# 复习

## directory



## 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

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

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
- `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;}
```

- 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。

# 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
  - 按声明顺序初始化
  - 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
  - 就地初始化仍然在构造时执行
  - 默认构造函数
    - 显式声明 `A() = default;`
    - 显式删除 `A(char ch) = delete`
- 析构函数：
  - 无参数无返回值
  - `~function() {  // 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**
- 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
  - 不能改指向
  - 函数返回引用时不指向临时变量

# 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 );

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
  - 不对称、不传递、不继承
- 静态变量/函数（static）
  - 静态变量
    - 只在定义时进行一次初始化
      - 不在类内初始化，使用作用域解析运算符在类外初始化。
    - 全局变量内部可链接（只在声明的文件中使用）
    - 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
  - 类似全局变量，程序开始前初始化
- 静态成员函数 `static void func();`
  - 属于整个类，被该类的所有对象共享
  - 实例化前就分配内存，不属于某个对象而属于整个类，**不能访问非静态**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

```
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
```

- 修饰函数
  - `const Type Func(..) {..}`
  - 返回值不可变，可重载
- 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）
- 常量成员函数
  - `ReturnType Func(..) const {..}`
  - 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

- 静态局部对象：
  - 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
- 参数对象的构造和析构：
  - 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

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

- 同类对象的常量左值引用（可绑定左值、右值）；
- 位拷贝，注意隐式拷贝得到的指针成员指向相同内存，应自定义；
- 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 。

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
```

- 右值引用：
  - 不能取地址 没有名字的值
  - `int &&e = a + b;` （无法绑定左值）

    例外：

```
const int& a = 1;  //常量引用也能绑定右值，但优先级低
```

- 拷贝构造函数
  - 使用左值引用作为参数的构造函数
  - 只有定义对象时的`=`才是拷贝构造
  - `ClassName(ClassName& VariableName);`
  - 拷贝构造函数：
- 移动构造函数：
  - 使用右值引用作为参数的构造函数
  - `ClassName(ClassName&& VariableName);`
  - 相比拷贝构造函数，新对象不占新的堆内存
- `std::move`
  - 使用移动构造函数，加快左值初始化的构造速度
  - 左值强制转为右值

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

- 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
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
        // 将right对象中的内容移动到当前对象中...
    }
   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子类类型
  - 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
- 对象构造与析构函数的次序：
  - 先完成子对象构造，再完成当前对象构造
  - 子对象构造的次序 == 类中声明的次序
  - 析构函数的次序与构造函数相反
- 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、友元函数不会被继承
  - 继承方法，默认 private；
  - 构造顺序：基类->派生类->函数体；
  - 析构顺序：函数体->派生类->基类。

```
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
```

- **重载**（overload）
  - 函数名相同，函数参数必须不同，作用域相同
- **重写隐藏**（redefining）
  - 重写隐藏：
    - 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（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**
  - **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，自动隐式转换；
  - 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；
- 捆绑：
  - 实现代码与调用的函数名绑定
  - 早绑定：编译器完成
  - 晚绑定：
    - 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

        - 编译器根据对象的实际类型决定时调用基类中的函数还是派生类重写的函数
        - 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）
    - override 辅助检查
- 成员函数仍然是虚函数，可不声明
- 只对指针和引用有效（类外）
- 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）
- 终止继承：final
- 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

```
auto obj_2 = dynamic_cast<Type/Type&>(obj_1); //安全、开销大，必须有虚函数
auto obj_2 = static_cast<Type/Type&>(obj_1); //不安全、开销小
```

- 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

```
template <typename/class 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：更具通用性的类

```
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
```