L1-绪论 & L2-编程环境

makefile

- 编译链接
  - 编译链接 `-o`
  - 只编译不链接 `-c`
  - GDB调试 `-g`
- makefile

```
## VERSION1
main: main.cpp a.h b.h c.cpp
  g++ -std=c++11 -o main main.cpp c.cpp

## VERSION2
CXX = g++
TARGET = hello
CXXFLAGS = -Wall -Wextra -std=c++11
OBJ = main.o printhello.o factorial.o

$(TARGET): $(OBJ)
  $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJ)

main.o: main.cpp
  $(CXX) $(CXXFLAGS) -c main.cpp

printhello.o: printhello.cpp
  $(CXX) $(CXXFLAGS) -c printhello.cpp

factorial.o: factorial.cpp
  $(CXX) $(CXXFLAGS) -c factorial.cpp

## VERSION3
CXX = g++
TARGET = hello
CXXFLAGS = -Wall -Wextra -std=c++11 # Wall: warning all
OBJ = main.o printhello.o factorial.o

$(TARGET): $(OBJ)
  $(CXX) $(CXXFLAGS) -o $@ $^

%.o: %.cpp
  $(CXX) $(CXXFLAGS) $< -o $@

.PHONY: clean
clean:
  rm -f *.o $(TARGET)

## VERSION4
CXX = g++
TARGET = hello
SRC = $(wildcard *.cpp)
OBJ = $(patsubst %.cpp, %.o, $(SRC))
```

```
CXXFLAGS = -Wall -Wextra -std=c++11

$(TARGET): $(OBJ)
  $(CXX) $(CXXFLAGS) -o $@ $^

%.o: %.cpp
  $(CXX) $(CXXFLAGS) $< -o $@

.PHONY: clean
clean:
  rm -f *.o $(TARGET)
```

用make指令调用的时候，其编译和链接规则为：

- 如果工程没有编译过，所有cpp文件都要编译并被链接
- 如果某几个cpp文件被修改，只编译被修改的cpp文件，并链接目标程序
- 如果头文件被改变了，编译引用它们的cpp文件，并链接目标程序
- 所有的cpp文件都被修改时，才会执行所有任务分别编译生成.o文件的任务

自动变量： $@ $^ $< (P 44)

```
$@：目标文件，$^：所有的依赖文件，$<：第一个依赖文件
```

声明与定义

- 函数声明与定义

```
int ADD(int a, int b); // 声明

int ADD(int a, int b) {return a + b;} // 定义（实现）
```

头文件只声明函数，不定义（实现）：多个实现会报错

- 变量声明与定义
  - 声明：`extern`
  - 定义 = 声明 + 内存分配

```
extern int x; //变量声明，此外都是定义

int x = 0; // 定义并初始化，全局变量不初始化默认值为0
```

宏定义

- 防止头文件被重复包含

```
#pragma once    //宏定义，防止头文件被重复包含

#ifndef __BODYDEF_H__
#define __BODYDEF_H__
{body...}
#endif  //同上
```

- 宏定义是简单替换：

```
#define sqr(y) ((y)*(y))    //括号不可删，建议使用inline替代
```

L3-封装与接口

- 函数重载：
  - 至少有一个参数类型不同；或参数数目不同
- 缺省值
  - 必须是最后一个参数
  - 缺省值可能造成二义性，编译不通过
- `auto`关键字：代替类型声明，尤其在变量依赖模板参数时。
- `decltype`关键字：对变量或表达式结果的类型进行推导

```
struct { char name[17]; } anon_u;
decltype(anon_u) as;    //重用匿名类型
```

```
//自动追踪返回类型
auto func(int x, int y) -> decltype(x+y) //c++14可省略
{return x+y;}
```

```
//结合auto和decltype，自动追踪返回类型
template <typename _Tx, typename _Ty>
//C++11语法，C++14可省略"->"和decltype
auto multiply(_Tx x, _Ty y)->decltype(x*y){ return x*y; }
//使用时
auto a = multiply(2, 3.3); //a=6.6
```

- 内联函数：`inline`修饰
  - 将内联函数实现写在头文件中
  - 定义在类声明中的函数，默认为内联函数
  - 一般构造函数、析构函数都被定义为内联函数

拓展：如何在C++中打印变量类型

[https://stackoverflow.com/questions/81870/is-it-possible-to-print-a-variables-type-in-standard-c]

L4-创建与销毁

类基础

- 类 = 数据 + 函数
- 访问权限
  - public
  - private
  - protected
- 只能在类的成员函数和友元函数中访问对象的私有成员和保护成员

