# 编程基础

## 变量声明和定义区别？

* + 声明仅仅是把变量的声明的位置及类型提供给编译器，并不分配内存空间；定义要在定义的地方为其分配存储空间。
  + 相同变量可以再多处声明（外部变量extern），但只能在一处定义。

## "零值比较"？

* + bool类型：if(flag)
  + int类型：if(0 == flag)
  + 指针类型：if(null == flag)
  + float类型：if((flag >= -0.000001) && (flag <= 0. 000001))

## strlen和sizeof区别？

* + sizeof是运算符，并不是函数，结果在编译时得到而非运行中获得；strlen是字符处理的库函数。
  + sizeof参数可以是任何数据的类型或者数据（sizeof参数不退化）；strlen的参数只能是字符指针且结尾是'\0'的字符串。
  + **因为sizeof值在编译时确定，所以不能用来得到动态分配（运行时分配）存储空间的大小。**

## 同一不同对象可以互相赋值吗？

* + 可以，但含有指针成员时需要注意。
  + 对比类的对象赋值时深拷贝和浅拷贝。

## 结构体内存对齐问题？

* + 结构体内成员按照声明顺序存储，第一个成员地址和整个结构体地址相同。
  + 未特殊说明时，按结构体中size最大的成员对齐（若有double成员），按8字节对齐。

## static作用是什么？在C和C++中有何区别？

* + static可以修饰局部变量（静态局部变量）、全局变量（静态全局变量）和函数，被修饰的变量存储位置在静态区。对于静态局部变量，相对于一般局部变量其生命周期长，直到程序运行结束而非函数调用结束，且只在第一次被调用时定义；对于静态全局变量，相对于全局变量其可见范围被缩小，只能在本文件中可见；修饰函数时作用和修饰全局变量相同，都是为了限定访问域。

修饰普通函数：定义该函数的文件内

* + C++的static除了上述两种用途，还可以修饰类成员（静态成员变量和静态成员函数），静态成员变量和静态成员函数不属于任何一个对象，是所有类实例所共有。
  + static的数据记忆性可以满足函数在不同调用期的通信，也可以满足同一个类的多个实例间的通信。
  + 未初始化时，static变量默认值为0。

## 结构体和类的区别？

* + 结构体的默认限定符是public；类是private。
  + C++中结构体也可以继承。

## malloc和new的区别？

* + malloc和free是标准库函数，支持覆盖；new和delete是运算符，并且支持重载。
  + malloc仅仅分配内存空间，free仅仅回收空间，不具备调用构造函数和析构函数功能，用malloc分配空间存储类的对象存在风险；new和delete除了分配回收功能外，还会调用构造函数和析构函数。
  + malloc和free返回的是void类型指针（必须进行类型转换），new和delete返回的是具体类型指针。

1. 指针和引用区别？    - 引用只是别名，不占用具体存储空间，只有声明没有定义；指针是具体变量，需要占用存储空间。
   * 引用在声明时必须初始化为另一变量，一旦出现必须为typename refname &varname形式；指针声明和定义可以分开，可以先只声明指针变量而不初始化，等用到时再指向具体变量。
   * 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
   * 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

## 宏定义和函数有何区别？

* + 宏在编译时完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时不存在函数调用，执行起来更快；函数调用在运行时需要跳转到具体调用函数。
  + 宏函数属于在结构中插入代码，没有返回值；函数调用具有返回值。
  + 宏函数参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
  + 宏函数不要在最后加分号。

## 宏定义和const区别？

* + 宏替换发生在编译阶段之前，属于文本插入替换；const作用发生于编译过程中。
  + 宏不检查类型；const会检查数据类型。
  + 宏定义的数据没有分配内存空间，只是插入替换掉；const定义的变量只是值不能改变，但要分配内存空间。

## 宏定义和typedef区别？

* + 宏主要用于定义常量及书写复杂的内容；typedef主要用于定义类型别名。
  + 宏替换发生在编译阶段之前，属于文本插入替换；typedef是编译的一部分。
  + 宏不检查类型；typedef会检查数据类型。
  + 宏不是语句，不在在最后加分号；typedef是语句，要加分号标识结束。
  + 注意对指针的操作，typedef char \* p\_char和#define p\_char char \*区别巨大。

