类对象和引用

声明变量

我们可以粗略的把对象分为原始对象和类对象两类：

```
//新声明的本地原始对象没有值
int i; //i的值不确定

//新声明的本地类对象值由构造函数决定
string s;//string有构造函数 s初始化为空

//但是所有的指针都是原始对象（因为本质上都是一个东西） 即使指向类对象
string *sp;//做本地变量时sp的值不确定
```

只有CPP11之后才能在定义类时为成员变量设定默认值，这之前只能在构造函数中进行初始化。

动态分配

```
//对原始对象 new/delete 和 malloc/free 可以认为基本没有区别
  //除了new没有空间时抛异常，malloc没有空间时返回NULL
//但是对类对象 new/delete 会调用构造和析构函数而 malloc/free 不会
//两套操作不能混用

class Student{...}

Student *sp = new Student;  //new Student(); 加不加圆括号都一样
Student *ssp = new Student[10];

delete ss;
delete [] ssp;  //注意删除new数组的语法
```

引用

```
//引用可以理解为 为一个**左值** 取别名
typename m, n;
typename &x = m;  //为m取别名x 从此所有对x的操作都是对m的操作
typename &y = 10*2  //不合法 10*2 不能做左值

//引用必须在声明的同时定义 并且一经定义不能改为指代其他变量 这与指针不同
typename &z;    //不合法 不能仅声明不定义 引用不能是NULL

//没有引用的引用 也没有指向引用的指针
typename &_x = x;   //合法 引用的引用的引用……不管套几层 最终都指向同一个本体
typename *px = &x;    //合法 注意引用的指针最终指向本体 &x和&m值相同
typename & & __x = _x;  //不合法 有指针的指针 但是没有引用的引用
```

STL容器

CPP中的容器以STL模板类的形式提供。

- 有序容器：内部按照进入顺序排序
  - vector: varialble array
  - deque: dual-end queue
  - list: double-linked-list
  - array
  - string: char.array
- 无序容器：内部元素按一定规则组织，而不是按照进入容器的顺序
  - map: hash table

Vector

push_back() / emplace_back() 默认是深拷贝。

vector用[ ]取下标时不检查index越界。用下标向不存在的index中放东西可以运行，但是不会影响vector.size()，也不会影响依赖size的遍历等函数。

List

和vector主要区别是内部实现，vector是数组而list是链表。random access多时应该用vector，大量头尾插入删除应该用list。综合场景中两者性能差不多，但是应该优先用vector因为指针会占用额外的空间。

出于链表的实现，empty()的性能远高于count()==0。

Map

map<key_type, value_type> 有两个类型参数，内部实现是哈希表。

map可以用[ ]访问元素，[ ]内是key的值，其不一定是数字，这就为使用提供了方便。

但是map和vector不一样的是

- vector尝试访问不存在的下标可以进行，虽然读出的数据没有意义并且不会影响这个vector
- **map尝试访问不存在的下标时会自动放进去一个对应的元素，key就是尝试访问的key，val是对应类型的默认值。**
    - 所以检查map中某个key是否存在时，应该用count(key) == 0? 而不是尝试访问map[key]。
    - 又因为map是hash，key不允许重复，所以count(key)的值只会是0或1。

构造、析构函数

默认构造、析构函数

```
//当没有构造函数且属性都是public时 对象可以用大括号初始化 相当于一个class
class point_class{
//不标明public显然不能过编译
public:
    int x, y;

    //有对应参数的构造函数时 大括号初始化也会调用构造函数
    //取消下一行注释 下面的语句仍然合法 但是实现的过程从直接拷贝属性变成了构造函数内的内容
    //point_class(int x, int y){};

    //有构造函数但是参数不匹配时 不能使用大括号进行初始化
    //取消下一行注释 不能过编译
    //point_class(){};
}
point_class p = {1, 3}; //合法 直接拷贝给对象的x、y

//所有的类都需要构造和析构函数 编程者不写 编译器也会自动加上一个没有参数什么都不做的构造函数
//让编译器帮我添加构造、析构函数也可以用default关键字显示声明
//不写构造函数、写default、写空构造函数 三者是等价的 析构函数同理
class another_class{
    //another_class() = default;
    //another_class(){};
}
```