构造与析构

- 构造函数：
  - 无返回值类型，函数名是类名
  - 按声明顺序初始化，而不是按照出现在初始化列表中的顺序
  - 初始化列表可委派（调用其它构造函数）
  - 可显式删除、显示声明
  - 就地初始化只是一种简便的表达方式，仍然在构造时执行
  - 默认构造函数
    - 只有什么构造函数都不定义的时候才会隐式合成默认构造函数
    - 显式声明 `A() = default;`
    - 显式删除 `A(char ch) = delete`
  - 构造顺序
    - **1.基类的构造函数**
    - **2.成员对象的构造函数**
    - **3.派生类本身的构造函数**
    - **注意：成员对象构造先于自身，但析构先自身再成员对象**
- 析构函数：
  - 无参数无返回值
  - `~ClassName() {  // 析构函数`
  - 自动调用（private也能访问）
  - **先析构自身、后析构成员变量**
  - 隐式定义的析构函数不会delete指针成员
- 全局对象
  - 在main()函数调用之前进行初始化
  - 同一编译单元中（源文件），按照定义顺序进行初始化
  - 不同编译单元中，对象初始化顺序不确定
  - 在main()函数执行完return之后，对象被析构
- 引用：
  - 定义时初始化为一个对象
  - 不能改指向，没有空引用
  - 函数返回引用时不能指向临时变量

运算符重载

- 类的运算符重载：
  - 全局：`ClassName operator+(ClassName a, ClassName b）{}`
    - 访问private成员怎么办？声明为友元
  - 成员：`ClassName operator+（ClassName b）{}` （类内）

```
//类内：成员函数
A& operator+=(A& a) { data += a.data; return *this;}

class A{
  int data;
public:
  A operator+(A b) {
    A new_b(data + b.data);
```

```
    return new_b;
  }
};

//类外：全局函数
A operator+(A& a1, A& a2) {
  A new_a(a1.data + a2.data);
  return new_a;
}
```

- 自增自减重载：后缀时加哑元
  - 哑元可以没有变量名

```
//前缀自增（先增后赋值）
A& operator++ () {
  ++data;
  return *this;
}
//后缀自增(先赋值后增)
A operator++ (int) {
  Test test(data);
  ++data;
  return test;
}
```

- =,[],(),-> 只能通过成员函数来重载
  - 函数运算符()重载
  - 数组下标[]重载
    - 如果返回类型是引用，则数组运算符调用可以出现在等号左边，接受赋值
      - `Obj[index] = value;`
    - 如果返回类型不是引用，则只能出现在等号右边
      - `Var = Obj[index];`
- 更多示例

```
//Test类内重载函数运算符()
int operator() (int a, int b) {
  return a + b;
}
//调用
Test sum;
int s = sum(3, 4); // sum 是“函数对象”

//重载数组下标运算符
int& operator[] (const char* name) // 字符串作下标
{
  return name[i];
}
```

- 输入输出流：必须加友元，只能全局重载

```
istream& operator>>(istream& in, Test& dst ) {
    in >> dst.data;
    return in;
}

ostream& operator<<(ostream& out, const Test& src) {
  out << src.id << endl;
  return out;
}
```

- 运算符重载规则：
- C++ 中的所有运算符都可以重载。例外：类属关系运算符 `.` 成员指针运算符 `.*` 作用域运算符 `::` `sizeof`运算符、三目运算符 `? :` ；
- 不能创建新的运算符；
- 不能改变运算符的优先级和结合性；
- 不能改变运算符操作数的个数及语法结构

L5-创建与销毁

- 友元：
  - 类内声明，与所在域为 private/public 皆可
  - 不是成员函数，可以**访问对象的私有成员**
  - 不对称、不传递、不继承
  - 可以是多个类的友元函数，可以访问这几个类的所有私有数据
  - 友元类，该类的所有成员函数均为友元函数
- 静态变量/函数（static）
  - 静态变量
    - **不能在任何函数和局部作用域中初始化**
    - 全局变量内部可链接（只在声明的文件中使用），非静态全局变量可用于其他文件
    - 程序最后析构
  - 静态函数
    - `static int func() {…}`
    - 内部可链接, 不能被其他文件使用
    - 静态数据成员——类的所有对象共享（可通过类名、对象访问）

```
static Type static_var; //.h文件类内声明
Type ClassName::static_var = Value; //cpp文件中赋初值
```

- 静态成员变量
  - 属于整个类，被该类的所有对象共享，可以同构对象访问，也可以通过类名访问
  - 类似全局变量，程序开始前初始化
  - 在.h中声明，.cpp中定义，不能.h中同时完成声明和定义`int ClassName::static_var = 0`
  - int 类型的常量静态数据成员可以在类内初始化
- 静态成员函数 `static void func();`
  - 属于整个类，被该类的所有对象共享，可以同构对象访问，也可以通过类名访问
  - 实例化前就分配内存，**不能访问非静态**成员，只能使用静态成员函数访问静态成员
- 常量（const）：
  - 修饰变量——必须就地初始化，值不改变
  - 修饰引用/指针——无法通过引用/指针修改变量值
  - 修饰函数返回值——返回值的内容或指向的内容不能被修改

```
int a = 1; const int& b = a;     //不能通过b修改变量的值
const int* p = &a;  //不能通过p修改变量的值
int* const p = &a;  //不能改变p的指向
```