## 宏定义和内联函数(inline)区别？

* + 在使用时，宏只做简单字符串替换（编译前）。而内联函数可以进行参数类型检查（编译时），且具有返回值。
  + 内联函数本身是函数，强调函数特性，具有重载等功能。
  + 内联函数可以作为某个类的成员函数，这样可以使用类的保护成员和私有成员。而当一个表达式涉及到类保护成员或私有成员时，宏就不能实现了。

## 条件编译#ifdef, #else, #endif作用？

* + 可以通过加#define，并通过#ifdef来判断，将某些具体模块包括进要编译的内容。
  + 用于子程序前加#define DEBUG用于程序调试。
  + 应对硬件的设置（机器类型等）。
  + 条件编译功能if也可实现，但条件编译可以减少被编译语句，从而减少目标程序大小。

## 区别以下几种变量？

const int a;

int const a;

const int \*a;常量指针

int \*const a;指针常量

* + int const a和const int a均表示定义常量类型a。
  + const int \*a，其中a为指向int型变量的指针，const在 \* 左侧，表示a指向不可变常量。(看成const (\*a)，对引用加const)
  + int \*const a，依旧是指针类型，表示a为指向整型数据的常指针。(看成const(a)，对指针const)

## volatile有什么作用？

* + volatile定义变量的值是易变的，每次用到这个变量的值的时候都要去重新读取这个变量的值，而不是读寄存器内的备份。
  + 多线程中被几个任务共享的变量需要定义为volatile类型。

## 什么是常引用？

* + 常引用可以理解为常量指针，形式为const typename & refname = varname。
  + 常引用下，原变量值不会被别名所修改。
  + 原变量的值可以通过原名修改。
  + 常引用通常用作只读变量别名或是形参传递。
  + 既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用

## 区别以下指针类型？

int \*p[10]

int (\*p)[10]

int \*p(int)

int (\*p)(int)

* + int \*p[10]表示指针数组，强调数组概念，是一个数组变量，数组大小为10，数组内每个元素都是指向int类型的指针变量。
  + int (\*p)[10]表示数组指针，强调是指针，只有一个变量，是指针类型，不过指向的是一个int类型的数组，这个数组大小是10。
  + int \*p(int)是函数声明，函数名是p，参数是int类型的，返回值是int \*类型的。
  + int (\*p)(int)是函数指针，强调是指针，该指针指向的函数具有int类型参数，并且返回值是int类型的。

## 常量指针和指针常量区别？

* + 常量指针是一个指针，读成常量的指针，指向一个只读变量。如int const \*p或const int \*p。
  + 指针常量是一个不能给改变指向的指针。如int \*const p。

## a和&a有什么区别？

假设数组int a[10];

int (\*p)[10] = &a;

* + a是数组名，是数组首元素地址，+1表示地址值加上一个int类型的大小，如果a的值是0x00000001，加1操作后变为0x00000005。\*(a + 1) = a[1]。
  + &a是数组的指针，其类型为int (\*)[10]（就是前面提到的数组指针），其加1时，系统会认为是数组首地址加上整个数组的偏移（10个int型变量），值为数组a尾元素后一个元素的地址。
  + 若(int \*)p ，此时输出 \*p时，其值为a[0]的值，因为被转为int \*类型，解引用时按照int类型大小来读取。

## 数组名和指针（这里为指向数组首元素的指针）区别？

* + 二者均可通过增减偏移量来访问数组中的元素。
  + 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作。
  + 当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但sizeof运算符不能再得到原数组的大小了。

## 野指针是什么？

* + 也叫空悬指针，不是指向null的指针，是指向垃圾内存的指针。
  + 产生原因及解决办法：
    - 指针变量未及时初始化 => 定义指针变量及时初始化，要么置空。
    - 指针free或delete之后没有及时置空 => 释放操作后立即置空。

## 堆和栈的区别？

* + 申请方式不同。
    - 栈由系统自动分配。
    - 堆由程序员手动分配。
  + 申请大小限制不同。
    - 栈顶和栈底是之前预设好的，大小固定，可以通过ulimit -a查看，由ulimit -s修改。
    - 堆向高地址扩展，是不连续的内存区域，大小可以灵活调整。
  + 申请效率不同。
    - 栈由系统分配，速度快，不会有碎片。
    - 堆由程序员分配，速度慢，且会有碎片。