运算顺序

```
#include <iostream>

struct X {
    X()  { std::cout << "X::X()" << std::endl; }
    ~X() { std::cout << "X::~X()" << std::endl; }
};

struct Y {
    Y()  { std::cout << "Y::Y()" << std::endl; }
    ~Y() { std::cout << "Y::~Y()" << std::endl; }
};

struct Parent {
    Parent()  { std::cout << "Parent::Parent()" << std::endl;}
    ~Parent() { std::cout << "Parent::~Parent()" << std::endl; }
    X x;
};

struct Child : public Parent {
    Child()  { std::cout << "Child::Child()" << std::endl; }
    ~Child() { std::cout << "Child::~Child()" << std::endl; }
    Y y;
};

int main() {
    Child c;
}
```

创建类的新对象时执行顺序为：构造父类->构造成员变量->自己的构造函数

删除对象时相反：自己的析构函数->析构成员变量->析构父类

上面程序的输出结果必须理解。

继承

三种继承模式

	Public继承	Protected继承	Private继承
Public成员	Public	Protected	Private
Protected成员	Protected	Protected	Private
Private成员	不能继承	不能继承	不能继承

Upcasting与虚函数

CPP允许把子类的对象赋给父类的指针，称为Upcasting（造型）。难点在于，当父类和子类存在同名函数时，Upcasting得来的指针该调用谁。

```
class p {
public:
  int i;
  void f(){
    cout << "p::f()" << endl;
  }
};

class s : public p {
public:
  int i;
  void f(){
    cout << "s::f()" << endl;
  }
};

int main (){
  s s_obj;
  p* p_ptr = &s_obj;

  // 结果还算可以理解 什么类型的对象、引用、指针就调用什么f()
  s_obj.f();  // s::f()
  p_ptr->f(); // p::f()
  // 但是这并不方便使用 upcasting往往是为了用统一的基类指针管理不同的继承类
  // 如果只能像上面那样使用 基类指针就丢失了所有继承类的相关函数特性 不满足多态

  // 如果为父类声明virtual （父类声明了virtual子类可以不声明 但是出于可读性建议加上）
  // 那么无论这个指针是指向基类的还是upcasting得来的 用基类指针总能够调用到原对象的类型对应的函数
```

```
    // 对应的类没有定义这个函数 会使用基类的函数
    // 还可以在基类中定义virtual func_name() = 0;
    // 表明这是一个纯虚函数 意义是基类只提供接口 继承类必须override
}
```

下面是对虚函数的一些补充：

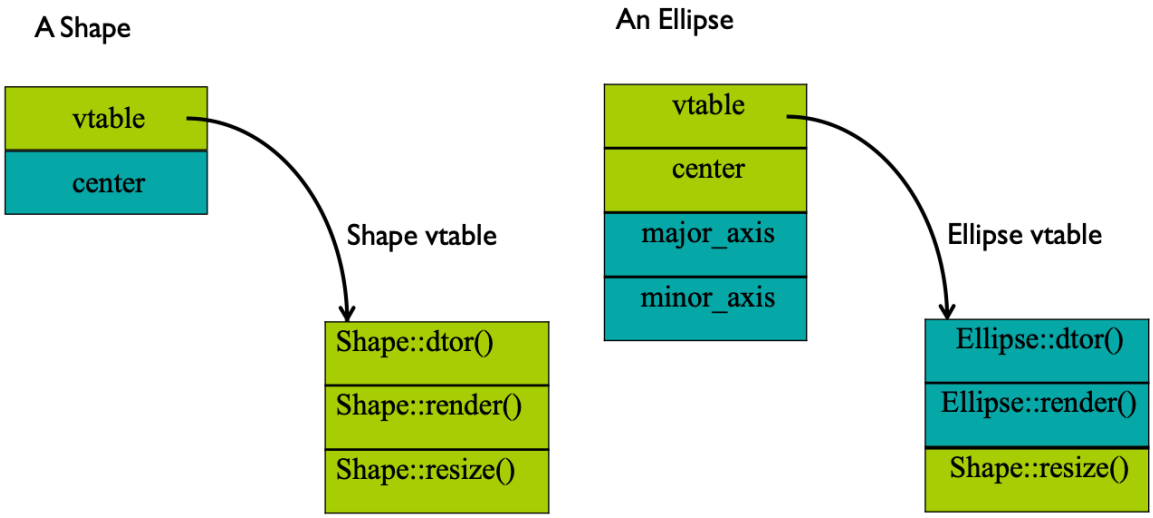
- 是否是const将被视为两个不同的函数
- 构成override的虚函数应该有相同的名称、参数表和返回类型。注意因为存在upcasting，具有继承关系的类的指针和引用被视为同一种返回类型。
- 父类定义了多个相互重载的虚函数，子类override任意一个就会让所有同名重载全部失效。（这个特性只有CPP有，与编译器有关）
- 构造和析构函数当然也可以virtual，并且出于内存安全的考虑，我们应该把所有析构函数定义为虚函数。
- **如果一个类中至少有一个纯虚函数，这个类就是为抽象类**。它的主要作用是为一个类族提供统一的公共接口。
- 抽象类只能做基类，自身不能被实例化，不能作参数类型、函数返回值或显式转换的类型，但是可以声明抽象类的指针和引用以配合upcasting。
- 抽象类的派生类必须给出所有纯虚函数的函数实现（于是不再是抽象类）， 才可以声明自己的对象；反之，这时的派生类仍然是一个抽象类。

