C++问题

1、STL 中关系型容器底层为什么使用红黑树而不是其它的平衡二叉树

- 其它的平衡二叉树主要是 **AVL 树**，AVL 树是高度平衡的树，其结构相较 **红黑树** 来说更为平衡，在插入和删除结点更容易引起树的 **不平衡。**因此在大量数据需要插入或者删除时，AVL 树需要 **rebalance** 的频率会更高，而 **红黑树** 在这类场景下，效率更高。
- 如果插入一个结点引起了树的不平衡，**AVL 树** 和 **红黑树** 都最多只需要 **2 次旋转操作**，即两者旋转操作的时间复杂度都是 $O(1)$；但在删除结点引起树的不平衡时，最坏情况下，AVL 树需要维护从

被删结点到根这条路径上所有节点的平衡性，因此旋转操作的时间复杂度量级是 $O(log(N))$，而红黑树删除结点时最多只需要 **3 次旋转**，即旋转操作时间复杂度为 $O(1)$；
- STL 中容器实现时折衷了两者在search、insert以及delete下的效率。总体来说，**红黑树的统计性能是高于 AVL 树的。**

2、C++ 多重继承及其优缺点?

- 多重继承的派生类继承了所有父类的属性。派生类的对象包含有每个基类的子对象。
- 构造一个派生类对象将同时构造并初始化它的所有 **直接基类** 的子对象。基类的构造顺序 **与派生列表中的顺序保持一致。**析构函数的调用顺序正好相反。
- 允许派生类从它的一个或几个基类中继承构造函数，但如果从多个基类中继承了相同的构造函数（即形参列表完全相同），会产生错误。如果一个类从它的多个基类中继承了相同的构造函数，则这个类必须为该构造函数 **定义它自己的版本。**
- 编译器不会在派生类向基类的几种转换中进行比较和选择，在它看来转换到任意一种基类都一样好。
- 派生类可能从两个或更多基类中继承了同名成员，此时，不加前缀限定符直接使用会引发二义性错误，即使派生类继承的两个函数形参列表不同也可能发生错误。此外，即使同名函数在一个基类中是私有的，而在另一个基类中是公有的或受保护的，也会发生错误。因为 **先查找名字后进行类型检查，当编译器在两个作用域中同时发现了同名成员时，将直接报告一个二义性错误。**要避免这类错误，最好的办法时在派生类中为该函数 **定义一个新版本。**
- 默认情况下，派生类中含有继承链上每个类对应的子部分。如果某个类在派生过程中出现了多次，则派生类中将包含该类的多个子对象。这对于某些类是行不通的。C++ 通过 **虚继承** 的机制解决上述问题。虚继承的目的是令某个类做出声明。承诺愿意共享它的基类。其中，共享的基类子对象称为 **虚基类。**
- 虚派生只影响从指定了虚基类的派生类中进一步派生出的类，他不会影响派生类本身。
- 派生类的成员比共享虚基类中的同名成员优先级更高；但如果共享虚基类的成员被多于一个基类覆盖，则派生类必须为该成员定义一个新版本，否者会产生二义性错误。
- 在虚派生中，虚基类是由 **最底层的派生类初始化的** 并且在构造顺序中也是 **最优先的。**
- 一个类可以有多个虚基类。此时，这些虚的子对象按照它们在派生列表中出现的顺序 **从左向右** 构造。

虚继承

- 虚继承是解决 C++ 多重继承问题的一种手段，从不同途径继承来的同一基类，会在子类中存在多份拷贝。这将存在两个问题：其一，浪费存储空间；第二，存在二义性问题；
- 虚继承底层实现原理与编译器相关，一般通过虚基类指针和虚基类表实现，每个虚继承的子类都有一个虚基类指针和虚基类表；当虚继承的子类被当做父类继承时，虚基类指针也会被继承

3、在构造函数和析构函数中使用虚函数会发生什么?

- 一个非正式的说法是：**在基类构造期间，虚函数不是虚函数**；
- 更根本的原因是：
    - **在派生类对象的基类构造期间，对象的类型是基类而不是派生类。**不只虚函数会被编译器解析为基类，若使用运行期类型信息（dynamic_cast、typeid），也会把对象视为基类类型。
- 析构函数也是如此，一旦派生类的析构函数开始执行，对象内的派生类成员变量便呈现未定义值，所以C++视他们仿佛不存在。
- 如果一定要在构造期间实现虚函数的动态绑定功能，可以在基类中将成员函数设为非虚函数，然后通过派生类向基类传递必要信息来实现。

4、野指针/空悬指针

- 野指针不是空指针，而是指向一个垃圾内存的指针，或者说是指向一块曾经保存数据对象但现在已经无效的内存的指针。
- 形成原因：
  - 指针变量没有被初始化；任何指针变量刚被创建时不会自动成为NULL指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为NULL，要么让它指向合法的内存。
  - 指针 p 被 free 或者 delete 之后，没有置为NULL，让人误以为 p 是个合法的指针。free 和 delete 只是把指针所指的内存给释放掉，但并没有把指针本身干掉。free 以后其地址仍然不变。
  - 指针操作 **超越了变量的作用范围。**返回局部变量的地址，指针越界等。
- 动态内存的一个基本问题是可能有多个指针指向相同的内存，在 `delete` 内存之后重置指针只对这一个指针有效，对其他任何仍指向（已释放的）内存的指针是没有作用的。

5、sizeof

- 其作用是返回一个对象或类型所占的内存字节数。`sizeof` 对对象求内存大小，最终都是转换为对对象的数据类型进行求值。
- 结构体的 `sizeof` 涉及到 **字节对齐问题。**字节对齐有助于加快计算机的取数速度。因此，编译器会对结构体进行处理，让宽度为 2 的基本数据类型（`short` 等）都位于能被 2 整除的地址上，让宽度为 4 的基本数据类型（`int` 等）都位于能被 4 整除的地址上，依次类推。这样，两个数中间就可能需要加入填充字节，所以整个结构体的 `sizeof` 值就增长了：
  - 结构体变量的首地址能够被其最宽基本类型成员的大小所整除；
  - 结构体的每个成员相对于结构体首地址的偏移量（offset）都是成员大小的整数倍，如有需要，编译器会在成员之间加上填充字节；
  - 结构体的总大小为结构体 **最宽基本类型成员大小的整数倍**，如有需要，编译器会在最末一个成员后加上填充字节。
- 空结构体（不含数据成员）的 `sizeof` 值为1。试想一个“不占空间”的变量如何被取地址、两个不同的“空结构体”变量又如何得以区分呢，于是，“空结构体”变量也得被存储，这样编译器也就只能为其分配一个字节的空间用于占位了。只包含成员函数的类，其 `sizeof` 值也为 1， 因为成员函数只与类型相关，而与具体实例无关。
- 对象的存储空间等于 **非静态成员变量总和加上编译器为了 CPU 计算做出的数据对齐处理和支持虚函数所产生的负担的总和。**
- 联合体的 `sizeof` 是每个成员 `sizeof` 的最大值。
- 数组的 `sizeof` 值等于数组所占用的内存字节数：
  - 当字符数组表示字符串时，其 `sizeof` 值 **将** `'/0'` **计算进去**；
  - 当 **数组为形参** 时，其 `sizeof` 值相当于 **指针的** `sizeof` **值。**
- 指针的内存大小等于计算机内部 **地址总线的宽度**，在 32 位计算机中，一个指针变量的大小为 4。指针变量的 `sizeof` 值 **与指针所指的对象没有任何关系。**
- `sizeof` 也可对一个函数调用求值，其结果是 **函数返回值类型的大小**，函数并不会被调用。
  - 不可以对返回值类型为空的函数求值；
  - 不可以对函数名求值；
  - 对有参数的函数求值时，需写上实参表。

6、对象指针为NULL，为什么还是可以调用成员函数

- 类的成员函数并不与具体对象绑定，所有的对象共用同一份成员函数体，当程序被编译后，成员函数的地址即已确定，这份共有的成员函数体之所以能够把不同对象的数据区分开来，靠的是隐式传递给成员函数的 this 指针，成员函数中对成员变量的访问都是转化成 **"this->数据成员"** 的方式。因此，从这一角度说，成员函数与普通函数一样，只是多了一个隐式参数，即指向对象的 this 指针。
- 当调用一个成员函数时，是用请求该函数的对象地址初始化 **this指针**。
- 类的静态成员函数只能访问静态成员变量，不能访问非静态成员变量，所以静态成员函数不需要指向对象的this指针作为隐式参数。
- 有虚函数的类会有一个成员变量，即虚表指针，当调用虚函数时，会使用虚表指针，对虚表指针的使用也是通过隐式指针使用的。

7、C++ 构造函数为什么不能是虚函数?

- 虚函数调用是在部分信息下完成工作的机制，它允许我们只知道接口而不知道对象的确切类型。要创建一个对象，你需要知道对象的完整信息。特别是，你需要知道你想要创建的确切类型。因此，构造函数不应该被定义为虚函数。
- 虚函数对应一个指向虚函数表 `vtable` 的指针，但是这个虚指针事实上是存储在对象的内存空间的。假设构造函数是虚函数，就需要虚函数表来调用，但是对象还没有实例化，即还没有内存空间，因此就没有初始化虚指针，找不到虚函数表。所以构造函数不能是虚函数。

8、面向对象程序设计

- **数据抽象、继承和动态绑定** 是其三个基本概念。
- 类的基本思想是 **数据抽象** 和 **封装**。数据抽象是一种依赖于 **接口** 和 **实现** 分离的编程技术。封装则实现了类的接口和实现的分离。类的接口包括用户所能执行的操作；类的实现则包括类的数据成员、负责接口实现的函数体以及定义类所需的各种私有函数。
- **继承** 是我们可以更容易地定义与其它类相似但不完全相同的新类；
- **动态绑定** 使我们在使用这些彼此相似的类编写程序时，可以在一定程度上忽略它们的区别；
- 在 C++ 中，基类将类型相关的函数与派生类不做改变直接继承的函数区分对待；对于某些函数，基类希望它的派生类各自定义适合自身的版本，此时基类就将这些函数声明成 **虚函数**。
- **多态性** 是 OOP 的核心思想之一。把具有继承关系的多个类型称为多态类型，我们可以使用这些类型的 多种形式 而无须在意它们的差异。**引用或指针的静态类型与动态类型不同** 这一事实正是 C++ 支持多态性的根本所在。

9、string 类

- 由 C++ 字符串转换成对应的 C 字符串的方法是使用 `data()`、`c_str()` 和 `copy()` 来实现。`data()` 以字符数组的形式返回字符串内容，但并不添加 `\0`； `c_str()` 返回一个以 `\0` 结尾的字符数组；而 `copy()` 则把字符串的内容复制或写入既有的 `c_string` 或字符数组内。
- 使用 `c_str()` 和 `data()` 得到字符串 `str` 对应的字符数组 `cstr` 后，改变字符串 `str` 的内容，`cstr` 的内容也会随着改变。

10、红黑树

- 红黑树确保没有一条路径会比其它路径长出两倍，因而是接近平衡的；
- 红黑树左旋、右旋操作：

```
inline void _rb_tree_rotate_left(_tb_tree_node_base *x, _rb_tree_node_base
*&root) {
    // x 为旋转点
    _rb_tree_node_base *y = x->right;
```

```
    x->right = y->left;
    if (y->left != 0)
        y->left->parent = x;
    y->parent = x->parent;

    if (x == root)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;

    x->parent = y;
    y->left = x;
}

inline void _rb_tree_rotate_right(_tb_tree_node_base *x, _rb_tree_node_base
*&root) {
    // x 为旋转点
    _rb_tree_node_base *y = x->left;
    x->left = y->right;
    if (y->right != 0)
        y->right->parent = x;
    y->parent = x->parent;

    if (x == root)
        root = y;
    else if (x == x->parent->right)
        x->parent->right = y;
    else
        x->parent->left = y;

    y->right = x;
    x->parent = y;
}
```

11、set

- 对于关联容器而言，插入和删除的一切操作都是指针指向变化，不需要做内存拷贝和内存移动，所以 `map` 和 `set` 的插入删除效率比其他序列容器高。
- `set` 和 `map` 的迭代器相当于指向结点的指针，插入操作不会改变已有指针指向的内存。
- 在 `set` 中查找使用的是 **二分查找**，即 $\log N$ 的时间复杂度。当数据量增大了一倍时，搜索次数也只是多了一次。

12、关联容器中插入重复键值的处理方法?