## delete和delete[]区别？

* + delete只会调用一次析构函数。
  + delete[]会调用数组中每个元素的析构函数。

# 面向对象基础

能够准确理解下面这些问题是从C程序员向C++程序员进阶的基础。当然了，这只是一部分。

## 面向对象三大特性？

* + 封装性：数据和代码捆绑在一起，避免外界干扰和不确定性访问。
  + 继承性：让某种类型对象获得另一个类型对象的属性和方法。
  + 多态性：同一事物表现出不同事物的能力，即向不同对象发送同一消息，不同的对象在接收时会产生不同的行为（重载实现编译时多态，虚函数实现运行时多态）。

## public/protected/private的区别？

* + public的变量和函数在类的内部外部都可以访问。
  + protected的变量和函数只能在类的内部和其派生类中访问。
  + private修饰的元素只能在类内访问。

## 对象存储空间？

* + 非静态成员的数据类型大小之和。
  + 编译器加入的额外成员变量（如指向虚函数表的指针）。
  + 为了边缘对齐优化加入的padding。

## C++空类有哪些成员函数?

* + 首先，空类大小为1字节。
  + 默认函数有：6个
    - 构造函数
    - 析构函数
    - 拷贝构造函数
    - 赋值运算符
    - 取址运算符
    - 取址运算符const

## 构造函数能否为虚函数，析构函数呢？

* + 析构函数：
    - 析构函数可以为虚函数，并且一般情况下基类析构函数要定义为虚函数。
    - 只有在基类析构函数定义为虚函数时，调用操作符delete销毁指向对象的基类指针时，才能准确调用派生类的析构函数（从该级向上按序调用虚函数），才能准确销毁数据。
    - 析构函数可以是纯虚函数，含有纯虚函数的类是抽象类，此时不能被实例化。但派生类中可以根据自身需求重新改写基类中的纯虚函数。
  + 构造函数：
    - 构造函数不能定义为虚函数。在构造函数中可以调用虚函数，不过此时调用的是正在构造的类中的虚函数，而不是子类的虚函数，因为此时子类尚未构造好。

## 构造函数调用顺序，析构函数呢？

* + 调用所有虚基类的构造函数，顺序为从左到右，从最深到最浅
  + 基类的构造函数：如果有多个基类，先调用纵向上最上层基类构造函数，如果横向继承了多个类，调用顺序为派生表从左到右顺序。
  + 如果该对象需要虚函数指针(vptr)，则该指针会被设置从而指向对应的虚函数表(vtbl)。
  + 成员类对象的构造函数：如果类的变量中包含其他类（类的组合），需要在调用本类构造函数前先调用成员类对象的构造函数，调用顺序遵照在类中被声明的顺序。
  + 派生类的构造函数。
  + 析构函数与之相反。

## 拷贝构造函数中深拷贝和浅拷贝区别？

* + 深拷贝时，当被拷贝对象存在动态分配的存储空间时，需要先动态申请一块存储空间，然后逐字节拷贝内容。
  + 浅拷贝仅仅是拷贝指针字面值。
  + 当使用浅拷贝时，如果原来的对象调用析构函数释放掉指针所指向的数据，则会产生空悬指针。因为所指向的内存空间已经被释放了。

## 拷贝构造函数和赋值运算符重载的区别？

* + 拷贝构造函数是函数，赋值运算符是运算符重载。
  + 拷贝构造函数会生成新的类对象，赋值运算符不能。
  + 拷贝构造函数是直接构造一个新的类对象，所以在初始化对象前不需要检查源对象和新建对象是否相同；赋值运算符需要上述操作并提供两套不同的复制策略，另外赋值运算符中如果原来的对象有内存分配则需要先把内存释放掉。
  + 形参传递是调用拷贝构造函数（调用的被赋值对象的拷贝构造函数），但并不是所有出现"="的地方都是使用赋值运算符，如下：
  + Student s;
  + Student s1 = s; // 调用拷贝构造函数
  + Student s2;
  + s2 = s; // 赋值运算符操作