虚函数的实现机制

虚函数的实现机制依赖两点：

- 父类对象在子类对象内存的前部
- **v-table**

1. vtable总是定义有virtual函数的类的对象的第一个成员。
2. 当构造继承类成员时，首先构成父类成员，父类中所有声明了virtual的函数指针会被放在vtable中。
3. 完成父类构造开始自己的构造时，自己声明了virtual的函数指针会覆盖vtable中的同名函数。
4. 在upcasting时，vtable不变。
    - 一方面，vtable和来自父类的成员在子类对象内存的前部，所以父类指针可以正常解析内存
    - 另一方面，vtable不随指针类型的退化而退化，所以仍然可以调用到对应子类的virtual函数

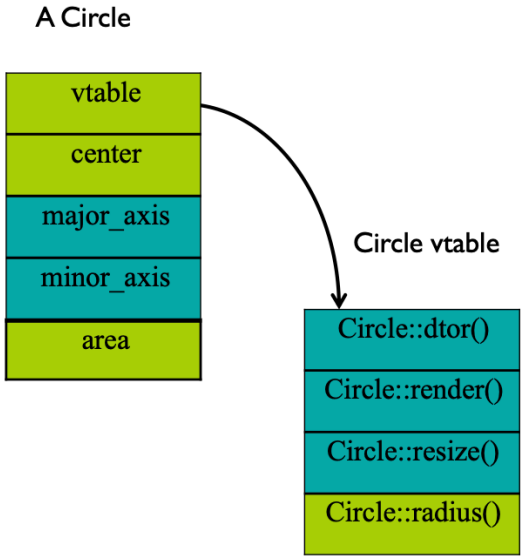



```
class Circle: public Ellipse{

public:
    Circle(float radius);
    virtual void render();
    virtual void resize();
    virtual float radius();

protected:
    float area;

};
```



code & demo

Upcasting和Slice

```
Ellipse* elly = new Ellipse(20F, 40F);
Circle* circ = new Circle(60F);
elly = circ;  // 派生类指针/引用赋给基类指针/引用 是之前介绍的Upcasting
              // vtable的内容不会改变以实现多态
```

```
Ellipse elly(20F, 40F);
Circle circ(60F);
elly = circ;  // 但是将派生类对象赋给基类对象 称为Slice
              // 因为基类的成员总是在派生类对象内存的前部
              // 所以直接截取（Slice）属于基类的成员进行内存拷贝即可
              // 需要注意的是 发生Slice时 编译器会把vtable中的函数指针同步退化
```

友元

在类中声明friend，让别人可以访问这个类中的所有成员（包括public和private）。注意friend声明是由所有者做的，因此只能放在类的声明中，不由使用者控制也不能在所有者声明之外亡羊补牢。

Inline

inline关键字修饰的“函数定义”只是声明。定义是在调用时传入参数后由编译器实时生成的。

因为inline只是声明，所以inline函数的body必须放在头文件里而不能像一般的函数一样将声明和定义分装在h和cpp两处。

递归的函数不能inline。

对象与Const关键字

Const修饰对象和指针

const对象和指针不可被修改的特性是由编译器检查程序逻辑实现的，在物理上与一般的变量没有区别。const变量也不能被非const指针指向，因为非const指针可能试图修改指向的值，这存在逻辑漏洞；但是反过来const指针指向非const变量没问题。

const声明和值分开的情景有两种：函数参数和类的成员。但是你仍然需要构造函数等形式在创建对象时将类的const成员初始化，而不能像非const成员一样先放着不管，用到时再说。

const指针的意义是不能通过这个指针修改其中变量的值，但是可以改变这个指针指向的位置。

Const修饰类的成员函数

现在想象一个具体的困境：你传入一个const对象，尝试直接通过这个对象直接修改成员变量很容易被编译器发现。但是如果通过const对象调用函数呢？const对象接收这并不知道这个函数会不会修改成员变量的值，这就与产生了危险。