- 常量数据成员
  - 初始化：初始化列表 / 就地**（不能在构造函数体内）**
  - 不可修改
- 常量成员函数
  - `ReturnType Func(..) const {..}`
    - 注意const写在参数后面，写在最前面是返回值类型是const的函数
  - 不能修改类的数据成员
- 常量对象
  - `const ClassName a;`
  - 对象中数据不变
  - **只能调用 const修饰的成员函数（常量成员函数）**
- 常量静态变量
  - `static const`
  - 不存在常量静态函数
  - 类外定义（例外：int 和 exturn 可以就地初始化）

```
class A {
    static const char* cs; // 不可就地初始化
    static const int i = 3; // 可以就地初始化
    static const int j; // 也可以在类外定义
};
const char* A::cs = "string";
const int A::j = 4;
```

数据成员 / 操作类型	静态 static	常量 const	常量静态 (除int, enum外)	常量静态(int, enum)
就地初始化		√		√
初始化列表初始化		√		
构造函数体内初始化				
类外初始化	√		√	√
普通成员函数	√√	√	√	√
静态成员函数	√√		√	√
常量成员函数	√√	√	√	√

注：√ 表示仅可访问，√√ 表示可访问也可修改

- 常量对象的构造与析构
  - 全局和局部都与普通对象相同
  - 常量全局对象：在main()函数调用之前进行初始化，在main()函数执行完return时析构

    - 常量局部对象：在程序执行到该局部对象的代码时被初始化。在局部对象作用域结束后被析构
- 静态对象的构造与析构：
    - 全局与普通函数相同
    - 局部
        - 函数中静态对象：
            - 初始化：在程序第一次执行到该静态局部对象的代码时被初始化
            - 析构：离开作用域不析构，在main()函数结束后被析构
            - 第二次执行到代码时，不再初始化，直接使用上一次的对象
        - 类静态对象

```
class A {};

class B {
    static A a; //
};
```

            - a在main()函数调用之前初始化，在main函数执行完return时析构
            - 和B是否实例化无关
- 参数对象的构造和析构：
    -

```
void fun(A b) {
    cout << "In fun: b.s=" << b.s << endl;
}
fun(a);
//函数调用时b被构造，调用拷贝构造函数
//函数结束时调用析构函数，b才被析构
```

        - 如果一个类含有指针，更会因为析构两次释放同一块内存地址导致出错
    - 如果参数是类对象的引用

```
void fun(A &b) {
    cout << "In fun: b.s=" << b.s << endl;
}
fun(a);
//在函数调用时，b不需要调用拷贝构造函数进行初始化，因为b是a的引用
//在函数结束时，也不需要调用析构函数，因为b只是一个引用，而不是A的对象
```

        - 尽量使用对象引用作为参数
    - 使用 new/delete 生成对象数组：`A* p = new A[3];` 内存分配多4字节存数组大小
    - `delete []p;` 会释放包括4字节的所有内存

L6-引用与复制

- 引用：
    - 定义时初始化，且不能改引用指向
    - 函数参数引用，形参等于实参
    - 函数返回值，不可返回临时变量
- 常量引用：
    - 函数参数 `const int& a`，仅读取，没有权限修改a的值（最小特权原则）

```
ClassName(const ClassName& a);
```

- 右值引用：
  - 不能取地址，但可以被&&引用
  - 右值引用无法绑定左值 `int &&e = a + b;`
    - 例外：

```
const int& a = 1;  //常量左值引用可以绑定左值也能绑定右值，但优先级低
```

- 拷贝构造函数
  - 调用时机：
    - 1.用一个类对象定义另一个新的类对象 `Test b(a);`
    - 2.函数调用时以类的对象为形参 `Func(Test a)`
    - 3.函数返回类对象 `Test Func(void)`
  - 执行顺序

```
Myclass func(Myclass c) {   //拷贝构造
    Myclass tmp;    //默认构造
    return tmp;     //拷贝构造
}   //temp析构，c析构

//编译选项，禁止编译器进行返回值优化
g++ test.cpp --std=c++11 -fno-elide-constructors -o test
```

  - 使用左值引用作为参数的构造函数：`ClassName(ClassName& VariableName);`
  - 默认拷贝构造：位拷贝，复制指针对象时，多个指针类型的变量会指向同一个地址
  - 拷贝构造函数
  - 应尽量避免拷贝构造，函数参数传引用或常量引用，函数返回值设引用，并将拷贝构造函数设为 `private` 或 `= delete ;`

- 移动构造函数：
  - 使用右值引用作为参数的构造函数：`ClassName(ClassName&& VariableName);`
  - 直接利用即将析构的临时类的堆内存，相比拷贝构造函数，新对象不占新的堆内存
- 移动：`std::move()`
  - 使用移动构造函数，加快左值初始化的构造速度
  - 输入左值，返回该左值对应的右值
  - 仅作类型转换，移动的具体操作在构造函数内实现

```
std::string str = "Hello";
std::string str2 = std::move(str);
cout<<str; // 输出空字符串
```

```
//拷贝构造
class Test {
public:
    Test(const Test& other) {
        //将other中的内容拷贝到当前对象
    }
};
```

```
//移动构造
class Test {
public:
    Test(const Test&& other) {
  //Test(const Test&& std::move(一个左值))
    //
    }
};
```