- `map` 和 `set` 均调用底层 **红黑树** 的 `insert_unique` 函数，若检查发现键值重复，直接返回：

```
return pair<iterator, bool>(j, false);
```

- `multimap` 和 `multiset` 则调用底层 **红黑树** 的 `insert_equal` 函数，允许节点键值重复，插入位置在已有重复键值节点的右子树上：

```
iterator insert_equal(const Value &val) {
    link_type y = header;
    link_type x = root();
    while(x != 0) {
        y = x;
        x = key_compare(KeyOfValue()(v), key(x))? left(x) : right(x);
        // 遇“大”往左，遇“小于或等于”则往右
    }
    return __insert(x, y, v);
    // x 为新值插入点，y 为插入点的父节点，v 为新值
}
```

- `hash_set` 和 `hash_map` 调用底层 `hash_table` 的 `insert_unique` 函数，若检查发现键值重复，就不插入，立刻返回：

```
return pair<iterator, bool>(iterator(cur, this), false);
```

- `hash_multimap` 和 `hash_multiset` 则是调用底层 `hash_table` 的 `insert_equal` 函数，允许节点键值重复，插入位置在已有重复键值节点之后：

```
if (equals(get_key(cur->val), get_key(obj))) {
    node *tmp = new_node(obj);
    tmp->next = cur->next;
    cur->next = tmp;
    ++num_elements;
    return iterator(tmp, this);
}
```

13、字符数组和字符指针的区别：

- 字符指针是一个变量，可以改变它使它指向不同的字符串，但不能改变其所指的字符串常量
- 字符数组是一个数组，可以改变数组中保存的内容。

14、extern 关键字的用法

- `extern` 是指全局的意思，一般有两个用途
- **声明变量**：`extern` 声明不是定义，也不分配存储空间。事实上，它只是说明变量定义在程序的其他地方。程序中变量可以声明多次，但只能定义一次；只有当 `extern` 声明位于函数外部时，才可以含有初始化式。同理，不同编译方式，如 C 编译和 C++ 编译对变量的编译命名不同，`extern C` 就是告诉编译器要以 C 编译方式来命名变量，避免 **因符号修饰导致 C++ 代码不能和C语言库中的符号进行链接的问题**；
- **声明 const 常量**：默认情况下，`const` 常量仅在文件内有效，当多个文件出现了同名的 `const` 变量时，相当于在不同文件中分别定义了独立的变量；通过指定 `const` 变量为 `extern`，就可以在多个文件之间共享 `const` 对象；
- **显式实例化模板**：解决在多个文件中实例化相同模板的开销；当编译器遇到 `extern` 模板声明时，它不会在本文件中生成实例化代码；将一个实例化声明为 `extern` 表示承诺在程序其他位置有该实例化的一个非 `extern` 声明（定义）。

15、new 和 malloc 的区别

- new 操作符从 **自由存储区（free store）**上为对象动态分配内存空间，而 malloc 函数从 **堆上** 动态分配内存；特别的，new 甚至可以不为对象分配内存，如 **定位 new**；
- new 操作符内存分配成功时，返回的是 **对象类型的指针**，类型严格与对象匹配，无须进行类型转换，故 new 是符合类型安全性的操作符；malloc 内存分配成功则是返回 **void *** ，需要通过强制类型转换将 void* 指针转换成我们需要的类型；
- new内存分配失败时，会抛出 **bac_alloc 异常**，它不会返回NULL；malloc 分配内存失败时返回 **NULL**；
- 使用 new 操作符申请内存分配时无须指定内存块的大小，编译器会 **根据类型信息自行计算**；malloc 则需要 **显式地指出所需内存的尺寸**；
- new 操作符来分配对象内存时会经历三个步骤：
  - 调用operator new 函数（对于数组是operator new[]）分配一块足够大的，原始的，未命名的内存空间以便存储特定类型的对象；
  - 编译器运行相应的构造函数以构造对象，并为其传入初值；
  - 对象构造完成后，返回一个指向该对象的指针；
- delete 操作符来释放对象内存时会经历两个步骤：
  - 调用对象的析构函数；
  - 编译器调用operator delete(或operator delete[])函数释放内存空间；
- malloc/free 来处理 C++ 的自定义类型不合适，其实不止自定义类型，标准库中凡是需要构造/析构的类型通通不合适；
- new 对数组的支持体现在它会分别调用构造函数函数初始化每一个数组元素，释放对象时为每个对象调用析构函数；至于 malloc，它并知道你在这块内存上要放的数组还是啥别的东西，反正它就给你一块原始的内存，再给你个内存的地址就完事；
- operator new /operator delete 的实现可以基于 malloc，而 malloc 的实现不可以去调用 new；
- opeartor new /operator delete 可以被重载，而 malloc/free 并不允许重载；
- 使用 malloc 分配的内存后，如果在使用过程中发现内存不足，可以使用 realloc 函数进行内存重新分配实现内存的扩充；
- 在 operator new 抛出异常以反映一个未获得满足的需求之前，它会先调用一个用户指定的 **错误处理函数**，这就是 **new-handler**；对于 malloc，客户并不能够去编程决定内存不足以分配时要干什么事，只能看着 malloc 返回 NULL。

```
void *operator new(size_t size) {
    if (size == 0)
        size = 1;
    while(1) {
        if (void *mem = malloc(size))
            return mem;

        new_handler globalHandler = set_new_handler(0);
        set_new_handler(globalHandler);

        if (globalHandler)
            (*globalHandler)();
        else
            throw bad_alloc();
    }
}

void *operator delete(void *mem) nonexcept {
```

```
    if (mem == 0)
        return;

    free(mem);
}
```

16、拷贝构造函数中参数为什么使用引用?

- 在函数调用过程中，具有非引用类型的参数要进行拷贝初始化；
- 拷贝构造函数就是被用来初始化非引用类型的参数
- 如果其参数不是引用类型，则调用永远不会成功：为了调用拷贝构造函数，我们必须拷贝它的实参，但为了拷贝实参，又需要调用拷贝构造函数，如此无限循环。

17、析构函数能不能是析构函数?

- 能
- 继承关系对基类拷贝控制最直接的影响是基类通常应该定义一个 **虚析构函数**；
- 如果基类的析构函数不是虚函数，则 `delete` 一个指向派生类对象的基类指针将产生未定义的行为；

18、map 为什么用红黑树不用 B+ 树？相反，Mysql 为什么用 B+ 树不用红黑树或者 B 树?

- 树看重两个性能 **插入和查找。**插入时有可能要调整树的结构重新平衡树，B+树 调整树的结构慢一些，所以B+树插入慢，查找快；红黑树插入快 ，查找慢。
- 在 `map` 中， 核心不是查询效率，而是在修改的效率。`map` 的元素是动态修改的，红黑树修改元素，不用移动元素的位置，因为直接修改左右父指针的值就好了，但是B+树的同一个块内的元素，没有指针这个东西，所以修改元素就需要 **移动元素**，这个开销就很大了，也会 **触发频繁内存分配和回收**；
- Mysql 中，B+ 树是用来充当索引的，一般来说索引非常大，尤其是关系性数据库这种数据量大的索引能达到亿级别，所以为了减少内存的占用，索引也会被 **存储在磁盘** 上，影响磁盘数据读取效率的一个重要因素是 **磁盘 IO 次数。**B+ 树分支多，结构矮胖；B+ 树除了叶子节点其它节点并不存储数据，节点小，磁盘IO次数就少，进一步地，B+ 树一个叶节点可以存储多个元素，**相对于红黑树的树高更低**，磁盘 IO 次数更少；
- 相较于 B 树，**B+ 树的内部节点并没有指向关键字具体信息的指针。**因此其内部节点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说 **IO读写次数更低**；
- **B+ 树的查询效率更加稳定**；由于非叶子结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当；
- **B+ 树元素遍历效率高**；B 树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+ 树应运而生。B+ 树只要遍历叶子节点就可以实现整棵树的遍历。而且**在数据库中基于范围的查询是非常频繁的**，而 B 树不支持这样的操作（或者说效率太低）。

19、static 关键字的作用

- **隐藏**：当同时编译多个文件时，所有未加 static 前缀的全局变量和函数都具有全局可见性；如果加了static，就会对其它源文件隐藏，利用这一特性可以在不同的文件中定义同名函数和同名变量，而不必担心命名冲突；
- **保持变量内容的持久**：如果作为static局部变量在函数内定义，它的生存期为整个源程序，但是其作用域仍与自动变量相同，只能在定义该变量的函数内使用该变量。退出该函数后， 尽管该变量还继续存在，但不能使用它；

- 修饰 **全局变量** 时，表明一个全局变量只对定义在同一文件中的函数可见；修饰 **局部变量**时，表明该变量的值不会因为函数终止而丢失；修饰 **函数** 时，表明该函数只在同一文件中调用；修饰 **类的数据成员**，表明对该类所有对象，这个数据成员都只有一个实例。即该实例归所有对象共有；修饰不访问非静态数据成员的 **类成员函数**，这意味着一个静态成员函数只能访问它的参数、类的静态数据成员和全局变量；
- 在头文件把一个变量申明为static变量，那么引用该头文件的源文件能够访问到该变量吗？
    - 可以。声明static变量一般是为了在本cpp文件中的static变量不能被其他的cpp文件引用，但是对于头文件，因为cpp文件中包含了头文件，故相当于该static变量在本cpp文件中也可以被见到。**当多个cpp文件包含该头文件中，这个static变量将在各个cpp文件中将是独立的，彼此修改不会对相互有影响。**
- 为什么静态成员函数不能申明为 const?
    - const 修饰符用于表示函数不能修改成员变量的值，该函数必须是含有this指针的类成员函数，而一个静态成员函数访问的值是其参数、静态数据成员和全局变量，而这些数据都不是对象状态的一部分；
- 为什么不能在类的内部定义以及初始化 static 成员变量，而必须要放到类的外部定义？
    - 因为静态成员属于整个类，而不属于某个对象，如果在类内初始化，会导致每个对象都包含该静态成员，这是矛盾的；
- static 关键字为什么只能出现在类内部的声明语句中，而不能重复出现在类外的定义中？
    - 如果类外定义函数时在函数名前加了static，因为作用域的限制，就只能在当前cpp里用，类本来就是为了给程序里各种地方用的，其他地方使用类是包含类的头文件，而无法包含类的源文件；
- 为什么常量静态成员数据的初始化可以放在类内？
- 为什么静态成员函数只能访问静态成员变量？
    - 静态成员函数只属于类本身，随着类的加载而存在，不属于任何对象，是独立存在的；
    - 非静态成员当且仅当实例化对象之后才存在，静态成员函数产生在前，非静态成员函数产生在后，故不能访问；
    - 内部访问静态成员用 `self::`，而访问非静态成员要用 `this指针`，静态成员函数没有this指针，故不能访问；
- 静态成员函数与非静态成员函数的区别:
    - 根本区别：静态成员函数不存在this指针，不能访问非静态成员变量。
- 为什么要用得静态成员变量和静态成员函数:
    - 为了实现共享。因为静态成员函数和静态成员变量属于类，不属于类的实体，这样可以被多个对象所共享;
- 静态成员函数的作用、优点：
    - 静态成员函数主要为了调用方便，不需要生成对象就能调用。

20、用 C 实现 OOP

- 用结构体 `struct` 实现封装；
- 用结构体组合，子类 `struct` 的成员包含父类结构体；
- 用 **函数指针** 实现多态

```
struct Bird {
    void (*print)(void *p);
};

struct fBird {
    struct Bird p;
};
```

```
void printBird(void *Bird) {
    if (NULL == Bird)
        return;
    struct Bird *p = (struct Bird*)Bird;
    printf("run in the Bird!\n");
}

void printfBird(void *Bird) {
    if (NULL == Bird)
        return;
    struct Bird *p = (struct Bird*)Bird;
    printf("run in the fBird!\n");
}

void print(void *Bird){
    if(NULL == Bird)
        return ;
    struct Bird *p = (struct Bird *)Bird;
    p->print(Bird);
}
int main(){
    struct Bird bird;
    struct fBird fbird;
    Bird.print = printBird;
    fBird.p.print = printfBird;

    print(&bird);     //实参为Bird的对象
    print(&fbird);  //实参为fBird的对象

    return 0;
}
```

无论是 fBird 还是 Bird，他们在内存中只有一个变量，就是那个函数指针，而 void 表示任何类型的指针，当我们将它强制转换成 struct Bird 类型时，p->print 指向的自然就是传入实参的 print 地址。

21、使用位运算替代模运算

- 使用位运算替代取模运算效率高，但位运算只能在特定场景下才能替代%运算；
- `&` 操作用了：`3mov+1and+1sub`， `%`操作用了：`2mov+1cdp+1div`；前者只需要 5 个 CPU 时钟周期，后者则需要至少 26 个 CPU 时钟周期；
- 正常情况下：$a\%b = a - (a/b) * b$;
- 但如果 b 的值为2的n次方的时候（n为自然数），这时候就可以用位运算来替代模运算， 转化如下：

$$a\%b = a\&(b-1)$$

22、虚函数实现机制

- C++ 分为 **静态多态** 和 **动态多态**：
    - 静态多态就是重载，因为在 **编译期** 决议确定，所以称为静态多态。在编译时就可以确定函数地址；
    - 动态多态就是通过继承重写基类的虚函数实现的多态，因为实在 **运行时** 决议确定，所以称为动态多态。运行时在虚函数表中寻找调用函数的地址。
- 同一个类的不同实例 **共用同一份虚函数表**, 它们都通过一个所谓的虚函数表指针 **__vfptr(定义为 void 类型)** 指向该虚函数表；

- 编译器在 **编译时期** 为我们创建好的, 只存在一份；定义类对象时, 编译器自动将类对象的 `__vfptr` 指向这个虚函数表；目前 gcc 和微软的编译器都是将 `__vptr` 放在对象内存布局的最前面；
- 虚函数指针是在构造函数执行时初始化的；
- 虚函数表存储在进程的 **只读数据段**。

23、函数模板与函数重载

- 函数重载用于定义功能相似的同名函数，提高函数的易用性；函数模板则用于为实现逻辑一样只是参数类型不同的一类函数提供统一的模板，提高函数编写的效率；
- 函数重载要求参数个数或类型不同；函数模板则要求参数个数必须一样；
- 函数模板也可以进行重载。

24、左值与右值；

- 左值是可寻址的变量，有持久性；
- 右值一般是不可寻址的常量，或在表达式求值过程中创建的无名临时对象，短暂性的；
- 左值和右值主要的区别之一是左值可以被修改，而右值不能；
- **左值引用**：引用一个对象；
- **右值引用**：就是 **必须绑定到右值的引用**，C++11中右值引用可以实现“移动语义”，通过 && 获得右值引用。

```
int x = 6;
int &y = x;

int &z1 = x*6;          // 错误，x*6 是一个右值
const int &z2 = x*6;    // 正确，可以将一个 const 引用绑定到一个右值

int &&z3 = x*6;         // 右值引用
int &&z4 = x;           // 错误，x 是一个左值
```

25、为什么右值引用能减少内存拷贝？

- 当用临时对象构造新对象时，编译器会调用这个类的 **移动构造函数** 和 **移动赋值函数**。因为我们知道传入的是临时对象，因此，我们可以选择将资源（比如内存块）**从临时对象移动到新构造的对象中**，而不是拷贝内存。通过这样的方法，我们就节省了拷贝内存的时间。

26、C++ 智能指针

- C++ 引入了 `unique_ptr<T>`、`shared_ptr<T>` 和 `weak_ptr<T>` 三类智能指针；
- `shared_ptr`：共享资源所有权的指针；`shared_ptr` 允许多个指针指向相同的对象。`shared_ptr` 使用引用计数，每一个 `shared_ptr` 的拷贝都指向相同的内存。每使用他一次，内部的引用计数加 1，每析构一次，内部的引用计数减 1，减为 0 时，自动删除所指向的堆内存；
- `unique_ptr<T>`：独占资源所有权的指针；当我们独占资源的所有权的时候，可以使用 `std::unique_ptr` 对资源进行管理——离开 `unique_ptr` 对象的作用域时，会自动释放资源。这是很基本的 **RAII 思想**；
- `unique_ptr<T>` 的 **拷贝构造和拷贝赋值均被声明为 delete**。因此无法实施拷贝和赋值操作，但可以移动构造和移动赋值；
- `weak_ptr<T>`：共享资源的观察者；`weak_ptr` 是为了配合 `shared_ptr` 而引入的一种智能指针，因为它不具有普通指针的行为，没有重载 `operator*` 和 `->`,它的最大作用在于协助 `shared_ptr` 工作，像旁观者那样观测资源的使用情况；`weak_ptr` 可以从一个 `shared_ptr` 或者另一个 `weak_ptr` 对象构造，获得资源的观测权。但 `weak_ptr` 没有共享资源，它的构造不会引起指针引用计数的增加。