所以我们需要对类的成员函数的const修饰符。它实现的本质是将隐藏的this指针加上const。

```cpp
//成员函数声明为const有两个作用
//第一：const函数不能修改成员变量的值 由编译器负责检查
//第二：const对象不能调用非const成员函数

class A{
public:
  int i = 0;

  void non_const_f() {
    i++;  //当然没问题
  }

  void const_f() const {
    i++;  //非法！！
  }
}

int main(){
  const A a;

  a.const_f();       //当然没问题

  a.non_const_f();  //非法！！
  // const对象不能调用非const成员函数
  // 而const成员函数由编译器保证不会修改成员变量
  // 相当于用const对象禁止调用非const成员函数这种方式
  // 间接保证了const对象成员变量的不可修改

  return 0;
}
```

对象与Static关键字

static函数不能同时为const函数。

- 对类内的static变量：
  - **需要在外部加上声明才能使用**
  - 运行时即存在，不属于任何一个对象，或者说所有对象共用
- 对类内的static函数：
  - 不属于任何一个对象，因此只能访问static变量
  - 不能override

下面看例子：

```
class x{
    int data;
    static int s_data;
public:
    x() {}
    void set_data(int i) { data = i; }
    void set_s_data(int i) { s_data = i; }
    void foo() { cout << data; }
    static void s_foo() { cout << s_data; }
}



//这一行非常容易忘记！！！！！
//类内定义的static变量必须在全局区域进行声明
int x::s_data;

int main{

    //B.s_data的输出结果是20
    //因为static变量所有对象共用
    x A, B;
    A.set_s_data(20); A.set_data(20);
    cout << B.s_data;


    //下面通过对象和通过类调用static函数两种写法都能够成立
    //更推荐x::的写法，因为静态变量和函数不依赖特定对象实例，整样写合乎逻辑
    cout << A.s_data;
    cout << x::s_data;
    A.s_foo();
    x::s_foo();

    //如果static void s_foo() { cout << data; }会报错
    //因为static函数不依赖对象，因此也不存在this指针，不能访问成员变量

    //foo()等价于void foo(x *this) { cout << this.data; }
    //可以执行
    A.foo();
    //s_foo()静态函数不会加入this指针，因此如果试图{ cout << data; }
    //实质上是试图{ cout << this.data; }会报错，因为没有this
```

```
    //即使是通过A.s_foo()面向对象的调用方式也没有用，静态函数只能访问静态变量
    A.s_foo();
}
```

函数重载与默认参数

重载和隐式类型转换可以共存，但是隐式类型转换的方式不能有歧义。更复杂的应用见template一章。

默认参数只能从右往左写，也是因为如果默认参数不从右往左会造成参数理解的歧义。

默认参数只能写在函数声明里而不能在函数定义里。因为默认参数的实现原理是编译器帮你填空，与函数本体无关。加不加默认参数函数体本身没有区别。区别仅在于编译的时候编译器帮你做一些事情。

运算符重载

运算符重载既可以写为全局函数，又可以写为成员函数。

```
class Integer{
  int i;
  Integer(int data){
    i = data;
  }
  const int GetVal(){
    return i;
  }
  // 可以写为成员函数 运算符左边的成员this指针形式传入
  // 所以写为成员函数时 单目运算符没有参数 双目运算符只有一个参数
  // 返回值为const的目的是加法的结果不能做左值 否则不符合逻辑
  // 如果没有const 就可以做类似a+b = 7这样没有意义的语句
  // 可以做左值的运算符重载应该返回一个引用
  const Integer operator+(const Integer that){
    return i + that.i;
  }
}

// 也可以写为外部函数 这时单目运算符1个参数 双目运算符2个参数
const Integer operator+(const Integer &l, const Integer &r){
  return Integer(l.GetVal()+r.GetVal());
}

// 取决于写法的不同 调用方式略有区别
```

```
Integer a(10), b(5);
a = a + b;  // 定义为成员函数 a.operator+(b);
            // 定义为外部函数 opreator+(a, b);

// 两种调用方式的区别涉及隐式类型转换
a = b + 5; // 两种写法都可以进行 隐式类型转换
a = 5 + b; // 定义为外部函数时 operator+(5, b); 编译器会帮你尝试从5到Integer的隐式转换
           // 定义为成员函数时 5作为int不能重载operator+ 调用5.operator+(b); 不过编译!

// 因此我们一般认为
// 单目运算符写为成员函数 更符合逻辑也方便封装
// 双目运算符写为外部函数 目的是让编译器帮我们做一些事情 方便使用
// 常常在类中将外部函数重载声明为friend 方便直接取成员做计算
```