**注：类中有指针变量时要重写析构函数、拷贝构造函数和赋值运算符**

## 虚函数和纯虚函数区别？

* + 虚函数是为了实现动态编联产生的，目的是通过基类类型的指针指向不同对象时，自动调用相应的、和基类同名的函数（使用同一种调用形式，既能调用派生类又能调用基类的同名函数）。虚函数需要在基类中加上virtual修饰符修饰，因为virtual会被隐式继承，所以子类中相同函数都是虚函数。当一个成员函数被声明为虚函数之后，其派生类中同名函数自动成为虚函数，在派生类中重新定义此函数时要求函数名、返回值类型、参数个数和类型全部与基类函数相同。
  + 纯虚函数只是相当于一个接口名，但含有纯虚函数的类不能够实例化。

## 覆盖、重载和隐藏的区别？

* + 覆盖是派生类中重新定义的函数，其函数名、参数列表（个数、类型和顺序）、返回值类型和父类完全相同，只有函数体有区别。派生类虽然继承了基类的同名函数，但用派生类对象调用该函数时会根据对象类型调用相应的函数。覆盖只能发生在类的成员函数中。
  + 隐藏是指派生类函数屏蔽了与其同名的函数，这里仅要求基类和派生类函数同名即可。其他状态同覆盖。可以说隐藏比覆盖涵盖的范围更宽泛，毕竟参数不加限定。
  + 重载是具有相同函数名但参数列表不同（个数、类型或顺序）的两个函数（不关心返回值），当调用函数时根据传递的参数列表来确定具体调用哪个函数。重载可以是同一个类的成员函数也可以是类外函数。

## 在main执行之前执行的代码可能是什么？

* + 全局对象的构造函数。

## 哪几种情况必须用到初始化成员列表？

* + 初始化一个const成员。
  + 初始化一个reference（引用）成员。
  + 调用一个基类的构造函数，而该函数有一组参数。
  + 调用一个数据成员对象的构造函数，而该函数有一组参数。

## 什么是虚指针？

* + 虚指针或虚函数指针是虚函数的实现细节。
  + 虚指针指向虚表结构。

## 重载和函数模板的区别？

* + 重载需要多个函数，这些函数彼此之间函数名相同，但参数列表中参数数量和类型不同。在区分各个重载函数时我们并不关心函数体。
  + 模板函数是一个通用函数，函数的类型和形参不直接指定而用虚拟类型来代表。但只适用于参个数相同而类型不同的函数。

## this指针是什么？

* + this指针是类的指针，指向对象的首地址。
  + this指针只能在成员函数中使用，在全局函数、静态成员函数中都不能用this。
  + this指针只有在成员函数中才有定义，且存储位置会因编译器不同有不同存储位置。

## 类模板是什么？

* + 用于解决多个功能相同、数据类型不同的类需要重复定义的问题。
  + 在建立类时候使用template及任意类型标识符T，之后在建立类对象时，会指定实际的类型，这样才会是一个实际的对象。
  + 类模板是对一批仅数据成员类型不同的类的抽象，只要为这一批类创建一个类模板，即给出一套程序代码，就可以用来生成具体的类。

## 构造函数和析构函数调用时机？

* + 全局范围中的对象：构造函数在所有函数调用之前执行，在主函数执行完调用析构函数。
  + 局部自动对象：建立对象时调用构造函数，离开作用域时调用析构函数。
  + 动态分配的对象：建立对象时调用构造函数，调用释放时调用析构函数。
  + 静态局部变量对象：建立时调用一次构造函数，主函数结束时调用析构函数。

# 标准模板库

STL内容虽然看起来很多，单独成书都不是问题（《STL源码剖析》），但从实际使用状况来看，我认为只需要知道以下几点就可以了：

## 怎么用？

各种STL基本的增删改查怎么使用。每种容器都提供了很多操作，但实际增删改查我们通常只需要掌握透彻一种方式即可。有些功能只是出于通用性考虑才存在的，但对于相应的STL这些操作完全可以忽略。所以我对STL使用的看法是，不需要花太多时间去了解所有功能，只要掌握最基本的即可，要把精力放在对需求的了解并选择适合的数据结构。