- 实现一个简单的 `shared_ptr`：

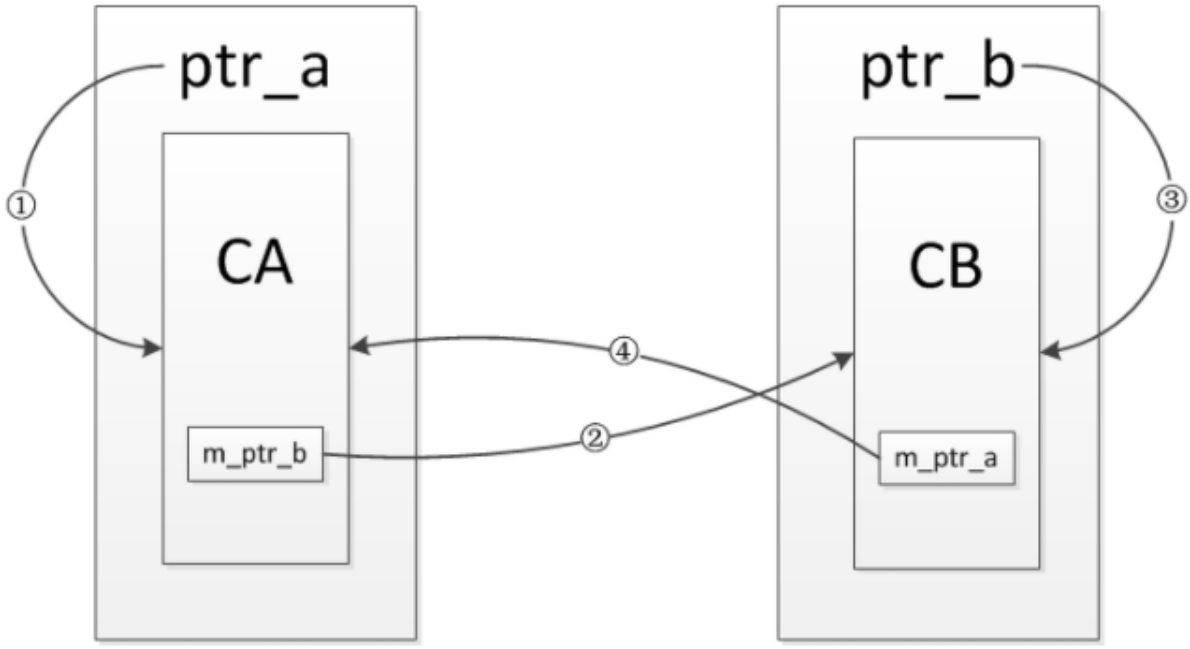
```
template <typename T>
class shared_ptr {
public:
    shared_ptr(T *p):count(new int(1)),_ptr(p) {}
    shared_ptr(shared_ptr<T> &other):count(&(++*other.count)),_ptr(other._ptr)
{}
    shared_ptr<T>& operator=(shared_ptr<T> &rhs) {
        if (this !== &rhs) {
            ++*rhs.count;
            if (this->_ptr && 0 == --*rhs.count) {
                delete count;
                delete _ptr;
            }
            this->_ptr = rhs._ptr;
            this->count = rhs.count;
        }
        return *this;
    }
    ~shared_ptr() {
        if (--*count == 0) {
            delete count;
            delete _ptr;
        }
    }
private:
    int *count;             // 计数器不能放在作为类的成员，应该放在动态内存中
    T* _ptr;
};
```

- 实现一个 `unique_ptr`：

```
template<typename T>
class unique_ptr {
public:
    explicit unique_ptr(T *p):_ptr(p) {}

    unique_ptr(unique<T> &other) = delete;
    unique_ptr& operator=(unique_ptr<T> &rhs) = delete;

    unique_ptr(unique<T> &&other) noexcept :_ptr(other._ptr) {
        other._pter = nullptr;
    }

    unique_ptr& operator=(unique_ptr<T> &&rhs) noexcept {
        /* if (*this != rhs) {
            if (_ptr)
                delete _ptr;
            _ptr = rhs._ptr;
            rhs._ptr = nullptr;
        }
        return *this;*/

        this->swap(*this, ths);
        return *this;
```

```
    }

    T* operator*() const noexcept {return _ptr;}
    T& operator->()const noexcept {return *_ptr;}
    explicit operator bool() const noexcept{return _ptr;}

    void reset(T* q = nullptr) noexcept
    {
        if(q != _ptr){
            if(_ptr)
                delete _ptr;
            _ptr = q;
        }
    }

    T* release() noexcept
    {
        T* res = _ptr;
        _ptr = nullptr;
        return res;
    }
    T* get() const noexcept {return _ptr;}
    void swap(unique_ptr &p) noexcept
    {
        using std::swap;
        swap(_ptr, p._ptr);
    }

    ~unique_ptr(){
        if (_ptr)
            delete _ptr;
    }
private:
    T *_ptr;
};
```

- 当两个 `shared_ptr` **互相引用**，那么它们就永远无法被释放了，即 **循环引用问题**；`weak_ptr` 的出现就是为了解决 `shared_ptr` 的循环引用的问题；



- `weak_ptr` 指向一个由 `shared_ptr` 管理的对象而不影响所指对象的生命周期，也就是，将一个 `weak_ptr` 绑定到一个 `shared_ptr` 不会改变 `shared_ptr` 的引用计数。不论是否有 `weak_ptr` 指向，一旦最后一个指向对象的 `shared_ptr` 被销毁，对象就会被释放。

27、什么是 RAII？如何实现

- RAII 全称是 “Resource Acquisition is Initialization”，直译过来是“**资源获取即初始化**”；
- 即在构造函数中申请分配资源，在析构函数中释放资源。因为 C++ 的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数；
- 每当处理需要 **配对的获取/释放函数调用的资源** 时，都应该将资源封装在一个对象中，实现自动资源释放，比如锁。

28、什么是 lambda 表达式？如何实现？

- 我们可以向一个算法传递任何类别的 **可调用对象**；如果一个对象或者一个表达式，可以对其使用调用运算符，则称它为可调用对象：
  - 函数、函数指针、函数对象（重载了函数调用运算符的类）、**lambda 表达式**
- 一个 lambda 表达式表示一个可调用的代码单元，可以将其理解为一个 **匿名（未命名）的内联函数**；

```
[capture list](parameter list) -> return type {function body}
```

其中，`capture list` 是一个 `lambda` 表达式所在函数中定义的局部变量的列表；`lambda` 必须使用 **尾置返回类型**。可以忽略参数列表和返回类型，但必须永远包含捕获列表和函数体。

- 每当定义一个 `lambda` 表达式以后，编译器会自动生成一个 **匿名类**，并且这个类重载了 `()` 运算符，我们将其称之为 **闭包类型（closure type）**。在运行时，这个 `lambda` 表达式会返回一个匿名的闭包实例；闭包的一个强大之处在于其可以通过传值或引用的方式捕捉其封装作用域内的变量，`lambda` 表达式前面的方括号就是用来定义捕捉模式以及变量的；
- lambda 匿名类与lambda表达式的对应关系：
  - lambda 表达式中的**捕获列表**，对应 lambda 类中的**private成员**；
  - lambda 表达式中的**形参列表**，对应 lambda 类成员函数 **operator() 的形参列表**；
  - lambda 表达式中的**返回类型**，对应 lambda 类成员函数 **operator() 的返回类型**；
  - lambda 表达式中的**函数体**，对应 lambda 类成员函数 **operator() 的函数体**；
- 捕获方式对应的 private 成员类型：

    - **值捕获**：private 成 的类型与捕获变量的类型一致；
    - **引用捕获**：private 成员的类型是捕获变量的引用类型；
- lambda 表达式与函数对象比较：
    - 有的函数对象类只用来定义了一个对象，而且这个对象也只使用了一次，编写这样的函数对象类就有点浪费；
    - 此外，定义函数对象类的地方和使用函数对象的地方可能相隔较远，看到函数对象，想要查看其 operator() 成员函数到底是做什么的也会比较麻烦；
    - 使用 Lambda 表达式可以减少程序中函数对象类的数量

29、模板全特化与模板偏特化

- 模板为什么要特化，因为编译器认为，对于特定的类型，如果你能对某一功能更好的实现，那么就该听你的；
- 模板分为类模板与函数模板，特化分为全特化与偏特化；
- **全特化** 就是限定死模板实现的具体类型；
- **偏特化** 就是如果这个模板有多个类型，那么只限定其中的一部分；偏特化并不是对模板参数的某个或某种组合指定某个参数，是针对任何模板参数更进一步的条件限制所设计出来的一个特化版本；

```
template<typename T>
class C {...};

template<typename T>
class C<T*> {...};              // T 为原生指针便是 "T 为任何型别"的一个更进一步的条件限制
```

- 类模板可以全特化和偏特化；
- **函数模板只能全特化**，这是因为可以通过 **重载函数模板** 实现偏特化功能。

30、什么是踩内存?

- 指访问了不该访问的内存地址，例如：数组越界、使用释放后的内存、随机踩等；

31、malloc 分配原理

- 当申请 **小内存** 时，`malloc` 使用 `sbrk` 分配内存；当申请 **大内存** 时，使用 `mmap` 函数申请内存；但是这 **只是分配了虚拟内存**，还没有映射到物理内存，当访问申请的内存时，才会因为缺页异常，内核分配物理内存；
- 内存堆顶是 `brk 指针`，`brk()` 和 `sbrk()` 函数原型：

```
#include <unistd.h>
int brk( const void *addr );
void* sbrk ( intptr_t incr );
```

这两个函数的作用主要是扩展堆的上界 `brk`。第一个函数的参数为设置的新的 `brk` 上界地址，如果成功返回 0，失败返回 -1。第二个函数的参数为需要申请的内存的大小，然后返回堆新的上界 `brk` 地址。如果 `sbrk` 的参数为 0，则返回的为原来的 `brk` 地址。

- `mmap()` 和 `munmap()` 函数原型：

```
#include <sys/mman.h>
void *mmap(void *addr, size\_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

`mmap` 函数第一种用法是映射磁盘文件到内存中；而 `malloc` 使用的 `mmap` 函数的第二种用法，即匿名映射，匿名映射不映射磁盘文件，而是向映射区申请一块内存。`munmap` 函数是用于释放内存，第一个参数为内存首地址，第二个参数为内存的长度。接下来看下 `mmap` 函数的参数。

- 由于 `brk/sbrk/mmap` 属于系统调用，如果每次申请内存，都调用这三个函数中的一个，那么每次都要产生系统调用开销（即 cpu 从用户态切换到内核态的上下文切换，这里要保存用户态数据，等会还要切换回用户态），这是非常影响性能的；其次，这样申请的内存容易产生碎片，因为堆是从低地址到高地址，如果低地址的内存没有被释放，高地址的内存就不能被回收；
- 因此，`malloc()` 采用内存池的内存管理方式：**先申请一大块内存，然后将内存分成不同大小的内存块，然后用户申请内存时，直接从内存池中选择一块相近的内存块即可。**（类似于 STL 库的空间配置器）。

32、用 malloc() 申请的内存调用 delete() 释放会发生什么?

- `new` 和 `delete` 操作时, 是把对象当成一个复杂类，而执行对应的构造/析构函数，而 `malloc` 和 `free` 则不执行它们；
- 当对象是简单类型时，没有复杂的构造/析构函数，混用两者不会出错；
- 对于复杂类对象，`new` 申请的内存用 `free` 释放则不会调用对象的析构方法。

33、 运算符重载

- `C++` 预定义中的运算符的操作对象只局限于基本的内置数据类型，但是对于我们自定义的类型（类）是没有办法操作的。但是大多时候我们需要对我们定义的类型进行类似的运算，这个时候就需要我们对这么运算符进行重新定义，赋予其新的功能，以满足自身的需求；
- 运算符重载的实质就是函数重载或函数多态。

```
return-type operator <op>(parameter-list) {
    function body;
}
```

- 运算符重载的规则：
  - 为了防止用户对标准类型进行运算符重载，C++ 规定 **重载后的运算符的操作对象必须至少有一个是用户定义的类型**；
  - 不能违法运算符原来的句法规则。如不能将 `%` 重载为一个操作数；
  - 不能修改运算符原先的优先级；
  - 不能创建一个新的运算符；
  - 不能进行重载的运算符：**成员运算符，作用域运算符，条件运算符，** `sizeof运算符`**，** `typeid` **（一个RTTI运算符），** `const_cast`、`dynamic_cast`、`reinterpret_cast`、`static_cast` **强制类型转换运算符**；
  - 大多数运算符可以通过成员函数和非成员函数进行重载但是下面这四种运算符只能通过成函数进行重载：`= 赋值运算符`、`() 函数调用运算符`、`[] 下标运算符`、`-> 指针访问类成员运算符`。

34、inline 函数优缺点？对比 #define 宏呢?

- 在类声明中定义的函数，除了虚函数的其他函数都会**自动隐式地当成内联函数。**
- 编译器对 inline 函数的处理步骤
  - 将 inline 函数体复制到 inline 函数调用点处；
  - 为所用 inline 函数中的局部变量分配内存空间；
  - 将 inline 函数的的输入参数和返回值映射到调用方法的局部变量空间中；
  - 如果 inline 函数有多个返回点，将其转变为 inline 函数代码块末尾的分支（使用 GOTO）。
- 优点

    - 内联函数同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。
    - 内联函数相比宏函数来说，在代码展开时，会做安全检查或自动类型转换（同普通函数），而宏定义则不会。
    - 在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量，宏定义则不能。
    - 内联函数在运行时可调试，而宏定义不可以。
- 缺点
    - 代码膨胀。内联是以代码膨胀（复制）为代价，消除函数调用带来的开销。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。
    - inline 函数无法随着函数库升级而升级。inline函数的改变需要重新编译，不像 non-inline 可以直接链接。
    - 是否内联，程序员不可控。**内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。**
- 虚函数（virtual ）可以是内联函数?
    - 虚函数可以是内联函数，内联是可以修饰虚函数的，但是**当虚函数表现多态性的时候不能内联。**
    - 内联是在编译器建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。
    - `inline virtual` 唯一可以内联的时候是：编译器知道所调用的对象是哪个类（如 `Base::who()`），这只有在编译器具有实际对象而不是对象的指针或引用时才会发生。
- C++ 语言支持函数内联，其目的是为了提高函数的执行效率（速度）；
- 以 `inline` 修饰的函数叫做内联函数，**编译时** C++ 编译器会 **在调用内联函数的地方展开**，没有函数压栈的开销， 内联函数提升程序运行的效率；
- `inline` 是一种以空间换时间的做法，省去调用函数的开销。所以代码很长或者有循环/递归的函数不适宜使用作为内联函数；
- `inline` 对于编译器而言只是一个建议，编译器会自动优化。如果定义为 `inline` 的函数体内有循环/递归等等，编译器优化时会忽略掉内联；
- `inline` 不建议声明和定义分离，分离会导致链接错误。因为 `inline` 被展开，就没有函数地址了，链接就会找不到；
- **宏常量** 一改全改，不需要针对代码中的所有相同常量重新修改；宏常量 **没有类型（不会参与到类型检测中，导致代码安全性降低）**，编译器一旦报错，报错的位置不明确；
- 不方便调试宏(因为预编译阶段进行了替换)；导致代码可读性差，可维护性差，容易误用；没有类型安全的检查 。容易误用体现在：