++/--因为可以做前缀可以做后缀，在重载时略有区别：

```
Integer& Integer::operator++() {
    this->i += 1;       // increment
    return *this;       // fetch
}
// int argument not used so leave unnamed so
// won't get compiler warnings
Integer Integer::operator++( int ){
    Integer old( *this );   // fetch
    ++(*this);              // increment
    return old;             // return
}

   Integer x(5);
   ++x;
      // calls x.operator++();
   x++;
      // calls x.operator++(0);
```

命名空间

命名空间可以嵌套，例如：

```
namespace mine{
  using namespace A;
}
//A::func()也可以通过mine::func()调用
```

老的C库在CPP工程中编译可能无法连接，这与namespace有关。

深浅拷贝

深浅拷贝只针对Object和Array这样的引用数据类型的：

- 浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存。
- 深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象。

CPP默认的拷贝构造函数和涉及拷贝的表达式都是浅拷贝。

拷贝构造函数

拷贝构造函数是特殊的构造函数，它唯一的参数是自身类的const引用。有三种情况会调用拷贝构造函数：

1. 用一个对象去初始化同类的另一个对象时
2. 函数的参数是一个对象时
3. 函数的返回值是一个对象时

下面看例子：

```
#include <cstring>     // #include <string.h>
#include <iostream>

using namespace std;

class person {
public:
    char *name;
    // 普通构造函数 参数只要不是(const person&)是什么都行
    // 也因此这个类里没有显式定义的拷贝构造函数
    person(const char *s) {
```

```
        name = new char[::strlen(s) + 1];
        ::strcpy(name, s);
    }

    ~person() { delete[] name; }
};

//最好写成(const person &m)原因下面讲
void try_copy_ctor(const person m) { cout << m.name << endl; }

int main() {
    person p("ABC");
    person q = p; // person q(person(p));

    /*
      这里调用了拷贝构造函数！属于情况1:用一个对象初始化另一个同类对象
        注意person q = p; 相当于person q(person(p)); 会调用拷贝构造
        但是{person q; q = p;}不会
    */
    try_copy_ctor(p);
    /*
      这里调用了拷贝构造函数！属于情况2:函数的参数是一个类的对象
      传入参数实际发生的是将实参 拷贝构造 到局部的形参
      情况3:返回一个对象时同理，将局部的参数 拷贝构造 到返回位置
    */

    /*
      这个程序会发生错误，因为 ***默认的拷贝构造是浅拷贝***
      person p("ABC"); 将name指针指向了存有“ABC\0”的内存地址
      因为其中的拷贝都是浅拷贝
      所以main里的q和try_copy_ctor里的p都指向同一个“ABC\0”内存地址

      又因为析构函数要释放name指向的字符串，这个程序中会将同一块空间释放三次:
        1. 退出try_copy_ctor()析构m时
        2. 退出main析构p时
        3. 退出main析构q时
      显然会发生错误
    */
}
```

上面的例子展示了默认的浅拷贝在操作指针时会出现的问题。**当类中有指针、引用、动态链接时，应该为这个类显式的定义构造函数，避免上述问题。同时，对情况2、3，我们并不建议直接将对象作为函数参数，如果需要传参又不希望原先的值被修改，应该用const引用或const指针作为参数。**

应该为person类自行定义深拷贝构造函数，在需要拷贝构造时它会自动被调用，程序可以运行。

```
person::person(const person &n) { // const引用自身类做参数，这是一个拷贝构造函数
    // 重新分配空间并复制内容，这是一个深拷贝
  name = new char[::strlen(n.name) + 1];
    ::strcpy(name, n.name);
}
```

拷贝相关运算符重载

上面的例子里有：

```
person p("ABC");
person q = p;
//line2 等价于 person q(p);
//    等价于 { person q; q = p; }
```