- 拷贝/移动赋值运算符：（类的非静态成员，不能是友元）

```
//拷贝赋值：已定义的对象之间相互赋值
ClassName& operator= (const ClassName& right) {
   if (this != &right) {//避免自己赋值给自己
        //将right对象中的内容拷贝到当前对象中...
    }
   return *this;
}
//移动赋值
ClassName& operator= (ClassName&& right) {
   if (this != &right) {//避免自己赋值给自己
       this->buffer = right->buffer; //直接复制地址
       right->buffer = nullptr;
    }
   return *this;
}
```

- 以上函数c++11编译器都会自动合成。
- 自动类型转换：
  - 显式 Classname a(b);
  - 隐式（可通过explicit禁用）
- 两种强制类型转换
  - `static_cast`
    - 内置数据类型的转换
    - 类的指针或者类的引用的强制转换
  - `dynamic_cast`
    - 主要用于将基类类型转化为派生类类型
    - 根据虚函数表的信息判断实际类型
    - 不能用于内置基本数据类型的强制转换
    - 不允许两个没有关联的类的指针相互转换

```
static_cast<NewType>(a)      //静态类型转换
dynamic_cast<NewType>(a)     //动态类型转换
```

L7-组合与继承

- 组合 has-a
  - 子对象构造如果需要参数，应该在当前类的构造函数的初始化列表中进行，使用默认构造函数来构造子类，就不用处理。
  - 对象构造与析构函数的次序：
    - 先构造子对象，再构造当前对象

        - 先析构当前对象，再析构子对象
    - 隐式定义的拷贝构造函数：
        - 递归调用所有子对象的拷贝构造函数
        - 对于基础类型，采用位拷贝
- 继承 is-a
    - 基类的构造函数、析构函数、友元函数、赋值运算符不会被继承
    - 构造顺序：基类->派生类->函数体；
    - 析构顺序：函数体->派生类->基类
    - 构造派生类对象时会调用基类的构造函数
        - 如果没有显式调用，则会自动合成一个对基类的默认构造函数的调用
        - 显式调用只能在派生类构造函数的初始化列表中进行
    - 继承基类构造函数
        - 构造派生类会直接调用基类的构造函数版本，如果基类有多个构造函数（非私有），都会继承过来，并且派生类不会再合成隐式构造函数

```
Base(int i) : data(i) {}     //基类构造函数
using Base::Base;   //子类中继承基类构造函数
Derive(int i) : Base(i){};  //效果同上
```

- 基类成员访问权限与三种继承方式
    - public继承
    - private继承
    - protected继承

成员访问权限

继承表		继承方法					
		public		private		protected	
基类中成员类型	public	OK	pub/yes	OK	prv/no	OK	pro/no
	private	NO	prv/no	NO	prv/no	NO	prv/no
	protected	OK	pro/no	OK	prv/no	OK	pro/no

派生类成员函数能否访问基类成员

基类成员在派生类中的成员类型，派生类对象能否访问基类成员

prv: private
pro: protected
pub: public

类似集合交运算(成员类型与继承类型之间取交)
Order: public ⊃ protected ⊃ private

- **重载**（overload）
    - 目的：同名函数的不同实现
    - 函数名相同，函数参数必须不同，作用域相同
- **重写隐藏**（redefining）
    - 目的：在派生类中重新定义基类函数，实现派生类的特殊功能
    - 函数名相同，参数可以不同，作用域不同（派生类和基类）
    - 屏蔽**基类所有同名函数**

- 使用`using Base::f;`可恢复全部基类同名的成员函数（基类该函数不是虚函数）

```
class Base {
public:
  void f() {}
  void f(int i) {} /// 重载
};
class Derive : public Base {
public:
  void f(int i) {} ///重写隐藏
};

int main() {
  Derive d;
  d.f(10);
  // d.f(); // 被屏蔽，编译错误
  return 0;
}
```

- 多重继承：
  - 同名成员存在二义性。

```
class IOFile: public InputFile, public OutputFile{};
```

L8-虚函数

- 向上类型转换：
  - **派生类（对象/引用/指针）转换为基类（对象/引用/指针）**
  - **凡是**接受基类对象/引用/指针的地方（如函数参数），**都**可以使用派生类对象/引用/指针（编译器会**自动**转换）；
  - 仅对public继承有效
  - 对象向上转换会被切片，丢弃派生类独有的部分数据和接口；
  - 指针（引用）被转换为基类指针（引用）时，不会创建新的对象，但只保留基类的接口
- 捆绑：函数体与函数调用相联系
  - 早绑定：运行前完成
  - 晚绑定：根据对象的实际类型决定是调用基类中的函数还是派生类重写的函数
    - 只对在**基类中**被声明为**虚函数**再被派生类重新定义的成员函数
    - 通过**基类引用/指针**调用
    - 通过虚函数表实现
      - 虚函数表(VTABLE)：在每个**包含虚函数的类**中，用于存储虚函数地址的表（虚函数表有唯一性，**编译期**建立）
      - 虚函数指针(vpointer/VPTR)：在每个包含虚函数的类**对象**中，指向这个类的VTABLE，占8字节（运行时，**构造函数**中发生，指针调用虚函数时引起晚绑定）
- 虚函数的**重写覆盖**：(override)
  - 派生类重新定义基类中的虚函数
  - 函数名相同、函数参数相同、返回值一般情况相同，作用域不同（派生类和基类）
  - 基类声明为虚函数（virtual）
  - 会发生虚函数指针覆盖
  - override 辅助检查`ReturnType Func(int)override {...}`
  - final终止继承：确保函数为虚且不能被派生类重写；在类定义中指定此类不可被继承