```
#define MAX(a,b) (a > b? a : b)

此时如果调用 MAX(++a, b)，会 ++ 两次
```

- C++ 中通过 **const 常量** 来替换宏常量， 用 **内联函数** 来替换宏函数。
- **内联函数和 constexpr 函数可以在程序中多次定义。**毕竟，编译器想要展开函数仅有函数声明是不够的，还需要函数的定义。不过，对于某个给定的内联函数或者 constexpr函数来说，它的 **多个定义必须完全一样。**基于这个原因，内联函数和 constexpr 函数 **通常定义在头文件中。**

35、C++ 为什么支持函数重载?

- 重载函数常用来实现 **功能类似而所处理的数据类型不同**的问题;
- C 语言不支持函数重载的原因：在汇编代码中， C 语言区别函数只依赖函数名;
- 在 C++ 编写生成的汇编代码中，通过函数名和形参列表来实现区分;
- 函数重载可以解决 **命名冲突。**

36、虚函数指针和虚表的创建时机?

- `vptr` 跟着对象走，所以对象什么时候创建出来， `vptr` 就什么时候创建出来，也就是 **运行期**；当程序在编译期间，编译器会为构造函数中增加为 `vptr` 赋值的代码(这是编译器的行为)，当程序在运行时，遇到创建对象的代码，执行对象的构造函数，那么这个构造函数里有为这个对象的 `vptr` 赋值的语句。
- 虚函数表创建时机是在 **编译期。**编译期间编译器就为每个类确定好了对应的虚函数表里的内容。所以在程序运行时，编译器会把虚函数表的首地址赋值给虚函数表指针，所以，这个虚函数表指针就有值了。

37、const 关键字用法？与 constexpr 关键字对比?

- `const` 是一个C++语言的限定符，它限定一个变量不允许被改变；只要一个变量前用 `const` 来修饰，就意味着该变量里的数据只能被访问，而不能被修改，也就是意味着 `const` **“只读”（readonly）**；
- 编译器在 **编译过程** 中把用 `const` 修饰变量的地方都替换成对应的值;
- 默认情况下， `const` 对象被设定为 **仅在文件内有效。**当多个文件中出现了同名的 `const` 变量时，其实等于在不同文件中分别定义了独立的变量；如果想在多个文件之间共享 `const` 对象，必须在变量的定义之前添加 `extern` 关键字;
- `const` 的引用能绑定 **临时量**;
- 与宏相比， `const` 修饰符的优点:
  - 预编译指令只是对值进行简单的替换，不能进行类型检查;
  - 可以保护被修饰的东西，防止意外修改，增强程序的健壮性;
  - 编译器通常不为普通 `const` 常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。
- `constexpr` 关键字声明表示函数返回的值或者变量在 **编译期** 可能被求出； `constexpr` 修饰的对象是 **隐式** `const` 的； `constexpr` 修饰的函数是 **隐式内联** 的。
- **常量表达式** 是指 **值不会改变** 并且在 **编译过程就能得到计算结果** 的表达式
- 一个 `constexpr` 变量要满足以下规则:
  - 变量类型必须是 **字面值类型**（算术类型、指针和引用）;
  - 变量必须立即被初始化;
  - 初始化表达式必须是常量表达式（其值在编译器就可确定）。
- 一个 `constexpr` 函数要满足:
  - 不是 `virtual` 函数;
  - 每一个参数都必须是字面值;
  - 只能包含：空语句、静态断言、不声明类和枚举的typedef和alias语句、using语句，最重要的是 **有且只有一个** `return` **语句**；返回的表达式必须是常量表达式。
- `const` 的翻译是 **常量**， `consexpr` 是 **常值**，从他们的名字就能体会其中的不同。常量是指其存在期间不可改变，但其实 `const` 变量的值是可以在程序运行中通过初始化决定的。而常值则是在程序编译期就确定了，在程序运行时，由于 `constexpr` 也是 `const` 的，所以其值也是一直固定的。

```
int a = 10;
const int &b = a;            // 只读 b，并非常量 b
constexptr int &c = a;       // 报错，因为 a 不是常量
```

38、程序在执行 main 函数之前都做了什么工作?

- 内存需要的是数据和指令(机器语言)但是程序是高级语言：
  - 先通过编译链接生成可执行文件(可执行文件在磁盘中存储,且是机器语言)；
  - 可执行文件通过 `mmap` 函数映射到虚拟内存上；
  - 再通过分段分页机制把需要的指令和数据加载到内存；
  - 把 `main 函数` 的入口地址写入到下一行指令寄存器中;
- 在调用 `main 函数` 之前，会先进行初始化栈，堆，打开标准输入，输出，错误流，把`main 函数`参数压栈。还有一些全局变量、对象和静态变量、对象的空间分配和赋初值；
- 在调用 `main函数` 之后，要销毁堆内存，关闭标准输入，输出，错误流。

39、类的成员模板函数可以是虚函数吗?

- 不可以；
- 编译器在编译一个类的时候，需要确定这个类的虚函数表的大小。一般来说，如果一个类有N个虚函数，它的虚函数表的大小就是N，如果按字节算的话那么就是4*N。 如果允许一个成员模板函数为虚函数的话，因为我们可以为该成员模板函数实例化出很多不同的版本，也就是可以实例化出很多不同版本的虚函数，那么编译器为了确定类的虚函数表的大小，就必须要知道我们一共为该成员模板函数实例化了多少个不同版本的虚函数。显然编译器需要查找所有的代码文件，才能够知道到底有几个虚函数，这对于多文件的项目来说，代价是非常高的，所以才规定成员模板函数不能够为虚函数；

40、全局静态变量和局部静态变量的初始化时机?

- 全局变量在调用主函数之前初始化；
- 静态局部变量初始化在第一次调用这个静态局部变量时初始化。

41、为什么需要对象移动?

- 在某些情况下，**对象拷贝后就立即被销毁了**。在这些情况下，移动而非拷贝对象会大幅度提升性能；
- 使用移动而不是拷贝的另一个原因源于 IO 类或 `unique_ptr` 这样的类，这些类都包含不能被共享的资源（如指针和 IO 缓冲），因此，这种类型的对象不能拷贝但可以移动；

42、std::move() 和 std::forward()

- `std::move()` 执行到右值的无条件转换。就其本身而言，它没有move任何东西；
- `std::forward()` 只有在它的参数绑定到一个右值上的时候，它才转换它的参数到一个右值；
- `std::move()` 和 `std::forward()` 只不过就是执行类型转换的两个函数；`std::move()` 没有move任何东西，`std::forward()` 没有转发任何东西。在运行期，它们没有做任何事情。它们没有产生需要执行的代码，一byte都没有。
- `std::forward<T>()` 不仅可以保持左值或者右值不变，同时还可以保持 const、Lreference、Rreference、validate 等属性不变。
- **完美转发**：当我们将一个右值引用传入函数时，他在实参中有了命名，所以继续往下传或者调用其他函数时，根据C++ 标准的定义，这个参数变成了一个左值。那么他永远不会调用接下来函数的右值版本，这可能在一些情况下造成拷贝。完美转发实现了参数在传递过程中保持其值属性的功能，即若是左值，则传递之后仍然是左值，若是右值，则传递之后仍然是右值。
- `std::move` 源码：

```
template<typename T>
inline typename std::remove_reference<T>::type&& move(T&& t) {
    return static_cast<typename std::remove_reference<T>::type&&>(t);
}
```

- `std::forward` 源码：

```
template<typename T>
inline T&& forward(typename std::remove_reference<T>::type&& t) {
    return static_cast<T>(t);
}

template<typename T>
inline T&& forward(typename std::remove_reference<T>::type&& t)
{
    static_assert(!std::is_lvalue_reference<T>::value, "template argument"
                  " substituting _Tp is an lvalue reference type");
    return static_cast<T&&>(t);
}
```

43、为什么模板需要定义在头文件中?

- 当编译器遇到一个模板定义时，并不生成代码，只有当实例化出模板的一个特定版本时，编译器才会生成代码；
- 为了生成一个实例化模板，编译器需要掌握函数模板或类模板成员函数的定义；
- 如果将模板定义放在源文件中，在本编译单元（或者叫做翻译单元）中，编译器没有得到模板函数定义，也就没有办法为其实例化了。没办法实例化不代表会编译错误。由于如果这个函数在其他编译单元实例化了，名字我们是知道的，因此编译器生成了函数调用的代码，期望在链接时能够找到；
- 在编译模板源文件时，虽然有能力实例化，但是本编译单元无人调用该函数，而编译模板定义代码本身是不会生成任何可执行代码的，所以编译模板源文件时没有生成可执行代码。

44、#ifndef #define 的作用

- 防止在一个编译单元内部头文件重复引用。

45、类的定义为什么要放在头文件中

- 类的定义，只是告诉编译器，类的数据格式是如何的，实例化后对象该占多大空间。类的定义也不产生目标代码；
- 类的定义是具有 **内部链接特性**，内部链接指的是该名称对于所在编译单元是局部的，在链接时不会与其他编译单元中同样的名称产生命名冲突，所以类如果要在单个编译单元之外使用它必须被定义在一个头文件中。

46、强制类型转换

- `static_cast<type>(expression)`：
    - 主要用于基本数据类型之间的转换；
    - 还可用于类层次结构中，基类和派生类之间指针或引用的转换，但是
        - 进行上行转换是安全的，即把派生类的指针转换为基类的；
        - 进行下行转换是不安全的，即把基类的指针转换为派生类的
        - 这是因为派生类包含基类信息，所以上行转换（只能调用基类的方法和成员变量），一般是安全的；而基类没有派生类的任何信息，而下行转换后会用到派生类的方法和成员

变量，这些基类都没有，很容易“指鹿为马”，或指向不存在的空间。
    - `static_cast` **没有运行时类型检查**来保证转换的安全性，需要程序员来判断转换是否安全；

- `dynamic_cast<T>(expression)`：
    - `dynamic_cast` 主要用于类层次间的上行转换或下行转换。在进行上行转换时，`dynamic_cast` 和 `static_cast` 的效果是一样的，但在下行转换时，`dynamic_cast` 具有类型检查的功能，比 `static_cast` 更安全；
    - 向下转换的成功与否还与将要转换的类型有关，即要转换的指针指向的对象的实际类型与转换以后的对象类型一定要相同，否则转换失败；
    - `dynamic_cast` 转换如果成功的话返回的是指向类的指针或引用，转换失败的话则会返回 NULL；
    - 在进行下行转换时，从基类 b2 到派生类 d2 时，d2 会改为空指针（0x0），这正是 dynamic_cast 提升安全的功能。这个检查主要来自虚函数表；
- `const_cast<type>(expression)`：
    - 该运算符用来修改 expression 的 `const` 或 `volatile` 属性。这里需要注意：expression 和 type 的类型一样的。
    - 需要特别注意的是 `const_cast` 不是用于去除变量的常量性，而是去除指向常数对象的指针或引用的常量性，其 **去除常量性的对象必须为指针或引用**
- `reinterpret_cast<type>(expression)`：
    - 主要有三种强制转换用途：改变指针或引用的类型、将指针或引用转换为一个足够长度的整形、将整型转换为指针或引用类型；

47、volatile 关键字

- `volatile` 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改。比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以 **提供对特殊地址的稳定访问**；
- 当要求使用 `volatile` 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。例如：

```
volatile int i = 10;
int a = i;
int b = i;
```

- `volatile` 指出 `i` 是随时可能发生变化的，每次使用它的时候必须从 `i` 的地址中读取，因而编译器生成的汇编代码会重新从 `i` 的地址读取数据放在 `b` 中。而优化做法是，由于编译器发现两次从 `i` 读数据的代码之间的代码没有对 `i` 进行过操作，它会自动把上次读的数据放在 `b` 中。而不是重新从 `i` 里面读。这样以来，如果 `i` 是 一个寄存器变量或者表示一个端口数据就容易出错，所以说 `volatile` 可以保证对特殊地址的稳定访问。

- `volatile` 用在如下的几个地方：
    - 中断服务程序中修改的供其它程序检测的变量需要加 `volatile`；
    - 多任务环境下各任务间共享的标志应该加 `volatile`；
    - 存储器映射的硬件寄存器通常也要加 `volatile` 说明，因为每次对它的读写都可能由不同意义；

48、如何定义一个只能在堆上（栈上）生成对象的类?

- 如何定义一个只能在堆/栈上生成对象的类

C++ 模板与泛型编程

1、定义模板

1.1 函数模板

- 一个函数模板就是一个公式，可以用来生成针对特定类型的函数版本；
- **模板参数** 表示在类或函数定义中用到的 **类型** 或 **值**。当使用模板时，显式地或隐式地指定 **模板实参**，将其绑定到模板参数上；
- 当调用一个函数模板时，编译器通常用 **函数实参** 来推断模板实参；
- 模板参数包括 **类型参数** 和 **非类型参数**；
- 一个非类型参数表示一个值而非一个类型；一个非类型参数可以是一个整型，或者一个指向对象或函数类型的指针或左值引用。绑定到非类型整型参数的实惨必须是一个 **常量表达式**。绑定到指针或引用非类型参数的实参必须具有 **静态的生存期**；
- **模板程序应尽量减少对实参类型的要求**；
- 当模板被实例化时，模板的定义，包括类模板的成员的定义，也必须是可见的；因此，函数模板和类模板成员函数的定义通常放在头文件中；
- **实例化** 定义：使用实际的模板实参来代替相应的模板形参来创建一个新的模板“实例”；

1.2 类模板

- 类模板与函数模板的不同之处是：**编译器不能为类模板推断模板参数类型**；使用类模板时，必须在模板名后的尖括号中提供额外信息；
- 类模板的成员函数具有和模板相同的模板参数；
- 默认情况下，对于一个实例化了的类模板，其成员只有在使用时才被初始化；
- 如果一个类模板包含一个非模板友元，则该友元被授权可以访问所有模板实例；如果友元自身是模板，类可以授权给所有友元模板实例，或只授权给特定实例；
- 为了让所有实例成为友元，友元声明中必须使用和类模板本身不同的模板参数。

1.3 模板参数

- 模板参数会隐藏外作用域中声明的相同名字，同时，在模板内不能重用模板参数名；一个模板参数名在一个特定模板参数列表中只能出现一次；
- 默认情况下，C++ 假定通过作用域运算符访问的名字不是类型；当我们希望通知编译器一个名字表示类型时，必须使用 **typename** 关键字