第二种写法 `{ person q; q = p; }` 虽然不涉及拷贝函数，但是并没有解决上面存在的问题，**因为和拷贝相关的运算符默认也是浅拷贝的**。这样写p、q的name仍然指向同一个位置，也仍然会重复释放。

所以**当类中有指针、引用、动态链接时，如果需要用到运算符，也需要特别考虑是否需要重载为深拷贝。**采用第二种写法时应该为person类重载operation=为深拷贝。

```
person &person::operation= (const person &n){
  this.name = new char[::strlen(n.name) + 1];
    ::strcpy(this.name, n.name);
    return *this;
}
```

类型转换与explicit

类的类型转换方法有两种形式：

```
class apple;
```

```cpp
class orange;

class orange {
public:
    orange() {};
    // 这种形式称为转换构造函数，表达orange类型如何接受apple类型对象实例
    orange(apple &A) {};
}

class apple {
public:
    apple() {};
    // 形如 operator type_name() {...}; 没有参数也没有返回类型
    // 这种形式表达了主动进行类型转换的方法 apple类型如何主动转换为orange类型
    operator orange() {};
}

int main () {
  // 众所周知C/C++的编译器会检查变量类型，并在需要时自动进行隐式的类型转换
    // 例如这里将一个浮点数隐式转换为了int类型
    // 这句话显式类型转换的写法为 int i = int(3.14);
    int i = 3.14;

    // 隐式类型转换能够自动进行的条件是：
    // ***存在 唯一的 非explicit的 对应类型转换方法***
    // 下面两行代码不能编译，因为现在有超过一个非explicit的类型转换函数，隐式转换时不知道应该使用哪一个
    apple a;
    orange o = a;

    /*
      explicit关键字的作用是，限制其修饰的类型转换函数不允许被隐式使用
      如果最初定义的两个函数都加上explicit，上面两行代码仍然不能编译
      因为所有的类型转换函数都限制不允许被隐式的使用

      如果只有其中一个加上了explicit关键字，那么上面两行可以编译
      将会使用无explicit关键字的那一个函数进行隐式转换

      一般情况下我们认为隐式类型转换的不好的
      所有类型转换都应该被显式的表达
    */
}

```

```cpp
// explicit与构造函数相结合
struct C{
  explicit C(int) { cout << 'i' << endl; }
  C(double) { cout << "d" << endl; }
}
int main () {
  C c1(7);  // output i
  // 这里的类型是显式的

  C c2 = 7; // output d
  // 注意这样写相当于 C c2( (int/double) 7);
  // 与第一种写法的区别在于这里的类型是隐式的
  // 而int参数的重载是explicit 不允许隐式类型转换 所以只能找double
}
```

Templates

CPP泛型编程的核心。思想是对不同类型的相似操作，相同的部分代码复用，不同的部分作为参数。

模板函数是在编译时实时根据传入的参数类型生成对应函数的，它长得像定义但其实是声明，定义在编译中实时生成，这与inline函数同理。因此必须放在头文件里。也因为这种实现方式，模板类和静态成员变量八字不合。

```cpp
template<typename T>
// 模版头 修饰范围仅限于一个函数或类定义
// 模板可以有多个类型参数 使用方法无异
// 下面以函数为例
void my_swap(T &a, T &b){
  T temp = a;
    a = b;
    b = temp;
}
// 当调用my_swap时 形参的类型会被送到typename T作为一个隐藏的参数（也可以手动指定）
// 这样就复用了不同类型共有的=运算符
// 注意模版函数内所有的操作应该对传入的数据类型成立
// 例如如果传入的T是一个自定义类型 你必须自行重载T的=运算符才能编译
```

模板参数中还可以定义变量，甚至还可以为这个变量赋默认值。这样做的目的是在编译时即将变量转化为常数，既保持了程序的可读性，又有助于程序优化。

模版与类型转换

调用模版函数时不确定的参数类型可能造成歧义，有歧义时无法编译，不会进行隐式类型转换。

```
// 下面两者当然成立
my_swap(int, int);
my_swap(double, double);

// 下面这样的语句不能过编译 因为这里有歧义
// my_swap定义中要求两个参数类型相同 那到底是把前面int当double还是把后面double当int?
my_swap(int, double);

// 但是模版可以直接指定类型
// 例如下面这样的用法 显式指定了这里以double类型调用my_swap 这时没有歧义
// 编译器会尝试隐式转换double(int) 可以编译
my_swap<double> (int, double);

// 一个有一些trick的例子是 如果我现在重载一个新的my_swap(double, double)
// 现在再调用my_swap(int, double)将会执行这个新的my_swap
// 因为模版my_swap的参数不匹配 自然而然编译器会寻找这个函数的其他重载
// 而这个新的my_swap没有模版 自然不会有歧义 隐式类型转换double(int)又可以发生了 可以编译
void my_swap(double a, double b){
    // ...
}
my_swap(int, double); // 成立

// 如果模板函数没有含typename T的参数 就必须手动指定数据类型
template <typename T> void foo (int a) {
    // ...
};
foo(5);        // 无法编译 模板没有类型
foo<double> (5);  // 成立
```