- **重写隐藏**
  - 基类的函数不是虚函数或函数参数不同
  - 不会发生虚函数表中指针的覆盖，仍然调用基类的函数
- 虚函数和构造函数、析构函数
  - 构造函数不能是虚函数
  - 析构函数常常是虚函数（否则可能内存泄漏）
- OOP的核心思想：
  - 数据抽象：类的接口与实现分离
  - 继承：建立相关类型的层次关系（基类与派生类）
  - 动态绑定：统一使用基类指针，实现多态行为

L9-多态与模板

- 纯虚函数：
  - `virtual void f() = 0;`
  - 由于编译器不允许被调用函数的地址为0，所以该类不能生成对象
  - 纯虚析构函数，也需要定义函数体 {}
- 抽象类：
  - 至少一个纯虚函数，不允许实例化为对象（保证只有指针/引用能向上类型转换，防切片）
  - 抽象类的派生类必须重写所有纯虚函数（除析构），否则仍为抽象类
- 向下类型转换：指针/引用的转换
  - dynamic_cast：使用dynamic_cast的对象必须有虚函数，因为它使用了存储在虚函数表中的信息判断实际的类型。
  - static_cast：在编译时静态浏览类层次，只检查继承关系。

```
//obj_p，obj_r分别是T1类型的指针和引用
//T1必须是多态类型（声明或继承了至少一个虚函数的类）
T2* pObj = dynamic_cast<T2*>(obj_p);
    //转换为T2指针，运行时失败返回nullptr
T2& refObj = dynamic_cast<T2&>(obj_r);
    //转换为T2引用，运行时失败抛出bad_cast异常

//obj_p，obj_r分别是T1类型的指针和引用
T2* pObj = static_cast<T2*>(obj_p);
  //转换为T2指针
T2& refObj = static_cast<T2&>(obj_r);
  //转换为T2引用
//不安全：不保证指向目标是T2对象，可能导致非法内存访问。
```

- 多态（概念）
  - 定义：按照基类的接口定义，调用指针或引用所指对象的接口函数，函数执行过程因对象实际所属派生类的不同而呈现不同的效果（继承 && 虚函数 && (引用 || 指针)）
  - 优点：提高程序的可复用性、可拓展性和可维护性
  - 静多态：模板、重载
  - 动多态：虚函数、继承
- 模板：
  - 对模板的处理是在编译期进行的，每当编译器发现对模板的一种参数的使用，就生成对应参数的一份代码(实例化)
  - 注：必须在头文件中实现，不能分开编译
- 函数模板

```
template <typename T>
T sum(T a, T b) { return a + b; }
//调用：实例化
cout << sum(9, 3);
cout << sum(2.1, 5.7);
cout << sum(9, 2.1); //编译错误
cout << sum<int>(9, 2.1); //正确，可以手动指定类型
```

- 类模板：将一些类型信息抽取出来，用模板参数来替换，从而使类更具通用性
    - 注意主函数实例化

```
template <typename T> class A {
    T data; //类内成员变量
public:
    A(T _data): data(_data) {}
};
template<typename T>    //类外成员函数定义
void A<T>::print() { cout << data << endl; }

int main() {
    A<int> a(1);    //实例化
    return 0;
}
//可包含非类型参数
template <typename/class T, unsigned size> //编译时必须确定size的值
```

- 模板参数

```
template<typename T, unsigned size>
class array {
  T elems[size];
};

int main(){
  int n = 5;
  //array<char, n> array0; //不能使用变量
  const int m = 5;
  array<char, m> array1; //可以使用常量
  array<char, 5> array2; //或具体数值
  return 0;
}
```

- 成员函数模板

```
//普通类的成员函数，也可以定义为模板函数，如：
  class normal_class {
  public:
      int value;
      template<typename T> void set(T const& v) {
          value = int(v);
    }     /// 在类内定义
      template<typename T> T get();
  };
  template<typename T>         /// 在类外定义
  T normal_class::get() {
    return T(value);
  }
```

- 额外模板参数

```
  template<typename T0> class A {
      T0 value;
  public:
      template<typename T1> void set(T1 const& v){
          value = T0(v); /// 将T1转换为T0储存
      }          /// 在类内定义
      template<typename T1> T1 get();
  };
  template<typename T0> template<typename T1>
  T1 A<T0>::get(){ return T1(value);} /// 类外定义， 将T0转换为T1返回
```