```
template<typename T> typename T::value_type top(const T& c);
```

1.4 成员模板

- 一个类（普通或模板）可以包含本身是模板的成员函数。这种成员被称为 **成员模板**。**成员模板不能是虚函数**；
- 类模板包含成员模板时，类和成员各自有自己的、独立的模板参数；因此，当在类外定义成员模板时，需要同时为类模板和成员模板提供模板参数列表，类模板的参数列表在前；
- 为了实例化一个类模板的成员模板，必须同时提供类和函数模板的参数；我们在哪个对象上调用成员模板，编译器就根据该 **对象的类型来推断类模板参数的实参**；与普通函数模板相同，编译器根据传递给成员模板的 **函数实参来推断它的模板实参**。

1.5 控制实例化

- 模板被使用时才会进行实例化这一特性意味着，**相同的实例可能出现在多个对象文件中**；当两个或多个独立编译的源文件使用了相同的模板，并提供了相同的模板参数时，每个文件中就都会有该模板的一个实例；
- **显示实例化** 可以避免这种开销：

```
extern template declaration;         // 实例化声明
template declaration;                // 实例化定义

extern template class Blob<String>;              // 声明
template int compare(const int&, const int&);    // 定义
```

- 当编译器遇到 `extern` 模板声明时，它不会在本文件中生成实例化代码；将一个实例化声明为 `extern` 表示承诺在程序其他位置有该实例化的一个非 `extern` 声明（定义）；
- 编译器在使用一个模板时自动对其实例化，因此 `extern` 声明必须出现在任何使用此实例化版本的代码之前；
- 对每个实例化声明，程序中某个位置必须有其显示的实例化定义；
- **一个类模板的实例化定义会实例化该模板的所有成员，包括内敛的成员函数**；

2、模板实参推断

- 从 **函数实参** 来确定模板实参的过程被称为 **模板实参推断**；

2.1 类型转换与模板类型参数

- 编译器通常不是对实参进行类型转换，而是生成一个新的模板实例；
- 将实参传递给带模板类型的函数形参时，有限的类型转换：
  - **const 转换**：可以将一个非 const 的引用或指针传递给一个 const 的引用或指针形参；
  - **数组或函数指针转换**：如果函数形参不是引用类型，则可以对数组或函数类型的实参应用正常的指针转换；
- 其它类型转换，如算术转换、派生类向基类的转换以及用户定义的转换都不能应用于函数模板；
- 对于使用相同模板参数类型的函数形参，传递给这些形参的实参必须具有相同的类型，如果推断处的类型不匹配，则调用出错；
- 如果函数参数不是模板参数，则对实参进行正常的类型转换；

2.2 函数模板显式实参

- 模板参数推导是从函数实参确定模板实参，**如果模板参数不在函数参数列表中，例如出现在返回类型中，则编译器不能推断出模板实参的类型**；

```
template<typename T1, typename T2, typename T3>
T1 sum(T2, T3);

// 编译器无法推断 T3，其未出现在函数参数列表中
```

- **显示模板实参** 在尖括号中给出，位于函数名之后，实参列表之前；
- 显示模板实参按从左至右的顺序与对应的模板参数匹配；

2.3 尾置返回类型与类型转换

- 在编译器遇到函数参数列表之前，函数参数都是不存在的；
- 尾置返回出现在参数列表之后，可以使用函数的参数；

```
template<typename It>
auto fcn(It beg, It end) -> decltype(*beg){
    // 处理流程
    return *beg;    // 返回序列中一个元素的引用
}
```

- 如果我们想要获得元素的类型，而非引用，可以使用标准库中的 **类型转换模板**，定义在 `type_traits` 头文件中；
- 组合使用 `remove_reference`、尾置返回以及 `decltype`，可以返回元素值的拷贝，而非引用：

```
template<typename It>
auto fcn(It beg, It end) ->
    typename remove_reference<decltype(*beg)>::type {
    // 处理流程
    return *beg;    // 返回序列中一个元素的拷贝
}
```

2.5 模板实参推断和引用

- 引用折叠
- move
- forward

3、重载与模板

- 函数模板可以被另一个模板或普通非模板函数重载；名字相同的函数必须具有不同数量或类型的参数；
- 函数模板重载对函数匹配规则的影响：
    - 对于一个调用，其候选函数包括所有模板实参推断成功的函数模板实例；
    - 候选的函数模板总是可行的，因为模板实参推断会排除任何不可行的模板；
    - 可行函数（模板与非模板）按类型转换来排序；
    - 如果有多个函数提供同样好的匹配：
        - 只有一个非模板函数，则选择此函数；
        - 多个函数模板，其中一个比其他模板 **更特例化**，
        - 调用有歧义

```
template <typename T> string debug_rep(const T &t) {
    ostringstream ret;
    ret << t;
    return ret.str();
}

template <typename T> string debug_rep(T *p) {
    ostringstream ret;
    ret << "pointer: " << p;
    if (p)
        ret << " " << debug_rep(*p);
```

```
    else
        ret << " null pointer.";
    return ret.str();
}

string s("hi");
cout << debug_rep(s) << endl;

cout << debug_rep(&s) << endl;       // 调用指针版本，第一个版本存在普通指针到 const 指针的转换

const string *sp = &s;
cout << debug_rep(sp) << endl;       // 两个都是精确匹配，此调用被解析为 debug_rep(T*)，即更特例化的版本
                                     // 模板 debug_rep(const T&) 本质上可以用于任何类型，比 debug_rep(T*) 更通用，后者只能用于指针类型
```

编译、调试

1、编译与链接的过程

- 编译与链接可以分解为 4 个步骤：**预处理、编译、汇编和链接。**
- 预处理过程主要处理源码文件以 `#` 开始的预编译指令，生成 `.i` 文件。比如 `#include`、`#define`：

  ```
  // -E 表示只执行到预编译，直接输出预编译结果
  g++ -E helloworld.cpp -o helloworld.i
  ```

  - 将所有的 `#define` 删除，并且展开所有的宏定义；
  - 处理所有条件预编译指令，比如 `#if`、`#ifdef`、`#elif`、`#else`、`#endif`；
  - 处理 `#include` 预编译指令，将被包含的文件插入到预编译指令的位置，这个过程是 **递归进行的。**
  - 过滤所有的注释；
  - 添加行号和文件名标识，以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号；
  - 保留所有的 `#pragma` 编译器指令，因为编译器需要它们。
- 编译过程就是把预处理完的文件进行一系列的 **词法分析、语法分析、语义分析** 以及 **优化** 后产生的汇编代码文件，即 **未链接的目标文件。**

  ```
  // -S 表示只执行到源代码到汇编代码的转换
  g++ -S helloworld.i -o helloworld.s
  ```

  - 编译的过程一般分为 6 步：扫描（词法分析）、语法分析、语义分析、源代码优化、代码生成和目标代码优化；
- 链接的主要内容是 **把各个模块之间相互引用的部分都处理好，把一些指令对其它符号地址的引用加以修正，使得各模块之间能够正确的衔接。**
- 每个目标文件除了拥有自己的数据和二进制代码外，还提供了 3 个表：**未解决符号表、导出符号表、地址重定向表**：
  - 未解决符号表提供了所有在 **该编译单元里引用但是定义并不是在本编译单元的符号以及其出现的地址**；
  - 导出符号表提供了 **本编译单元里具有定义，并且愿意提供给其他单元使用的符号及其地址**；
  - 地址重定向表提供了 **本编译单元所有对自身地址的引用的记录。**

编译器将 `extern` 声明的变量置入未解决符号表，这属于 **外部链接**；将 `static` 声明的全局变量不置入未解决符号表，也不置入导出符号表，因此其他单元无法使用，这属于 **内部链接**；普通变量及函数被置入导出符号表。

- 链接分为 **静态链接** 和 **动态链接**。对函数库的链接是放在 **编译时期完成** 的是静态链接。这些函数库称为静态库，通常文件命名为 `libxxx.a` 的形式。
  - 先将 `.cpp` 文件编译成 `.o` 文件；

```
g++ -c add.cpp
g++ -c sub.cpp
```

  - 由 `.o` 文件创建静态库

```
ar cr libmymath.a sub.o add.o
```

`ar` 命令的 `c` 选项是创建一个库，`r` 选项是在库中插入模块。

  - 在程序中使用静态库：

```
g++ -o main main.cpp -L. -lmymath
```

- 把对一些 **库函数的链接推迟到程序运行时期**，这就是动态链接库。
  - 动态链接库的生成：

```
g++ -fPIC -o add.o -c add.cpp
g++ -fPIC -o sub.o -c sub.cpp
g++ -shared -o libmymath.so add.o sub.o