## 怎么实现？

本身STL就是封装了我们常用的数据结构，所以最先需要了解每种数据结构的特性。而且了解实现方式对我们能够准确、高效使用STL打下了基础。

## 如何避免错误？

在第二阶段了解了STL的实现之后，我们已经可以很清楚地知道他们底层使用的是什么数据结构以及该数据结构做什么操作比较高效。但还有一点需要注意的就是怎么才能用对他们，避免一些未知的错误，比如迭代器失效问题。

## String、vector

用法：

定义：

vector<T> vec;

插入元素：

vec.push\_back(element);

vec.insert(iterator, element);

删除元素：

vec.pop\_back();

vec.erase(iterator);

修改元素：

vec[position] = element;

遍历容器：

for(auto it = vec.begin(); it != vec.end(); ++it) {......}

其他：

vec.empty(); //判断是否空

vec.size(); // 实际元素

vec.capacity(); // 容器容量

vec.begin(); // 获得首迭代器

vec.end(); // 获得尾迭代器

vec.clear(); // 清空

## 实现：

[模拟Vector实现](https://github.com/linw7/Skill-Tree/blob/master/code/my_vector.cpp)

* 线性表，数组实现。
  + 支持随机访问。
  + 插入删除操作需要大量移动数据。
* 需要连续的物理存储空间。
* 每当大小不够时，重新分配内存（\*2），并复制原内容。

## 错误避免：

[迭代器失效](https://github.com/linw7/Skill-Tree/blob/master/code/vector_iterator.cpp)

* 插入元素
  + 尾后插入：size < capacity时，首迭代器不失效尾迭代实现（未重新分配空间），size == capacity时，所有迭代器均失效（需要重新分配空间）。
  + 中间插入：size < capacity时，首迭代器不失效但插入元素之后所有迭代器失效，size == capacity时，所有迭代器均失效。
* 删除元素
  + 尾后删除：只有尾迭代失效。
  + 中间删除：删除位置之后所有迭代失效。

## map

用法：

定义：

map<T\_key, T\_value> mymap;

插入元素：

mymap.insert(pair<T\_key, T\_value>(key, value)); // 同key不插入

mymap.insert(map<T\_key, T\_value>::value\_type(key, value)); // 同key不插入

mymap[key] = value; // 同key覆盖

删除元素：

mymap.erase(key); // 按值删

mymap.erase(iterator); // 按迭代器删

修改元素：

mymap[key] = new\_value;

遍历容器：

for(auto it = mymap.begin(); it != mymap.end(); ++it) {

cout << it->first << " => " << it->second << '\n';

}

实现：

[RBTree实现](https://github.com/linw7/Skill-Tree/tree/master/code/RBTree)

* 树状结构，RBTree实现。
  + 插入删除不需要数据复制。
  + 操作复杂度仅跟树高有关。
* RBTree本身也是二叉排序树的一种，key值有序，且唯一。
  + 必须保证key可排序。

基于红黑树实现的map结构（实际上是map, set, multimap，multiset底层均是红黑树），不仅增删数据时不需要移动数据，其所有操作都可以在O(logn)时间范围内完成。另外，基于红黑树的map在通过迭代器遍历时，得到的是key按序排列后的结果，这点特性在很多操作中非常方便。

面试时候现场写红黑树代码的概率几乎为0，但是红黑树一些基本概念还是需要掌握的。

1. 它是二叉排序树（继承二叉排序树特显）：
   * 若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值。
   * 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值。
   * 左、右子树也分别为二叉排序树。
2. 它满足如下几点要求：
   * 树中所有节点非红即黑。
   * 根节点必为黑节点。
   * 红节点的子节点必为黑（黑节点子节点可为黑）。
   * 从根到NULL的任何路径上黑结点数相同。
3. 查找时间一定可以控制在O(logn)。
4. 红黑树的节点定义如下：

enum Color {

RED = 0,

BLACK = 1

};

struct RBTreeNode {

struct RBTreeNode\*left, \*right, \*parent;

int key;

int data;

Color color;

};

所以对红黑树的操作需要满足两点：1.满足二叉排序树的要求；2.满足红黑树自身要求。通常在找到节点通过和根节点比较找到插入位置之后，还需要结合红黑树自身限制条件对子树进行左旋和右旋。