- 多个参数的模板
- 函数模板特化
  - 只能全部特化，部分特化只能用重载代替

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

template<class T>
T div2(const T& val) {
    cout << "using template" << endl;
    return val / 2;
}

template<> //函数模板特化
int div2(const int& val) {
    cout << "better solution!" << endl;
    return val >> 1; //右移取代除以2
}
int main() {
    cout << div2(1.5) << endl;
    cout << div2(2) << endl;
    return 0;
}
```

- 类模板特化
  - 可以部分特化或全部特化

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

template<class T1, class T2>
class Sum { //类模板
public:
    Sum(T1 a, T2 b) {cout << "Sum general: " << a + b << endl;}
};

template<>
class Sum<int, int> { //类模板全部特化
public:
    Sum(int a, int b) {cout << "Sum specific: " << a + b << endl;}
};

template<class T1>
class Sum<T1, int>{ //类模板部分特化
public:
    Sum(T1 a, int b){cout << "Sum partial_specific: " << a + b << endl;}
};
int main(){
    Sum<int, int> s1(1, 2);
    Sum<int, double> s2(1, 2.5);
    Sum<double, int> s3(1.4, 2);
    return 0;
}
```

L11-模板与stl初步

命名空间:

```cpp
namespace A{int x,  y;} //定义
A::x = 3;   //使用
using A::y; y = 6;  //使用
```

任何情况下，都不应出现命名冲突

STL:

- 标准模板库（Standard Template Library），是一个高效的C++软件库。
- 包含4个组件：算法、容器、函数、迭代器。
- 基于模板编写。
- 关键理念：将“在数据上执行的操作”与“要执行操作的数据”分离。
- 容器
    - 简单容器
    - 序列容器
    - 关系容器

pair:

```
//创建方式1
std::pair<int, int> t;
t.first = 4; t.second = 5;
//创建方式2
auto t = std::make_pair("abc", 7.8); //自动推导成员类型
//创建方式3
std::pair<std::string, double>p1("abc", 90.5);
//比较
std::make_pair(1, 4) > std::make_pair(1, 2); //先比较 first
                                //要求成员类型支持比较(实现比较运算符重载)
```

tuple:

- pair 的扩展，下标在编译时确定

```
//创建
auto t = std::make_tuple("abc", 7.8, 123, '3');
//创建：tie函数-返回左值引用的元组
std::string x; double y; int z;
std::tie(x, y, z) = std::make_tuple("abc", 7.8, 123);
//get获取数据
auto v0 = std::get<0>(t); //v_0 = "abc";
```

vector: （序列容器）

自动扩展容量的数组，允许下标访问（高速）

```
std:vector<int> x;  //创建
x.size();   //当前数组长度
x.clear();  //清空
x.push_back(1); x.pop_back();   //在末尾添加/删除（高速）
x.insert(x.begin()+1, 5);   //在中间添加（低速）
x.erase(x.begin()+1);   //在中间删除（低速）
```

元素被删或删除时，后续所有元素的迭代器失效；扩容时全失效

迭代器: iterator （一种数据类型）

`vector<int>::iterator iter;`

```
for(auto & x : vec) //遍历vector 直接利用vec中元素x
for(vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)  //使用 *it，即it是指向元素的指针
*iter = 5;//解引用运算符*访问元素值
iter += 5;//迭代器加法
int dist = iter1 - iter2;//元素位置差
```

绝对安全准则：在修改过容器后，不使用之前的迭代器

list: （序列容器）

- **不支持下标**等随机访问
- 支持在**任意位置插入/删除**数据（高速）
- 其访问主要依赖迭代器
- 插入和删除不会导致迭代器失效（除非指向被删除的元素）

set: （关联容器）

- **不重复**元素构成的**无序**集合（内部从小到大排）
- 无序指不保持插入顺序，容器内部排列顺序是根据元素大小排列

map：关联数组（关联容器）

- 元素key必须互不相同
- 可以通过**下标访问**（即使key不是整数）
- 下标访问时如果元素不存在，则**创建对应元素**
- 可使用 insert 函数进行插入
- 常用作稀疏数组或以字符串为下标的数组

```
#include <string>
#include <map>
int main() {
    std::map<std::string, int> s;
    s["Monday"] = 1;
    s.insert(std::make_pair(std::string("Tuesday"), 2));
    return 0;
}
```

L12-STL和字符串处理

string字符串类

构造：

```
//字符串 string 构造：
string str0;
string str1("Initial string");
string str2(str1, 1, 3);    // nit
string str3(str1, 3);   // Ini
string str4("Another string", 10);  // Another st
string str5(5, 'x');        // xxxxx
string str6(str1.begin(), str1.begin() + 7) // Initial
//转换为c风格字符串
const char* c = str1.c_str() //返回值为常量字符指针(const char*)，c不可修改
```

用法：

```
//常见用法，和vector类似
cout << str[1]; str[1]='a'; //访问/修改元素
str.size()  //查询长度
str.clear() //清空
str.empty() //查询是否为空
for(char c : str)   //迭代访问
str.push_back('a'); str.append(s2); //向尾部增加
//不同于vector：
str.length()    //查询长度
str1 += str2; str1 += 'a';  //拼接
```