模版函数和普通函数的重载

```
// 模板函数可以和普通函数构成重载
// 下面这个例子 参数(int)的普通函数和模版函数共存 尽管int可以作为模板的类型
int type(int){
    return 0;
}
```

```

template<class T> int type(T){
    return 1;
}

//不指定模版参数时 优先匹配普通函数
cout << type(1) << endl;       // 0 因为有int为参数的普通函数 优先匹配
cout << type(1.001) << endl;   // 1 没有浮点为参数的普通函数 只能用模板函数

//指定模板参数时 只会使用模板函数
cout << type<int>(1) << endl; // 1 <int>指定模板参数 于是只会使用模板函数
cour << type<>(1) << endl;     // 1 有<>出现 即使中间没有参数 也认为指向模板
```

```
// 模板和模板之间也可以重载
// 下面这个例子 两个模板函数共存 尽管(T*)也可以当作一种(T) 逻辑上两者都说得通
template<class T> int type<T>{
    return 0;
}

template<class T> int type<T*>{
    return 1;
}

// 优先调用匹配更精准的模板
char *ptr;
cout << type(ptr) << endl;     // 1 因为传入的是指针 后者模板参数的匹配更精准

// 但是当不太容易区分类型谁更 “精准” 尤其是涉及引用时 编译器也不知道应该用谁 这种写法会报错
// 多个模板的重载是不良的写法 这里仅作为八股文式的介绍 请勿使用
```

CRTP

单层的模板类的用法和必须手动指定数据类型的模板函数基本相同。难点在于模板类和继承结合。

```
// 下面是一种套路做法 称为CRTP (Curiously Recurring Template Pattern)
// 目的是实现静态的多态
template<typename T>
class base {
    // ...
}

class derived: base<derived> {
    // ...
}
```

下面是一个更具体的CRTP例子。

```
template <typename T>
class B
{
public:
    B() : i_(0) {}
    void f(int i) {
        static_cast<T*>(this)->f(i); // 强制转换为子类，调用子类的f(int i)
    }
    int get() const { return i_; }
protected:
    int i_;
};

class D : B<D>
{
public:
    void f(int i) {
        i_ += i;
    }
};

B<D>* b = ...;
b->f(5);//实际调用的是子类 逻辑是B<D>.f() ->类型转换-> D:B<D>.f()
```

CRTP可能造成代码逻辑的死循环，但是使用得当时相当于在编译期就将模版与正确的函数相绑定，于是用静态的形式实现了类的多态。

Exception

用于实现业务代码和错误处理代码的分离。

```
double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main ()
{
   int x = 50;
   int y = 0;
   double z = 0;

   try {
     // try内写需要保护的代码
     // 其中任何一行抛出了异常 就会跳出try代码块跳转到相应的错误处理
     // 错误处理完成就整段try-catch跳出往下执行 不会回来try中继续
     z = division(x, y);
     cout << z << endl;
   }catch (const char* msg) {
     // catch内写异常处理的代码
     // 注意区分不同异常的标准是throw的 类型 而不是具体内容
     // 常用的做法是自行套皮定义一些用于表示异常的类型
     cerr << msg << endl;
   }catch (...){
     // 三个点表示其他所有类型的异常 类似switch-case的default
   }

   return 0;
}

// 为了配合异常 提高可读性 你可以在声明/定义函数时显示的限定其可以抛出的异常种类
void func() throw( exception_type1, exception_type2, ...) {};
```

右值引用

左值/右值

可以放在赋值语句左侧的变量是左值，否则是右值。另外一个简单的判断方法是，可以取地址的是左值，否则是右值。

语法

C++11起引入了右值引用，用&&来表示。

左/右值引用只能引用一个左/右值，**但是常量左值引用都可以**（编译器就是这么规定的）。

```
int a = 5;    // a是左值
int& b = a;   // b是左值引用
int&& c = 6;  // 3是右值 c是右值引用
const int& d = 7; // d是常量左值引用 特殊的 既可以应用左值也可以引用右值
```