g++ -fPIC -shared -o libmymath.so add.cpp sub.cpp

-fPIC：表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的，载入时是通过代码复制的方式来满足不同进程的需要，不是真正的共享；
-Lpath：表示在 path 目录中搜索库文件，如 -L. 表示在当前目录；
-Ipath：表示在 path 目录中搜索头文件；
-ltest：查找库
```

- 静态库和动态库对比：
  - 动态链接库有利于进程间资源共享；
  - 链接动态库的程序升级时更简单，而静态库若发生变化，使用该库的程序要重新编译；
  - 动态库的链接载入可以完全由程序员在程序代码中控制；
  - 由于静态库在编译时就将库函数装到程序中了，所以程序执行时，速度更快些。

2、Makefile

教程1

教程2

3、gdb

- `gdb` 是 `gcc` 的调试工具，主要用于调试 C 和 C++ 这两种语言编写的程序。要调试 C/C++ 的程序，首先在编译时，必须要把调试信息加到可执行文件中。使用编译器的 `-g` 参数。如果没有 `-g` ，将看不见程序的函数名、变量名，所显示的全是运行时的内存地址。
- `gdb` 启动方法：
  - gdb program (core)/(PID)；
- `gdb` 命令：
  - `l`：列出函数代码及其行数；
  - `b row`：在代码 `row` 行处设置断点；
  - `r`：运行程序；
  - `n`：单步执行语句；
  - `b func`：在函数 `func` 处设置断点；
  - `p i`：打印变量 `i` 的值；
  - `bt`：查看函数的堆栈；
  - `info break`：查看断点的信息；
  - `finish`：退出函数；
  - `q`：结束调试。

4、 用 gdb 分析 coredump 文件

- `coredump` 文件含有当进程被终止时内存、CPU 寄存器和各种函数调用堆栈信息。
- 产生 `coredump` 文件的原因：
  - 内存访问越界：使用错误的下标，导致数组访问越界；搜索字符串时，依靠字符串结束符控制程序结束，但是字符串没有正常的使用结束符；
  - 多线程程序使用了线程不安全的函数；
  - 多线程读写数据时未加锁保护；
  - 非法指针，包括使用空指针或随意使用指针转换；随意使用指针转换是指一个指向一段内存的指针，除非确定这段内存原先就分配为某种结构或类型，否者不要将它转换为这种结构或类型的指针；
  - 堆栈溢出，不要使用大的局部变量（因为局部变量都分配在栈上）。

5、top 命令

- `top` 命令能够实时显示系统中各个进程的资源占用状况；
- 显示系统中 CPU 最敏感的任务列表，可以按 CPU 使用，内存使用和执行时间对任务进行排序。

6、ps 命令

- Linux 中的 `ps` 命令列出的是当前在运行的进程的快照，即执行 `ps` 命令的那个时刻，因此查看的结果并不动态连续；如果想要对进程时间监控，应该用 `top` 命令；
- Linux 上进程有 5 种状态及 `ps` 标识码：
  - 运行， R；
  - 中断（休眠中，阻塞，在等待某个条件的形成或接收到信号），S；
  - 不可中断，D；
  - 僵死（进程已终止，但进程描述符存在），Z；
  - 停止，T；

7、Linux 程序内存空间布局

- 一个典型的 Linux 下的 C 程序内存空间由如下几部分组成：
  - 代码段：用来存放程序执行代码的一块内存区域；
  - 初始化数据段：用来存放程序中 **已初始化的全局变量** 的一块内存区域，如所有函数外的全局变量；
  - 未初始化数据段：用来存放程序中 **未初始化的全局变量** 的一块内存区域。
  - 堆：用于存放进程运行中被 **动态分配的内存段**；
  - 栈：用来存放程序的 **局部变量**（不包括 `static` 声明的变量，`static` 意味着存放在数据段中）；在函数被调用时，用来传递参数和返回值。
- 堆栈的区别：
  - 申请方式不同：栈由 **系统自动分配**；堆需要程序员 **自己申请，并指明大小。**
  - 申请后系统的响应不同：
    - 只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则报异常；
    - 操作系统由一块记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个大于所申请空间的堆结点，然后将该节点从空闲结点链表中删除，并将该节点的空间分配给程序；其次，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样 `delete` 语句才能正确的释放本内存空间；最后，系统会自动地将多余的那部分内存空间重新放入空闲链表中；
  - 申请大小的限制不同：
    - 栈是向低地址扩展的数据结构，是一块 **连续的内存区域**，即栈顶的地址和栈的最大容量是系统预先规定好的；
    - 堆是向高地址扩展的数据结构，是 **不连续的内存区域**；堆的大小受限于计算机系统中有效的虚拟内存。
  - 申请效率不同：
    - 栈由系统自动分配，速度较快；
    - 堆是由 `new` 分配的内存，一般速度慢，而且容易产生内存碎片
  - 存储内容不同：
    - 函数调用时，第一个进栈的是主函数中的下一条指令的地址，然后是函数的各个参数（大多数 C 编译器中，参数是从右往左入栈的），然后是函数中的局部变量。
    - 一般是在堆的头部用一个字节存放堆的大小，堆中的具体内容由程序员安排。

网络部分

1、TCP 如何保证传输可靠性?

- TCP 协议传输的特点是 **面向字节流、传输可靠、面向连接**；
- 其中， TCP 确保传输可靠性的方式主要有：**校验和、序列号/确认应答、超时重传、连接管理、流量控制和拥塞控制**；
- **校验和**：
  - 将发送的数据段都当作一个 16 位的整数，将这些整数加起来，并且前面的进位不丢弃，补在后面，最后取反；
  - 发送发负责在发送数据前计算校验和。并填充到头部的校验和字段；
  - 接收方收到数据后，对数据以同样的方式进行计算，求出校验和，与发送方的进行对比；
- **确认应答与序列号**：
  - TCP 传输时将每个字节的数据都进行了编号
  - TCP 传输的过程中，每次 接收方收到数据后，都会对传输方进行确认应答，即发送 ACK 报文；

    - 有了序列号能够将接收到的数据根据序列号排序，并且去掉重复序列号的数据。
- **超时重传**：
    - TCP传输时保证能够在任何环境下都有一个高性能的通信，因此这个最大超时时间（也就是等待的时间）是动态计算的；
    - TCP 第一次握手的 SYN 包超时重传最大次数是由 `tcp_syn_retries` 指定，默认值为 5 次；第二次握手的 SYN、ACK 包超时重传最大次数是由 `tcp_synack_retries` 指定，默认值为 5 次；TCP 建立连接后的数据包最大超时重传次数由 `tcp_retries2` 指定，默认值是 15次；
- **连接管理**：三次握手和四次挥手；
- **流量控制**：TCP 根据接收端对数据的处理能力，决定发送端的发送速度；
    - 接收端会在确认应答发送 ACK 报文时，将自己的即时窗口大小填入，并跟随 ACK 报文一起发送过去；
    - 发送方根据 ACK 报文里的窗口大小的值的改变进而改变自己的发送速度；
    - 如果接收到窗口大小的值为 0，那么发送方将停止发送数据。并定期的向接收端发送窗口探测数据段，让接收端把窗口大小告诉发送端。
- **拥塞控制**：是可靠性的保证，同时也是维护了传输的高效性

2、HTTP 请求过程如何?

- 域名解析 —> 发起 TCP 的 3 次握手 —> 建立 TCP 连接后发起 http请求 —> 服务器响应 http 请求 —> 浏览器解析HTML代码，并请求HTML代码中的资源—>关闭TCP连接，浏览器对页面进行渲染呈现给用户；
- HTTP 会发起一次请求request报文，它包括：**请求行（request line）**、**请求头部（header）**、**空行** 和 **请求数据** 这四个部分：
    - 请求行由请求方法字段、URL 字段和 HTTP 协议版本字段3个字段组成，它们用空格分隔。例如：

      `GET /index.html HTTP/1.1`
    - HTTP 协议的请求方法有 GET、POST、HEAD、PUT、DELETE、OPTIONS、TRACE、CONNECT：
        - GET: 完整请求一个资源；
        - POST：提交表单；
    - 请求头部由 **关键字/值对** 组成，每行一对，关键字和值用英文冒号":"分隔。请求头部通知服务器有关于客户端请求的信息，典型的请求头有：
        - Accept：客户端可识别的内容类型列表；
        - Accept-Encoding：声明浏览器支持的编码类型；
        - Cache-Control：指定了请求和响应遵循的缓存机制；
        - Connection：决定当前的事务完成后，是否会关闭网络连接。如果该值是"keep-alive"，网络连接就是持久的，不会关闭，使得对同一个服务器的请求可以继续在该连接上完成；
        - Host：请求的主机名，允许多个域名同处一个IP地址，即虚拟主机；
        - Referer：告诉服务器从哪个页面链接过来的，服务器藉此可以获得一些信息用于处理；
        - Upgrade-Insecure-Requests：让浏览器自动升级请求从 http 到 https，用于大量包含 http 资源的 http 网页直接升级到 https 而不会报错；
        - User-Agent：产生请求的浏览器类型；

请求头	说明
Host	接受请求的服务器地址，可以是IP:端口号，也可以是域名
User-Agent	发送请求的应用程序名称
Connection	指定与连接相关的属性，如Connection:Keep-Alive
Accept-Charset	通知服务端可以发送的编码格式
Accept-Encoding	通知服务端可以发送的数据压缩格式
Accept-Language	通知服务端可以发送的语言

    - 最后一个请求头之后是一个 **空行**，发送回车符和换行符，**通知服务器以下不再有请求头**；
    - **请求数据** 不在 GET 方法中使用，而是在 POST 方法中使用。POST 方法适用于需要客户填写表单的场合。
- HTTP响应也由四个部分组成，分别是：**状态行、响应头部、空行、响应正文**：

响应头	说明
Server	服务器应用程序软件的名称和版本
Content-Type	响应正文的类型（是图片还是二进制字符串）
Content-Length	响应正文长度
Content-Charset	响应正文使用的编码
Content-Encoding	响应正文使用的数据压缩格式
Content-Language	响应正文使用的语言

- `get` 和 `post` 的区别：
    - `get` 和 `post` **本质上就是 TCP 链接，并无差别。**但是由于 HTTP 的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同。
    - `post` 不会作为 url 的一部分，不会被缓存、保存在服务器日志、以及浏览器浏览记录中;
    - `post` 发送的数据更大（HTTP 协议本身对 URL 长度并没有做任何规定。实际的限制是由客户端/浏览器以及服务器端决定的）；
    - `post` 能发送更多的数据类型（`get` 只能发送ASCII字符）；

    - `post` 比 `get` 慢；
    - `post` 用于修改和写入数据，`get` 一般用于搜索排序和筛选之类的操作（淘宝，支付宝的搜索查询都是get提交），目的是资源的获取，读取数据。
- HTTP 协议中的安全性和幂等性：
    - **安全性**：仅指该方法的多次调用不会产生副作用，即不会修改服务器上的资源状态；
    - **幂等性**：多次执行相同的操作，结果都是相同的。

方法名	安全性	幂等性	请求方法的作用
get	√	√	请求指定的页面信息，并返回实体主体
head	√	√	只请求页面的首部
options	√	√	允许客户端查看服务器的性能
delete	×	√	请求服务器删除指定的数据
put	×	√	从客户端向服务器传送的数据取代指定的文档的内容
post	×	×	请求服务器接受所指定的文档作为对所标识的URI的新的从属实体

3、HTTP 长连接原理

- HTTP协议的长连接本质上就是 **TCP的长连接**；
- **保活机制**：
    - 通过保活机制，我们可以保证通讯双方的连接不被释放掉；
    - 如果客户端或者服务器发生了错误或者宕机，那么就可以依靠这种保活机制探测出网络通信出现了问题，进而可以释放掉这种错误连接；
- 保活机制的工作原理是：通过在服务器端设置一个保活定时器，当定时器开始工作后就定时的向网络通信的另一端发出保活探测的TCP报文，如果接收到了ACK报文，那么就证明对方存活，可以继续保有连接；否则就证明网络存在故障；
- Linux相关的TCP保活参数：
    - `tcp_keepalive_time`，单位：秒，表示发送探测报文之前的连接空闲时间，默认是7200s；
    - `tcp_keepalive_intvl`，单位：秒，表示两次探测报文之间的间隔时间，默认是 75s；
    - `tcp_keepalive_probes`，表示探测的次数，默认是9。
- 长连接与短连接的比较：
    - 短连接优点：不占服务器的内存，服务器能处理的连接数量会比较多；
    - 短连接缺点：在有实际的资源要进行数据通信的时候才建立连接，那么在客户端发送完数据释放连接之后当服务器有向客户端发送数据时就不能做到发送消息的实时性；频繁地建立连接、释放连接会耗费大量的CPU和网络带宽资源。
    - 长连接优点：通信双方因为在保活机制的保证下可以保证数据收发的实时性；
    - 长连接缺点：服务器需要一直保存和客户端的这条链接，因为是有状态的，那么在大量并发连接请求过来时，系统资源可能就不够了。
- 什么时候需要长连接：
    - 服务器需要主动发送资源给客户端时；
    - 客户端和服务器通信很频繁时；
    - 客户端宕机或者掉线时需要服务器做一些处理时。

4、URI 和 URL 的区别

- URI = Uniform Resource Identifier 统一资源标志符
- URL = Uniform Resource Locator 统一资源定位符
- URI 属于 URL 更高层次的抽象，一种字符串文本标准

5、为什么 TCP 在传输层分段，UDP 在网络 IP 层分片?

- TCP 在建立连接的时候，会协商双方的 MSS值，通常这个 MSS 会控制在 MTU 以内：最大 IP 包大小减去 IP 和 TCP 协议头的大小；
- 假设有一份数据，较大，在 TCP 层不分段，如果这份数据在发送的过程中出现丢包现象，TCP会发生重传，那么重传的就是这一大份数据；如果TCP把这份数据，分段为N个小于等于 MSS 长度的数据包，到了 IP 层后加上 IP 头和 TCP 头，还是小于 MTU，那么 IP 层也不会再进行分包。此时在传输路上发生了丢包，那么TCP重传的时候也只是重传那一小部分的 MSS 段。效率会比 TCP 不分段时更高；
- 而对于 UDP，它并没有协商的能力，所以它只能直接把用户发送的数据，传给网络层（IP层），由网络层来进行分片；那么如果网络发生了波动，丢失了某个 IP 包分片， 对于 UDP 而言， 它没有反馈丢失了哪个分片给发送方的能力，这就意味着数据全都丢失了，如果需要重传，就得再次完整的传递所有数据。

6、为什么在传输层以MSS大小对 TCP 报文分段，就能避免重传所有的片?

- 如果一个大的 TCP 报文是被 MTU 分片，那么 **只有「第一个分片」才具有 TCP 头部**，后面的分片则没有TCP 头部，接收方 IP 层只有重组了这些分片，才会认为是一个 TCP 报文，那么丢失了其中一个分片，接收方 IP 层就不会把 TCP 报文丢给 TCP 层，那么就会等待对方超时重传这一整个 TCP 报文。
- 如果一个大的 TCP 报文被 MSS 分片，那么 **所有「分片都具有 TCP 头部」**，因为每个 MSS 分片的是具有 TCP 头部的TCP报文，那么其中一个 MSS 分片丢失，就只需要重传这一个分片就可以。

7、TCP 第一次握手时会被丢弃的三种条件：

- 如果半连接队列满了，并且没有开启 `tcp_syncookies`，则会丢弃；
- 若全连接队列满了，且没有重传 `SYN+ACK` 包的连接请求多于 1 个，则会丢弃；
- 如果没有开启 `tcp_syncookies`，并且 `tcp_max_syn_backlog` 减去当前半连接队列长度小于 $(tcp\_max\_syn\_backlog >> 2)$，则会丢弃；

系统部分

1、CPU 调度算法中的抢占式和非抢占式?

- 非抢占式：先来先服务、短作业优先、最高响应比优先；
- 抢占式：最短剩余时间优先、时间片轮转法、多级反馈队列

2、cache 存在的原因，多个 cpu 之间的 cache 如何保持一致的?

- CPU 的频率太快了，快到主存跟不上，这样在处理器时钟周期内，CPU 常常需要等待主存，浪费资源。所以cache 的出现，是为了 **缓解 CPU 和内存之间速度的不匹配问题**；
- 既然 cache 不能包含 CPU 所需要的所有数据，那么 cache 的存在真的有意义吗？——有意义，**局部性原理**:
  - **时间局部性**：如果某个数据被访问，那么在不久的将来它很可能被再次访问
  - **空间局部性**：如果某个数据被访问，那么与它相邻的数据很快也可能被访问
- 缓存一致性：在多核CPU中，内存中的数据会在多个核心中存在数据副本，某一个核心发生修改操作，就产生了数据不一致的问题。而一致性协议正是用于保证多个CPU cache之间缓存共享数据的一致。
- **写传播**：某个 CPU 核心里的 Cache 数据更新时，必须要传播到其他核心的 Cache；
  - **总线嗅探**

- **事务串行化**：某个 CPU 核心里对数据的操作顺序，必须在其他核心看起来顺序是一样的；
- **MESI 协议**：*Modified*、*Exclusize*、*Shared*、*Invalidated*

当前状态	事件	行为
已失效 I(Invalid)	Local Read	如果其它核心的 Cache 没有这份数据，本地核心的 Cache 从内存中读取数据，Cache Line 状态变成 E； 如果其它核心的 Cache 有这份数据，且状态为 M，则将数据更新到内存，本地核心的 Cache 再从内存中取数据，这 2 个 Cache 的 Cache Line 状态都变成 S； 如果其它核心的 Cache 有这份数据，且状态为 S 或者 E，本地核心的 Cache 从内存中取数据，这些 Cache 的 Cache Line 状态都变成 S；
	Local Write	从内存中读取数据，缓存到 Cache，再在 Cache 中更新数据，状态变成 M； 如果其它核心的 Cache 有这份数据，且状态为 M，则要先将其他核心的 Cache 里的数据写回到内存； 如果其它核心的 Cache 有这份数据，则其它 Cache 的 Cache Line 状态变成 I；
	Remote Read	既然是 Invalid，其他核心的操作与它无关
	Remote Write	既然是 Invalid，其他核心的操作与它无关
独占 E(Exclusive)	Local Read	从 Cache 中取数据，状态不变
	Local Write	修改 Cache 中的数据，状态变成M
	Remote Read	数据和其它核心共用，状态变成了 S
	Remote Write	数据被修改，本地核心的 Cache Line 中的数据不能再使用，状态变成 I
共享 S(Shared)	Local Read	从 Cache 中取数据，状态不变
	Local Write	修改 Cache 中的数据，状态变成 M， 其它核心共享的 Cache Line 状态变成 I
	Remote Read	状态不变
	Remote Write	数据被修改，本地核心的 Cache Line 中的数据不能再使用，状态变成 I
已修改	Local Read	从 Cache 中取数据，状态不变
	Local Write	修改 Cache 中的数据，状态不变

M(Modified)	Remote Read	Cache Line 的数据被写到内存中，使其它核能使用到最新的数据，状态变成 S
	Remote Write	Cache Line 的数据被写到内存中，使其它核能使用到最新的数据，由于其它核会修改这行数据，状态变成 I

3、select、poll、epoll对比

- `select` ==> 时间复杂度 $O(n)$
  - 它仅仅知道有I/O事件发生了，却并不知道是哪几个流（可能有一个，多个，甚至全部），只能 **无差别轮询所有流**，找出能读出数据，或者写入数据的流，对他们进行操作;
- `poll` ==> 时间复杂度 $O(n)$
  - `poll` 本质上和 `select` 没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个 `fd` 对应的设备状态， 但是它 **没有最大连接数的限制**，原因是它是 **基于链表来存储** 的
- `epoll` ==> 时间复杂度 $O(1)$
  - `epoll` 可以理解为 `event poll`，不同于忙轮询和无差别轮询，`epoll` 会把哪个流发生了怎样的 I/O 事件通知我们。所以我们说 `epoll` 实际上是事件驱动（每个事件关联上 `fd`）的
- `epoll` 有 **EPOLLLT** 和 **EPOLLET** 两种触发模式，LT 是默认的模式，ET 是"高速"模式
  - LT 模式下，只要这个 `fd` 还有数据可读，每次 `epoll_wait` 都会返回它的事件，提醒用户程序去操作;
  - ET（边缘触发）模式中，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无论 `fd` 中是否还有数据可读。所以在 ET 模式下，read一个fd的时候一定要把它的buffer读光，也就是说一直读到read的返回值小于请求值;
  - 如果采用 LT 模式的话，系统中一旦有大量你不需要读写的就绪文件描述符，它们每次调 `epoll_wait` 都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率;而 ET 模式下，系统不会充斥大量你不关心的就绪文件描述符;
- `select` 的缺点:
  - 每次调用 `select`，都需要把 `fd` 集合从用户态拷贝到内核态，这个开销在 `fd` 很多时会很大;
  - 同时每次调用 `select` 都需要在内核遍历传递进来的所有 `fd`，这个开销在 `fd` 很多时也很大;
  - `select` 支持的文件描述符数量太小了，默认是1024。
- `epoll` 的优点:
  - **没有最大并发连接的限制**，能打开的FD的上限远大于1024;
  - 效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用 `callback` 函数;**即 epoll 最大的优点就在于它只管你"活跃"的连接，而跟连接总数无关，因此在实际的网络环境中，epoll 的效率就会远远高于 select 和 poll 。**
  - 内存拷贝，利用 `mmap()` 文件映射内存，加速与内核空间的消息传递;即 `epoll` 使用 `mmap` 减少复制开销。

4、进程都有哪些资源

- 一个进程拥有独立的地址空间（代码段、数据段），打开的文件描述符、自身的信号处理器、进程控制块，进程通信的一些资源

5、线程与进程的区别：

- 进程是资源（包括内存、打开的文件等）分配的单位，线程是 CPU 调度的单位；
- 同一进程的线程共享本进程的地址空间，而进程之间则是独立的地址空间；
- 同一进程内的线程共享本进程的资源如内存、I/O、cpu等，但是进程之间的资源是独立的；
- 一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都会死掉。所以多进程要比多线程健壮；
- 进程切换时，消耗的资源大，效率高。所以涉及到频繁的切换时，使用线程要好于进程。同样如果要求同时进行并且又要共享某些变量的并发操作，只能用线程不能用进程；
- 每个独立的进程有一个程序运行的入口、顺序执行序列和程序入口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

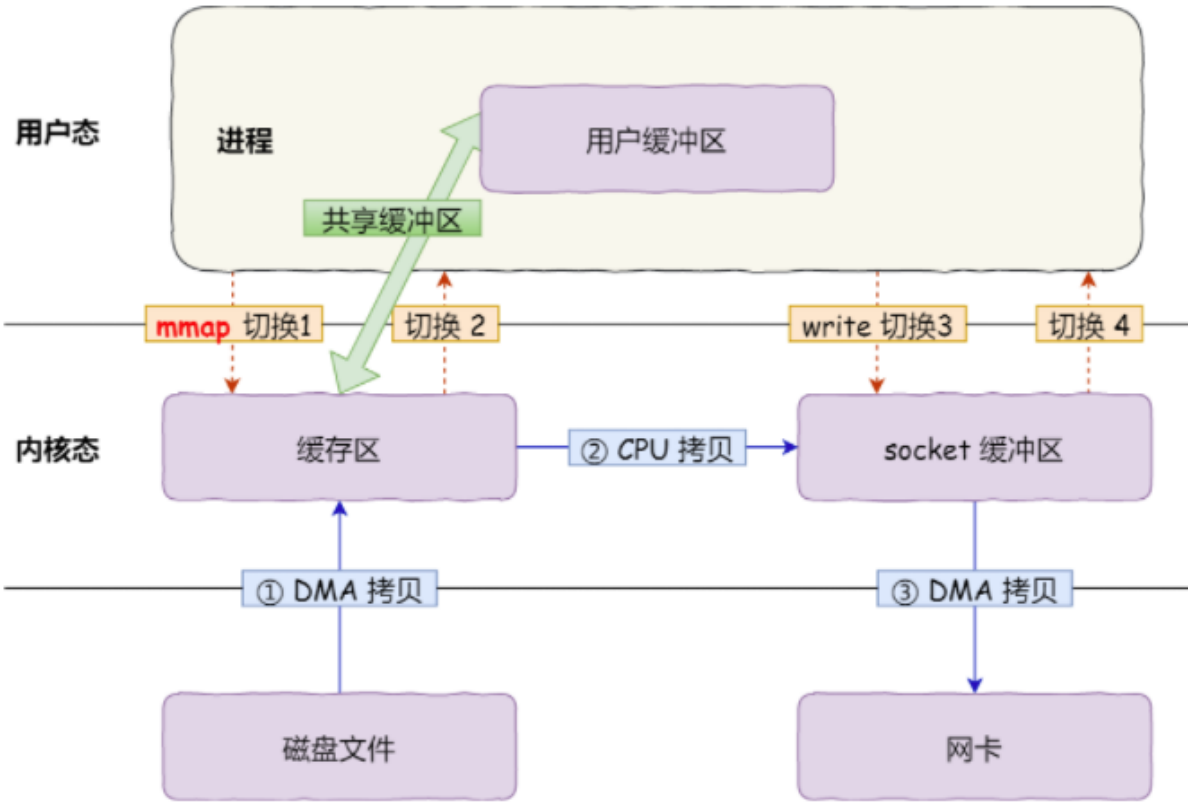
6、多进程与多线程的区别以及选择

维度	多进程	多线程	总结
数据共享、同步	数据是分开的，共享复杂，需要用IPC；同步简单	多线程共享进程数据，共享简单；同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度快	线程占优
编程调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会相互影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适应于多核、多机分布；如果一台机器不够，扩展到多台机器比较简单	适应于多核分布	线程占优

- 多进程优点：
  - 编程相对容易，通常不需要考虑锁和同步资源的问题；
  - 更强的容错性：一个进程崩溃了不会影响其他进程；
- 多线程优点：
  - 创建速度快，方便高效的数据共享；多线程间可以共享同一虚拟地址空间；多进程间的数据共享就需要用到共享内存、信号量等 `IPC 进程间通信` 技术
  - 较轻的上下文切换开销：不用切换地址空间，不用更改寄存器，不用刷新 TLB 和页表；
- 两者选择：
  - 需要频繁创建销毁的优先用线程（进程的创建和销毁开销过大），如 web 服务器；
  - 需要进行大量计算的优先使用线程（CPU频繁切换）；所谓大量计算，当然就是要耗费很多CPU，切换频繁了，这种情况下线程是最合适的，如图像处理；
  - 强相关的处理用线程，弱相关的处理用进程；
  - 可能要扩展到 **多机分布的用进程**，**多核分布的用线程**；

7、零拷贝是啥？如何实现？

- 零拷贝用来减少文件传输时的上下文切换和数据拷贝；没有 **在内存层面去拷贝数据**，也就是说全程没有通过 CPU 来搬运数据，所有的数据都是通过 DMA 来进行传输的；
- 零拷贝技术实现的方式通常有 2 种：
  - `mmap() + write`
  - `sendfile`
- `mmap() + write`：
  - `read()` 系统调用的过程中会把内核缓冲区的数据拷贝到用户的缓冲区里，为了减少这一步开销，我们可以用 `mmap()` 替换 `read()` 系统调用函数；
  - `mmap()` 系统调用函数会 **直接把内核缓冲区里的数据「映射」到用户空间**，这样，操作系统内核与用户空间就不需要再进行任何的数据拷贝操作。
  - 应用进程调用了 `mmap()` 后，DMA 会把磁盘的数据拷贝到内核的缓冲区里。接着，应用进程跟操作系统内核「共享」这个缓冲区；
  - 应用进程再调用 `write()`，操作系统直接将内核缓冲区的数据拷贝到 `socket` 缓冲区中，这一切都发生在内核态，由 CPU 来搬运数据；
  - 最后，把内核的 `socket` 缓冲区里的数据，拷贝到网卡的缓冲区里，这个过程是由 DMA 搬运的；
  - 这还不是最理想的零拷贝，因为仍然需要通过 CPU 把内核缓冲区的数据拷贝到 `socket` 缓冲区里，而且仍然需要 4 次上下文切换，因为系统调用还是 2 次。

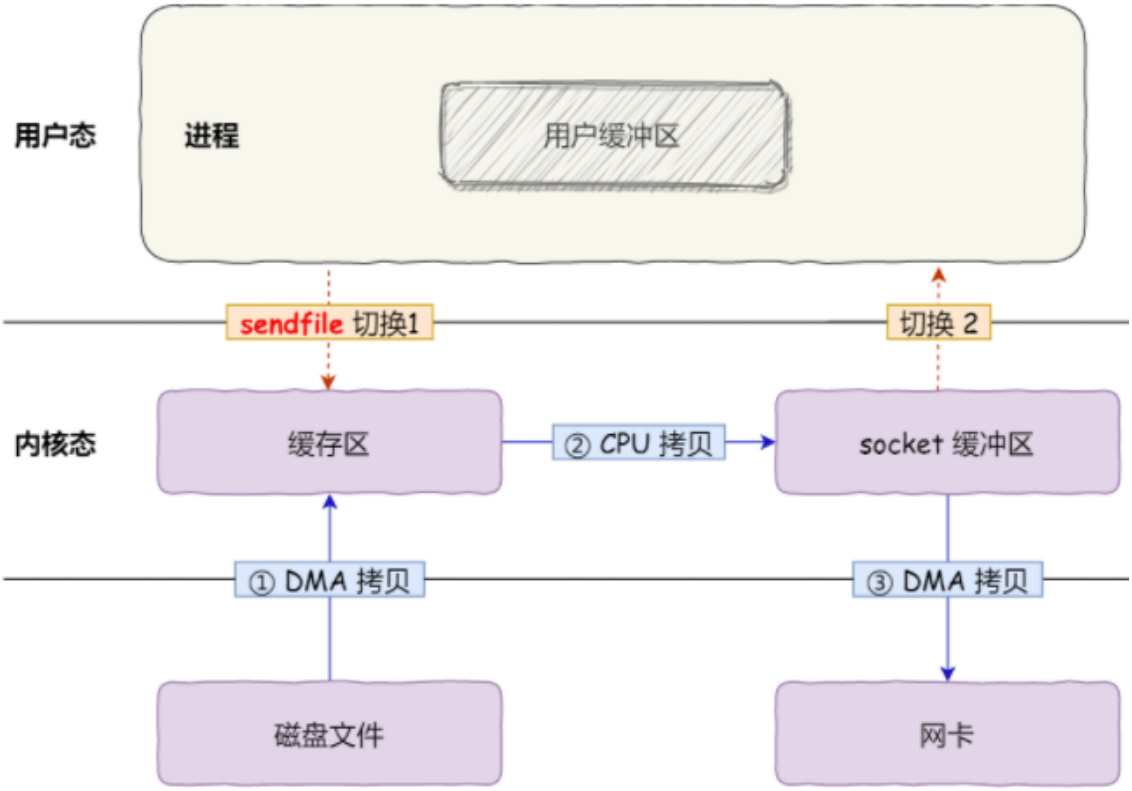


- `sendfile()`：
  - 函数原型：

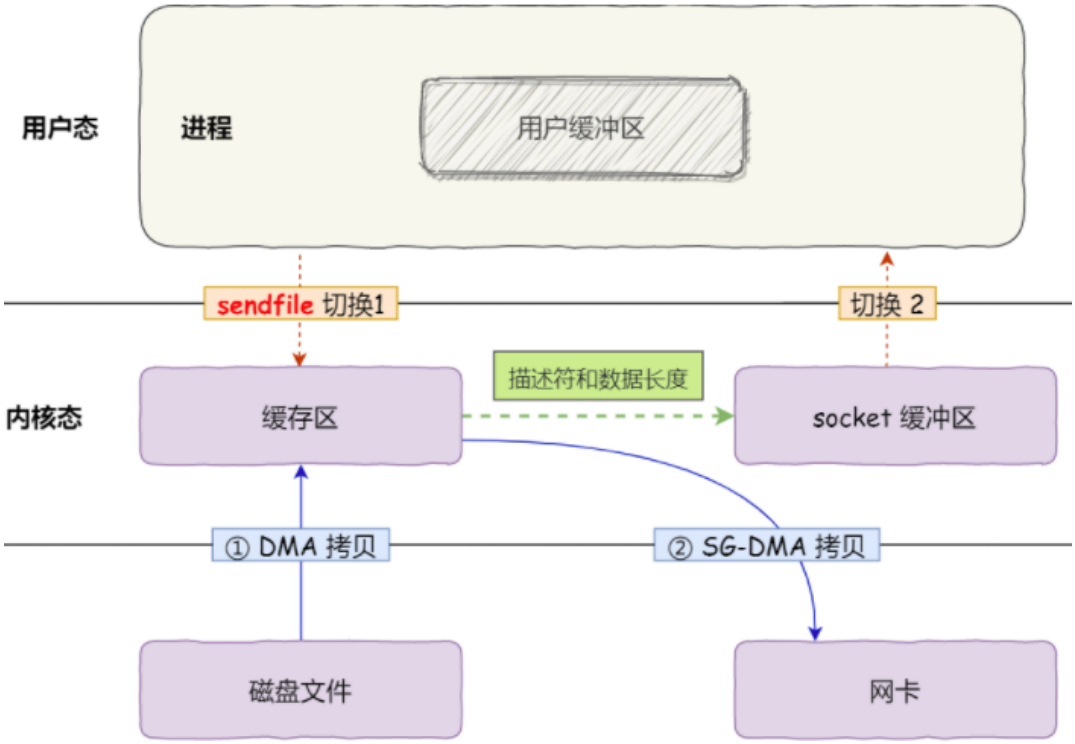
```
ssize_t sendfile(int out_fd. int in_fd, off_t *offset, size_t count);
```

它的前两个参数分别是目的端和源端的文件描述符，后面两个参数是源端的偏移量和复制数据的长度，返回值是实际复制数据的长度；

- 它可以替代前面的 `read()` 和 `write()` 这两个系统调用，这样就可以减少一次系统调用，也就减少了 2 次上下文切换的开销；
- 该系统调用，可以直接把内核缓冲区里的数据拷贝到 `socket` 缓冲区里，不再拷贝到用户态，这样就只有 2 次上下文切换，和 3 次数据拷贝；



- 但是这还不是真正的零拷贝技术，如果网卡支持 **SG-DMA（The Scatter-Gather Direct Memory Access）** 技术（和普通的 DMA 有所不同），我们可以进一步减少通过 CPU 把内核缓冲区里的数据拷贝到 socket 缓冲区的过程：
  - 第一步，通过 DMA 将磁盘上的数据拷贝到内核缓冲区里；
  - 第二步，缓冲区描述符和数据长度传到 socket 缓冲区，这样网卡的 SG-DMA 控制器就可以直接将内核缓存中的数据拷贝到网卡的缓冲区里，此过程不需要将数据从操作系统内核缓冲区拷贝到 socket 缓冲区中，这样就减少了一次数据拷贝；



mmap 内存映射原理：

- 进程在 **用户空间** 调用库函数 `mmap()` 启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域；`mmap` 函数原型为：

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, pff+t
pffset);
```