读入：

```
getline(cin, fullname); //读一行
getline(cin, str, "#"); //到指定分割符“#”时停止读入
```

比较：字典序

转数值：

```
int a = stoi(str, 2);   //读前两位
double e = stod("34.5")   //e=34.5
```

iostream输入输出流

- ostream（output stream）：STL库中所有输出流的**基类**
- cout：STL中内建的一个 ostream **对象**
- endl：流操纵算子，函数（输出 '/n' 再清除缓冲区）
- scanf 的格式字符串可以在运行时确定，需要在运行期间解析；istream在编译期间已经解析完毕
- 格式化输出`#include <iomanip>`

```
cout << fixed << 2018.0 << " " << 0.0001 << endl;
                //浮点数 -> 2018.000000 0.000100
cout << scientific << 2018.0 << " " << 0.0001 << endl;
                //科学计数法 -> 2.018000e+03 1.000000e-04
cout << defaultfloat;  //还原默认输出格式
cout << setprecision(2) << 3.1415926 << endl;
                //输出精度设置为2 -> 3.1
cout << oct << 12 << " " << hex << 12 << endl;
                //八进制输出 -> 14  十六进制输出 -> c
cout << dec;    //还原十进制
cout << setw(3) << setfill('*') << 5 << endl;
                //设置对齐长度为3，对齐字符为* -> **5
```

文件输入输出流：

```
ifstream ifs("input.txt");  //打开文件
ifstream ifs("binary.bin", ifstream::binary);   //以二进制形式打开文件
ifs.open("file")
//do something
ifs.close()
```

正则表达式：

```
//匹配的单个字符在某个范围中
[a-z] 匹配所有单个小写字母
[0-9] 匹配所有单个数字
//连用
[a-z][0-9] 匹配所有字母+数字的组合，比如a1、b9
    [Tt]he: The car parked in the garage.
//范围取反
[^a-z]: 匹配所有非小写字母的单个字符
[^c]ar: The car parked in the garage.
^[^0-9][0-9]$: 匹配长度为2的内容，且第一个不为数字，第二个为数字
//特殊字符
\d 等价[0-9]，匹配所有单个数字
```

```
\D 等价[^0-9]，匹配所有单个非数字
\w 匹配字母、数字、下划线，等价[a-zA-Z0-9_]
\W 匹配非字母、数字、下划线，等价[^a-zA-Z0-9_]
.匹配除换行以外任意字符
    .ar: The car parked in the garage.
\.可表示匹配句号
    ge\.: The car parked in the garage.
\s匹配任何空白
\S匹配任何非空白
//x{n,m}代表前面内容出现次数重复n~m次
a{4} 匹配aaaa
a{2,4} 匹配aa、aaa、aaaa
a{2,} 匹配长度大于等于2的a
^代表字符串开头，$代表字符串结尾
+ 前一个字符至少连续出现1次及以上
    a\w+: The car parked in the garage.
? 出现0次或1次
* 至少连续出现0次及以上
//创建一个正则表达式对象
regex re("^[1-9][0-9]{10}$")  11位数
```

- `<regex>` 库
- 原生字符串模式：R， 特殊字符不会被转义
- 密码匹配：同时包含数字和字母

```
^(?=.\d+)(?=.*_+)(?=.*[A-Za-z]+)[\w]{6,16}$
```

```
//regex_match(s, re)：询问字符串s是否能完全匹配正则表达式re
using namespace std;

int main() {
    string s("subject");
    regex e("sub.*");
    smatch sm;
    if(regex_match(s, e))
        cout << "matched" << endl;
    return 0;
}
```

- 捕获和分组

```
//使用()进行标识，每个标识的内容被称作分组
//正则表达式匹配后，每个分组的内容将被捕获
//用于提取关键信息，例如version(\d+)即可捕获版本号

//regex_match(s, result, re)：询问字符串s是否能完全匹配正则表达式re，并将捕获结果储存到
result中
//result需要是smatch类型的对象
using namespace std;
int main () {
    string s("version10");
    regex e(R"(version(\d+))");
    smatch sm;
    if(regex_match(s, sm, e)) {
        cout << sm.size() << " matches\n";
```

```
        cout << "the matches were:" << endl;
        for (unsigned i = 0; i < sm.size(); ++i) {
            cout << sm[i] << endl;
        }
    }
    return 0;
}
//分组会按顺序标号
//0号永远是匹配的字符串本身
//(a)(pple)： 0号为apple，1号为a，2号为pple
//用(sub)(.*)匹配subject：0号为subject，1号为sub，2号为ject
```