相比于AVL树，红黑树平衡性要稍微差一些，不过创建红黑树时所需的旋转操作也会少很多。相比于最简单的BST，BST最差情况下查找的时间复杂度会上升至O(n)，而红黑树最坏情况下查找效率依旧是O(logn)。所以说红黑树之所以能够在STL及Linux内核中被广泛应用就是因为其折中了两种方案，既减少了树高，又减少了建树时旋转的次数。

从红黑树的定义来看，红黑树从根到NULL的每条路径拥有相同的黑节点数（假设为n），所以最短的路径长度为n（全为黑节点情况）。因为红节点不能连续出现，所以路径最长的情况就是插入最多的红色节点，在黑节点数一致的情况下，最可观的情况就是黑红黑红排列......最长路径不会大于2n，这里路径长就是树高。

**set**

# 编译及调试

## 编译

### 预处理

* 展开所有的宏定义，完成字符常量替换。
* 处理条件编译语句，通过是否具有某个宏来决定过滤掉哪些代码。
* 处理#include指令，将被包含的文件插入到该指令所在位置。
* 过滤掉所有注释语句。
* 添加行号和文件名标识。
* 保留所有#pragma编译器指令。

### 编译

* 词法分析。
* 语法分析。
* 语义分析。
* 中间语言生成。
* 目标代码生成与优化。

### 链接

各个源代码模块独立的被编译，然后将他们组装起来成为一个整体，组装的过程就是链接。被链接的各个部分本本身就是二进制文件，所以在被链接时需要将所有目标文件的代码段拼接在一起，然后将所有对符号地址的引用加以修正。

#### 静态链接

静态链接最简单的情况就是在编译时和静态库链接在一起成为完整的可执行程序。这里所说的静态库就是对多个目标文件（.o）文件的打包，通常静态链接的包名为lib\*\*\*\*.a，静态链接所有被用到的目标文件都会复制到最终生成的可执行目标文件中。这种方式的好处是在运行时，可执行目标文件已经完全装载完毕，只要按指令序执行即可，速度比较快，但缺点也有很多，在讲动态链接时会比较一下。

既然静态链接是对目标文件的打包，这里介绍些打包命令。

gcc -c test1.c // 生成test1.o

gcc -c test2.c // 生成test2.c

ar cr libtest.a test1.o test2.o

首先编译得到test1.o和test2.o两个目标文件，之后通过ar命令将这两个文件打包为.a文件，文件名格式为lib + 静态库名 + .a后缀。在生成可执行文件需要使用到它的时候只需要在编译时加上即可。需要注意的是，使用静态库时加在最后的名字不是libtest.a，而是l + 静态库名。

gcc -o main main.c -ltest

#### 动态链接

静态链接发生于编译阶段，加载至内存前已经完整，但缺点是如果多个程序都需要使用某个静态库，则该静态库会在每个程序中都拷贝一份，非常浪费内存资源，所以出现了动态链接的方式来解决这个问题。

动态链接在形式上倒是和静态链接非常相似，首先也是需要打包，打包成动态库，不过文件名格式为lib + 动态库名 + .so后缀。不过动态库的打包不需要使用ar命令，gcc就可以完成，但要注意在编译时要加上-fPIC选项，打包时加上-shared选项。

gcc -fPIC -c test1.c

gcc -fPIC -c test2.c

gcc -shared test1.o test2.o -o libtest.so

使用动态链接的用法也和静态链接相同。

gcc -o main main.c -ltest

如果仅仅像上面的步骤是没有办法正常使用库的，我们可以通过加-Lpath指定搜索库文件的目录（-L.表示当前目录），默认情况下会到环境变量LD\_LIBRARY\_PATH指定的目录下搜索库文件，默认情况是/usr/lib，我们可以将库文件拷贝到那个目录下再链接。

比较静态库和动态库我们可以得到二者的优缺点。

* 动态库运行时会先检查内存中是否已经有该库的拷贝，若有则共享拷贝，否则重新加载动态库（C语言的标准库就是动态库）。静态库则是每次在编译阶段都将静态库文件打包进去，当某个库被多次引用到时，内存中会有多份副本，浪费资源。
* 动态库另一个有点就是更新很容易，当库发生变化时，如果接口没变只需要用新的动态库替换掉就可以了。但是如果是静态库的话就需要重新被编译。
* 不过静态库也有优点，主要就是静态库一次性完成了所有内容的绑定，运行时就不必再去考虑链接的问题了，执行效率会稍微高一些。