- 调用 **内核空间** 的系统调用函数 `mmap`（不同于用户空间函数），实现文件物理地址和进程虚拟地址的一一映射关系；此时，这片虚拟地址并没有任何数据关联到主存；
- 进程发起对这片映射空间的访问，引发缺页异常，实现文件内存到物理内存的拷贝。
    - 前两个阶段仅在于创建虚拟区间并完成地址映射，但是并没有将任何文件数据拷贝至主存。真正的文件读取是当进程发起读或写操作时；
    - 进程的读或写操作访问虚拟地址空间这一段映射地址，通过查询页表，发现这一段地址并不在物理页面上，引发缺页异常；
    - 调页过程 **先在交换缓存空间（swap cache）中寻找** 需要访问的内存页，如果没有则调用 `nopage 函数` 把所缺的页从磁盘装入到主存中；
    - 之后进程即可对这片主存进行读或者写的操作，如果写操作改变了其内容，一定时间后系统会自动回写脏页面到对应磁盘地址，也即完成了写入到文件的过程。（修改过的脏页面并不会立即更新回文件中，而是有一段时间的延迟，可以调用 `msync()` 来强制同步, 这样所写的内容就能立即保存到文件里了）
- 常规文件操作需要从磁盘到页缓存再到用户主存的 **两次数据拷贝** 。而 `mmap` 操控文件，只需要从磁盘到用户主存的 **一次数据拷贝** 过程。`mmap` 的关键点是实现了用户空间和内核空间的数据直接交互而省去了空间不同数据不通的繁琐过程。因此 `mmap` 效率更高。
- `mmap` 优点：
    - 对文件的读取操作 **跨过了页缓存（位于内核空间，不能被用户进程直接寻址），减少了数据的拷贝次数**，用内存读写取代I/O读写，提高了文件读取效率；
    - 实现了用户空间和内核空间的高效交互方式。两空间的各自修改操作可以直接反映在映射的区域内，从而被对方空间及时捕捉；
    - 提供进程间共享内存及相互通信的方式；不管是父子进程还是无亲缘关系的进程，都可以将自身用户空间映射到同一个文件或匿名映射到同一片区域。从而通过各自对映射区域的改动，达到进程间通信和进程间共享的目的；
    - 可用于实现高效的大规模数据传输。内存空间不足，是制约大数据操作的一个方面，解决方案往往是借助硬盘空间协助操作，补充内存的不足。但是进一步会造成大量的文件I/O操作，极大影响效率。这个问题可以通过 `mmap` 映射很好的解决。换句话说，但凡是需要用磁盘空间代替内存的时候，`mmap` 都可以发挥其功效。