- 搜索

```
//搜索
//regex_search(s, result, re)：搜索字符串s中能够匹配正则表达式re的第一个子串，并将结果存储在result中

//result是一个smatch对象
//对于该子串，分组同样会被捕获
int main() {
    string s("this subject has a submarine");
    regex e(R"((sub)([\S]*))");
    smatch sm;
    //每次搜索时当仅保存第一个匹配到的子串
    while(regex_search(s,sm,e)){
        for (unsigned i = 0; i < sm.size(); ++i)
            cout << "[" << sm[i] << "] ";
        cout << endl;
        s = sm.suffix().str();
    }
    return 0;
}
```

- 替换

```
//regex_replace(s, re, s1)：替换字符串s中所有匹配正则表达式re的子串，并替换成s1
int main() {
    string s("this subject has a submarine");
    regex e(R"(sub[\S]*)");
    //regex_replace返回值即为替换后的字符串
    cout << regex_replace(s,e,"SUB") << "\n";
    return 0;
}
```

L13-STL函数对象和智能指针

函数对象

- 需要重载 operator()，权限为 public
- greater 模板类
- 类的对象`#include <functional>`
  - `sort(arr, arr+5, less<int>());`从小到大
  - `sort(arr, arr+5, greater<int>())`从大到小

函数指针

- 函数指针：
  例如 `void (*func) (int&);`
- `std::function`类：
  统一函数指针和函数对象，来自`<functional>`

```
function<string()> readArr[] = { readFromScrean, readFromFile };
```

智能指针

- shared_ptr：来自库
  - 构造方法

```
shared_ptr<int>p1 ( new int(1) );
auto p2 = make_shared<MyClass>(1);
shared_ptr<MyClass> p3 = p2;
shared_ptr<int> p4; //空指针
```

- `use_count`
  - 计数归零时，销毁对象
- weak_ptr：弱引用，指向对象，但不计数，避免环状结构无法销毁内存

```
//弱引用指针的创建
shared_ptr<int> sp(new int(3));
weak_ptr<int> wp1 = sp;
//弱引用指针的用法
wp.use_count()  //获取引用计数
wp.reset()          //清除指针
wp.expired()        //检查对象是否无效
sp = wp.lock()  //从弱引用获得一个智能指针
```

- unique_ptr：
  - 不能复制，能移动
  - 可以放弃指针控制权，返回裸指针

```
#include <memory>
#include <utility>
using namespace std;
int main() {
    auto up1 = std::make_unique<int>(20);
    //unique_ptr<int> up2 = up1;
                //错误，不能复制unique指针
    unique_ptr<int> up2 = std::move(up1);
                //可以移动unique指针
    int* p = up2.release();
                //放弃指针控制权，返回裸指针
    delete p;
    return 0;
}
```

- 实现智能指针

```
template <typename T>
class SmartPtr { //智能指针
    U_Ptr<T> *rp;
public:
    SmartPtr(T *ptr) :rp(new U_Ptr<T>(ptr)) { }
    SmartPtr(const SmartPtr<T> &sp) :rp(sp.rp) {
        ++rp->count;
    }
    SmartPtr& operator=(const SmartPtr<T>& rhs) {
        ++rhs.rp->count;
        if (--rp->count == 0) //减少自身所指rp的引用计数 pA = pB
            delete rp; //删除所指向的辅助指针
        rp = rhs.rp;
        return *this;
    }
    ~SmartPtr() {
        if (--rp->count == 0)
            delete rp;
    }
    T & operator *() { return *(rp->p); }
    T* operator ->() { return rp->p; }
};

int main(int argc, char *argv[]) {
    int *pi = new int(2);
    SmartPtr<int> ptr1(pi); //构造函数
    SmartPtr<int> ptr2(ptr1); //拷贝构造
    SmartPtr<int> ptr3(new int(3)); //能否ptr3(pi)???
    ptr3 =  ptr2; //注意赋值运算
    cout << *ptr1 << endl; //输出2
    *ptr1 = 20;
    cout << *ptr2 << endl; //输出20

    return 0;
}
```

L14-行为型模式

设计模式

- 行为型模式（Behavioral Patterns）
  关注对象行为功能上的抽象，旨在提升对象在行为功能上的可拓展性，以**最少的代码变动完成功能的增减**；
- 结构型模式（Structural Patterns）
  关注对象之间结构关系上的抽象，旨在提升对象结构的可维护性、代码健壮性，在**结构层面上解耦合**；
- 创建型模式（Creational Patterns）
  将对象创建与使用分离，旨在规避复杂对象创建带来的资源消耗，以**简短代码完成对象的高效创建**；

行为型模式

- 模板方法（Template Method）模式
- 策略（Strategy）模式
- 迭代器（Iterator）模式

模板方法

- 继承
- 抽象类（父类）定义算法的骨架——父类是纯虚函数
- 算法的细节由实现类（子类）负责实现
- 在使用时，调用抽象类的算法骨架方法，再由这个方法来根据需要调用具体类的实现细节
- 使用基类指针进行晚绑定
- 模板方法更加侧重于逻辑复杂但结构稳定的场景，尤其是其中的某些步骤（部分功能）变化剧烈且没有相互关联。

策略模式

- 组合
- 统一的策略调用接口
- 策略模式则适用于算法（功能）本身灵活多变的场景，且多种算法之间需要协同工作。