## makefile编写

对于大的工程通常涉及很多头文件和源文件，编译起来很很麻烦，makefile正是为了自动化编译产生的，makefile像是编译说明书，指示编译的步骤和条件，之后被make命令解释。

* 基本规则
* A:B
* (tab)<command>

其中A是语句最后生成的文件，B是生成A所依赖的文件，比如生成test.o依赖于test.c和test.h，则写成test.o:test.c test.h。接下来一行的开头必须是tab，再往下就是实际的命令了，比如gcc -c test.c -o test.o。

* 变量

makefile的书写非常像shell脚本，可以在文件中定义"变量名 = 变量值"的形式，之后需要使用这个变量时只需要写一个$符号加上变量名即可，当然，和shell一样，最好用()包裹起语句来。

## 链接

### 符号解析

* 可重定位目标文件

对于独立编译的可重定位目标文件，其ELF文件格式包括ELF头（指定文件大小及字节序）、.text（代码段）、.rodata（只读数据区）、.data（已初始化数据区）、.bss（未初始化全局变量）、.symtab（符号表）等，其中链接时最需要关注的就是符号表。每个可重定位目标文件都有一张符号表，它包含该模块定义和引用的符号的信息，简而言之就是我们在每个模块中定义和引用的全局变量（包括定义在本模块的全局变量、静态全局变量和引用自定义在其他模块的全局变量）需要通过一张表来记录，在链接时通过查表将各个独立的目标文件合并成一个完整的可执行文件。

* 解析符号表

解析符号引用的目的是将每个引用与可重定位目标文件的符号表中的一个符号定义联系起来。

### 重定位

* 合并节

多个可重定位目标文件中相同的节合并成一个完整的聚合节，比如多个目标文件的.data节合并成可执行文件的.data节。链接器将运行时存储地址赋予每个节，完成这步每条指令和全局变量都有运行时地址了。

* 重定位符号引用

这步修改全部代码节和数据节对每个符号的符号引用，使其指向正确的运行时地址。局部变量可以通过进栈、出栈临时分配，但全局变量（"符号"）的位置则是在各个可重定位目标文件中预留好的。通过上一步合并节操作后，指令中所有涉及符号的引用都会通过一定的寻址方式来定位该符号，比如相对寻址、绝对寻址等。

### 可执行目标文件

* ELF头部

描述文件总体格式，并且包括程序的入口点（entry point），也就是程序运行时执行的第一条指令地址。

* 段头部表

描述了可执行文件数据段、代码段等各段的大小、虚拟地址、段对齐、执行权限等。实际上通过段头部表描绘了虚拟存储器运行时存储映像，比如每个UNIX程序的代码段总是从虚拟地址Ox0804800开始的。

* 其他段

和可重定位目标文件各段基本相同，但完成了多个节的合并和重定位工作。

### 加载

* 克隆

新程序的执行首先需要通过父进程外壳通过fork得到一个子进程，该子进程除了pid等标识和父进程不同外其他基本均与父进程相同。

* 重新映射

当子进程执行execve系统调用时会先清空子进程现有的虚拟存储器段（简而言之就是不再映射到父进程的各个段），之后重新创建子进程虚拟存储器各段和可执行目标文件各段的映射。这个阶段我们可以理解为对复制来的父进程页表进程重写，映射到外存中可执行文件的各个段。

* 虚页调入

加载过程并没有实际将磁盘中可执行文件调入内存，所做的工作紧紧是复制父进程页表、清空旧页表、建立新页表映射工作。之后加载器跳转到入口地址\_start开始执行程序，接下来的过程需要配合虚拟存储器来完成。CPU获得指令的虚拟地址后，若包含该指令或数据的页尚未调入内存则将其从外存中调入，调入内存后修改页表得到虚拟页号和物理页号的对应关系。之后重新取同一条指令或数据时因该页已经被调入内存，所以通过虚拟地址得到虚拟页号，虚拟页号通过查页表可以得到物理页号，通过物理页号 + 页内偏移得到具体的物理地址，此时可以通过物理地址取得想要的数据。