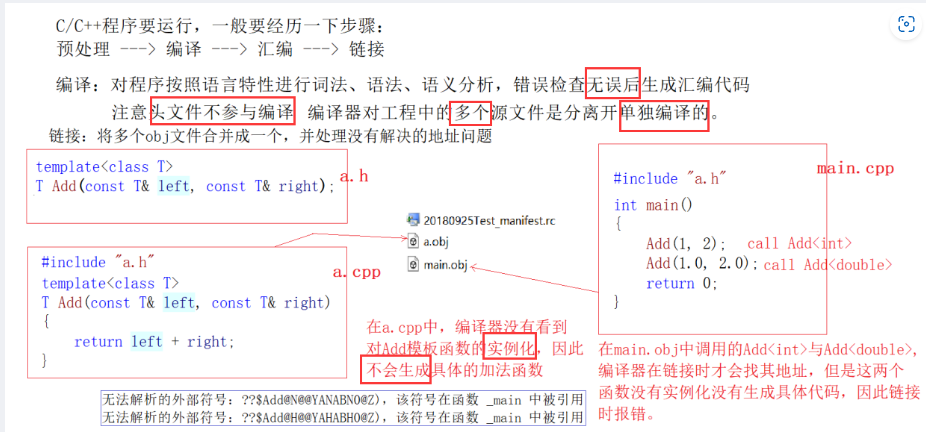
一、函数模板

在编译器编译阶段 ，对于模板函数的使用， 编译器需要根据传入的实参类型来推演生成对应类型 的函数 以供调用。比如： 当用 double 类型使用函数模板时，编译器通过对实参类型的推演，将 T 确定为 double 类型，然后产生一份专门处理 double 类型的代码 ，对于字符类型也是如此。

Tips:声明和实现可以分开写（.h下，不是分开写在.h和.cpp下），但是都需要写上Template



如果分开写到.h和.cpp下会出现链接问题，因为.cpp中，编译器没有看到对模板函数的实例化，因此不会生成具体的函数，因此如果有调用就会在链接时报错

\*模板也是支持缺省的(从右到左)

template<typename p,typename K=int>

template<typename T>

(typename) T Add(T a,T b) //typename可以不写

{

    return a+b;

}

二、类模板

类模板不是具体的类，是编译器根据被实例化的类型生成具体类的模具

Tips:类模板中函数放在类外定义时需要加模板参数列表

template<typename T>

class testclass

{

public:

void init(T Value);

private:

T value;

};

template <typename T>

void testclass<T>::init(T Value)

{

}

模板类型参数分为二种：类型参数和非类型参数（常量值）

template<class T, size\_t N = 10>

注意：

1，浮点数、类对象以及字符串是不允许作为非类型模板参数的

2，非类型的模板参数必须在编译器就能确认结果

三、模板特化

1，函数模板特化（在原模板函数的基础上，针对特殊类型进行特殊化的实现方式）

template<class T> bool IsEqual(T left, T right) { return left == right; }

template<>

bool IsEqual<Date\*>(Date\* left, Date\* right)

{

    return left->\_year == right->\_year

        && left->\_month == right->\_month

        && left->\_day == right->\_day;

}

2，类模板特化

①全特化

template<typename T1,typename T2>

class SpecialClass

{

public:

void show() {std::cout<<"normal one"<<std::endl;}

};

*//全特化(这儿必须加inline..如果不加inline的话，特化后的函数就是一个普通函数)*

template<>

inline void SpecialClass<int,int>::show() {std::cout<<"SpecialClass<int,int>"<<std::endl;};

template<>

class SpecialClass<double,int>

{

public:

void show() {std::cout<<"SpecialClass<double,int>"<<std::endl;}

};

SpecialClass<int,int>\* SpecialOne=new SpecialClass<int,int>();

SpecialOne->show();//SpecialClass<int,int>

SpecialClass<int,double>\* SpecialTwo=new SpecialClass<int,double>();

SpecialTwo->show();//normal one

②偏特化

偏特化是指针对模板参数进一步进行条件限制的特化版本，表现方式分为两种：

1，将部分参数特化

template<typename T2>

class SpecialClass<double,T2>

{

public:

void show(){std::cout<<"Not Fully Speiclal one "<<std::endl;}

};

SpecialClass<double,int>\* SpecialThree=new SpecialClass<double,int>();

SpecialThree->show();//Not Fully Speiclal one

2，参数做更进一步的限制

限制为指针类型

template<typename T1,typename T2>

class SpecialClass<T1\*,T2\*>

{

public:

void show(){std::cout<<"Pointer Special One"<<std::endl;}

};

SpecialClass<double\*,int\*>\* SpecialFour=new SpecialClass<double\*,int\*>();

SpecialFour->show();//Pointer Special One

四、关于模板的一些高级用法

1，enable\_if

本质上只是一个结构体

template <bool \_Test, class \_Ty = void>

struct enable\_if {}; *// no member "type" when !\_Test*

template <class \_Ty>

struct enable\_if<true, \_Ty> { *// type is \_Ty for \_Test*

using type = \_Ty;

};

明显可以看出，有且仅当使用enable\_if<true,\_Ty>这种偏特化时，才会有type这种别名

class person{};

class Dog{};

template<typename T>

struct is\_person

{

static const bool value=false;

};

template<>

struct is\_person<person>

{

static const bool value=true;

};

template<typename T,typename std::enable\_if<is\_person<T>::value,T>::type\* = nullptr>

void func(T t)

{

std::cout<<"enable\_if func"<<std::endl;

}

void func(int i)

{

std::cout<<"normal func"<<std::endl;

}

//main.cpp

func(person()); //enable\_if func

func(1); //normal func

因此参数为person()时，才有type\*这种指针，而当实参为1时，就没有这种指针，就不会使用这种偏特化，就调用func(int i)