直接作用

右值引用的直接作用是延长右值的生命周期，保留右值（常常是表达式）的结果以实现复用。

```
int b = 10;

int c = b + 5;     // b+5是右值，所得的临时结果在这一行完成后就会被抛弃

int&& d = b + 5;  // 而如果引用b+5，这个表达式的结果后续可以多次通过右值引用d获得，即延长了生命周期

int e = d;
int f = d;         // 通过右值引用复用表达式的结果，减少重复的计算
```

移动构造

设想一个非常常见的例子：

```
class AA{
}

AA getTempAA(){
  return AA();    // 返回一个AA对象
}

AA a = getTempAA(); // getTempAA()是一个右值 注意这里a会进行拷贝构造
```

- 函数以右值形式返回了我们需要的对象

- 我们**拷贝**这个对象（a在这里会进行**拷贝构造**）
- 再把这个右值对象丢掉

这里显然存在浪费，我们要先复制一个新对象（取决于拷贝构造的实现）再把原本就可用的对象丢掉。当对象作为参数传入时也有类似的情况。所以CPP设计了与拷贝构造对应的移动构造。移动构造的核心思路是将资源指针直接**移动**到右值对象上，避免重复的拷贝。（移动语义在有资源/指针存在时才有效，全是基本类型/栈区类型时没有意义）

```
class AA{
  // 拷贝构造和赋值函数 用于对比 内容略
  AA(const AA& a){ ... }
  AA& operation=(AA& a){ ... }

  // 移动构造和移动赋值函数
  AA(const AA&& a){ // 参数变为右值引用即可
    // 常见套路是三步走
    // 1. 清理自己
    // 2. 以指针形式转移a的资源（不同类型资源的转移方法不同 总之这里尽量拷贝即可）
    // 3. 清理a（注意 **转移意味着用自己代替a a的资源应该置空表示已经被移动 防止重复转移**）
  }
  AA& operation=(AA& a){ ... } // 同理
}
```

C++11起对所有的容器都实现了移动语义。如果没有实现移动语义，编译器将使用拷贝语义作为代替。

std::move()

很多临时变量虽然是左值，但是不会被修改或者复用，声明周期也很短，如果将其作为右值看待能够提高性能（避免拷贝）。std::move()将一个左值转换为右值。

```
AA a();

AA a2 = a; // 左值 拷贝构造
AA a3 = a; // 因为a2发生的是拷贝构造 a可以再次被使用 （如果需要重复使用就不要用移动构造）

AA a4 = std::move(a); // move将a变为右值 这里发生的是移动构造
AA a5 = std::move(a); // 移动构造中可能清理了a的资源 这一行可能报错（取决于移动构造的实现）
                      // **变量一经移动构造/赋值就不应该再次被使用**
```

完美转发

右值引用本身是一个左值，这在函数嵌套时会带来麻烦。

```
void func1 (AA a)  { ... }
// 如果形参不是引用类型 那么当实参是引用时会先解引用再拷贝
// 在函数内部 a是左值

void func2 (AA& a) { ... }
void func2 (AA&& a){ ... }
// 如果形参是引用类型 那么左/右值视作不同类型的实参（左右引用都是左值） 必须重载不能混用
// 但不管在那个函数重载的内部 a都是左值（左右引用都是左值）
```

这意味着，常规的函数会使实参的 左值/右值 性质完全丢失（再次转发时只能做左值），无法在函数调用链中保持最初实参的左右值性质。C++11使用模板实现能够保留实参左右值性质的“**完美的转发**”。

```
void funcInside(int& i){
    cout << "left " << i << endl;
}

void funcInside(int&& i){
    cout << "right " << i << endl;
}

template <typename T>
void func(T&& i){   // 模板函数的参数类型为T&&时 同时可以接受 左/右 值/引用 类型的实参

    i = 10;         // 右值引用本身是左值 可以修改

    funcInside(i);  // 直接使用i则不论实参的类型 一律按照左值转发参数
    funcInside(forward<T>(i));  // 如果使用forward<T>(参数) 则会应用实参的左右性质 完美转发
}

int main(){
    int&& i = 1;
    func(i);  // left 10\n left 10\n
              // 右值引用本身是左值
    func(1);  // left 10\n right 10\n
```

```
23 }
```