8、程序转化为进程的步骤？

- 内核将程序读入内存，为程序分配内存空间；
- 内核为该进程分配进程标识符（ PID ）和其他所需资源；
- 内核为进程保存 PID 及相应的状态信息，把进程放到运行队列中等待执行，程序转化为进程后就可以被操作系统的调度程序调度执行了。
- 同一个程序文件可以被加载多次成为不同的进程 。 因此，**进程与进程标识符之间是一对一的关系，而与程序文件之间是多对一的关系。**

9、调用 fork() 会发生什么?

- 对于父进程， `fork()` 函数返回新创建的子进程的 ID ;
- 对于子进程， `fork()` 函数返回 0;
- 如果创建出错，则 `fork()` 函数返回 -1 ;
- `fork()` 函数会创建一个新的进程，并从内核中为此进程分配一个新的可用的进程标识符(PID) ，之后，为这个新进程分配进程空 间，并将父进程的进程空间中的内容复制到子进程的进程空间中，**包括父进程的数据段和堆栈段，并且和父进程共享代码段** ;
- 由于创建的新进程和父进程在系统看来是 **地位平等** 的两个进程， 运行机会也是一样的，故不能够对其执行先后顺序进行假设，先执行哪一个进程取决于系统的调度算法 。

10、孤儿进程、僵尸进程和守护进程是什么?

- **孤儿进程**，是指一个父进程退出后，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程 ； 孤儿进程将被 **init 进程（进程号为 1 ）** 所收养，并由 init 进程对它们完成状态收集工作;
- **僵尸进程**，是指一个进程使用 `fork` 创建子进程，如果子进程退出，而父进程并没有调用 `wait` 或 `waitpid` 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中，这种进程称为僵尸进程;
- **守护进程**，是脱离于终端并且在后台运行的进程 。 守护进程脱离于终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的终端信息所打断 ； 创建一个守护进程的步骤:
    - 创建子进程，父进程退出 ;
    - 在子进程中创建新会话；使用的是系统函数 `setsid`,
    - 改变当前目录为根目录；通常的做法是让 `"/"` 作为守护进程的当前工作目录;
    - 重设文件权限掩码，设置为 0;
    - 关闭文件描述符。

10、什么是内存泄漏，如何检查?

- 内存泄露指的是，在程序中动态申请的内存，在使用完后既没有释放，又无法被程序的其他部分访问。
- 软件:

11、什么是线程安全的?

- 当多个线程访问某段代码时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在调用代码中不需要任何额外的同步或者协同，这个类都能表现出正确的行为，那么就称这个类是线程安全的。
- C++ 标准库里的大多数类都不是线程安全的，包 括std:: string、std::vector、std::map等，因为这些class通常需要在外部加锁才能供多个线程同时访问。

Linux 系统

1、硬链接与软链接

- 链接简单说实际上是一种文件共享的方式。Linux 中常用它来解决一些库版本的问题，通常也会将一些目录层次较深的文件链接到一个更易访问的目录中;
- **硬连接** 指通过 **索引节点** 来进行连接。在Linux的文件系统中，保存在磁盘分区中的文件不管是什么类型都给它分配一个编号，称为索引节点号(Inode Index)。硬连接的作用是允许一个文件拥有多个有效路径名，这样用户就可以建立硬连接到重要文件，以防止"误删"的功能。只有当最后一个连接

被删除后，文件的数据块及目录的连接才会被释放。也就是说，文件真正删除的条件是与之相关的所有硬连接文件均被删除；

- 另外一种连接称之为 **符号连接（Symbolic Link）**，也叫 **软连接。**软链接文件有类似于Windows的快捷方式。它实际上是一个特殊的文件。在符号连接中，文件实际上是一个文本文件，其中包含的有另一文件的位置信息。

项目相关

Dijsktra 算法源码

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

const int INF = 1e9; // int范围约为 (-2.15e9, 2.15e9)

/*Dijkstra算法解决的是单源最短路径问题，即给定图G(V,E)和起点s(起点又称为源点),边的权值为非负，
求从起点s到达其它顶点的最短距离，并将最短距离存储在矩阵d中*/
void Dijkstra(int n, int s, vector<vector<int>> G, vector<bool> &vis,
vector<int> &d, vector<int> &pre)
{
    /*
     *   n:         顶点个数
     *   s:         源点
     *   G:         图的邻接矩阵
     * vis:         标记顶点是否已被访问
     *   d:         存储源点s到达其它顶点的最短距离
     * pre:         最短路径中v的前驱结点
     */

    // 初始化
    fill(vis.begin(), vis.end(), false);
    fill(d.begin(), d.end(), INF);
    d[s] = 0;
    for (int i = 0; i < n; ++i)
    {
        pre[i] = i;
    }

    // n次循环,确定d[n]数组
    for (int i = 0; i < n; ++i)
    {
        // 找到距离s最近的点u,和最短距离d[u]
        int u = -1;
        int MIN = INF;
        for (int j = 0; j < n; ++j)
        {
            if (!vis[j] && d[j] < MIN)
            {
                u = j;
                MIN = d[j];
            }
        }

        // 找不到小于INF的d[u],说明剩下的顶点与起点s不连通
```

```
        if (u == -1)
        {
            return;
        }

        vis[u] = true;
        for (int v = 0; v < n; ++v)
        {
            // 遍历所有顶点，如果v未被访问 && 可以达到v && 以u为中介点使d[v]更小
            if (!vis[v] && G[u][v] != INF && d[u] + G[u][v] < d[v])
            {
                d[v] = d[u] + G[u][v];    // 更新d[v]
                pre[v] = u;               // 记录v的前驱顶点为u（新添加）
            }
        }
    }
}

// 输出从起点s到顶点v的最短路径
void DFSPrint(int s, int v, vector<int> pre)
{
    if (v == s)
    {
        cout << s << " ";
        return;
    }
    DFSPrint(s, pre[v], pre);
    cout << v << " ";
}


int main()
{
    int n = 6;
    /*邻接矩阵*/
    vector<vector<int>> G = {{  0,  4,INF,INF,  1,  2},
                             {  4,  0,  6,INF,INF,  3},
                             {INF,  6,  0,  6,INF,  5},
                             {INF,INF,  6,  0,  4,  5},
                             {  1,INF,INF,  4,  0,  3},
                             {  2,  3,  5,  5,  3,  0}};
    vector<bool> vis(n);
    vector<int> d(n);
    vector<int> pre(n);

    Dijkstra(n, 0, G, vis, d, pre);

    for (size_t i = 0; i < d.size(); ++i)
    {
        cout << "the shortest path " << i << " is: " << d[i] << endl;
    }
    cout << endl;

    // v = 2: 0->5->2  cost = 2 + 5 = 7
    // v = 3: 0->4->3  cost = 1 + 4 = 5
    int v = 2;
    DFSPrint(0, v, pre);
    cout << endl << "cost = " << d[v] << endl;
```

```
    return 0;
}
```

A* 算法源码

- 头文件:

```
#pragma once
/*
//A*算法对象类
*/
#include <vector>
#include <list>

const int kCost1=10; //直移一格消耗
const int kCost2=14; //斜移一格消耗

struct Point
{
    int x,y; //点坐标，这里为了方便按照C++的数组来计算，x代表横排，y代表竖列
    int F,G,H; //F=G+H
    Point *parent; //parent的坐标，这里没有用指针，从而简化代码
    Point(int _x,int _y):x(_x),y(_y),F(0),G(0),H(0),parent(NULL)  //变量初始化
    {
    }
};


class Astar
{
public:
    void InitAstar(std::vector<std::vector<int>> &_maze);
    std::list<Point *> GetPath(Point &startPoint,Point &endPoint,bool
isIgnoreCorner);

private:
    Point *findPath(Point &startPoint,Point &endPoint,bool isIgnoreCorner);
    std::vector<Point *> getSurroundPoints(const Point *point,bool
isIgnoreCorner) const;
    bool isCanreach(const Point *point,const Point *target,bool isIgnoreCorner)
const; //判断某点是否可以用于下一步判断
    Point *isInList(const std::list<Point *> &list,const Point *point) const; //
判断开启/关闭列表中是否包含某点
    Point *getLeastFpoint(); //从开启列表中返回F值最小的节点
    //计算FGH值
    int calcG(Point *temp_start,Point *point);
    int calcH(Point *point,Point *end);
    int calcF(Point *point);
private:
    std::vector<std::vector<int>> maze;
    std::list<Point *> openList;  //开启列表
    std::list<Point *> closeList; //关闭列表
};
```

- 源文件:

```cpp
#include <math.h>
#include "Astar.h"

void Astar::InitAstar(std::vector<std::vector<int>> &_maze)
{
    maze=_maze;
}

int Astar::calcG(Point *temp_start,Point *point)
{
    int extraG=(abs(point->x-temp_start->x)+abs(point->y-temp_start->y))==1?
kCost1:kCost2;
    int parentG=point->parent==NULL?0:point->parent->G; //如果是初始节点，则其父节点
是空
    return parentG+extraG;
}

int Astar::calcH(Point *point,Point *end)
{
    //用简单的欧几里得距离计算H，这个H的计算是关键，还有很多算法，没深入研究^_^
    return sqrt((double)(end->x-point->x)*(double)(end->x-point->x)+(double)
(end->y-point->y)*(double)(end->y-point->y))*kCost1;
}

int Astar::calcF(Point *point)
{
    return point->G+point->H;
}

Point *Astar::getLeastFpoint()
{
    if(!openList.empty())
    {
        auto resPoint=openList.front();
        for(auto &point:openList)
            if(point->F<resPoint->F)
                resPoint=point;
        return resPoint;
    }
    return NULL;
}

Point *Astar::findPath(Point &startPoint,Point &endPoint,bool isIgnoreCorner)
{
    openList.push_back(new Point(startPoint.x,startPoint.y)); //置入起点,拷贝开辟一
个节点，内外隔离
    while(!openList.empty())
    {
        auto curPoint=getLeastFpoint(); //找到F值最小的点
        openList.remove(curPoint); //从开启列表中删除
        closeList.push_back(curPoint); //放到关闭列表
        //1,找到当前周围八个格中可以通过的格子
        auto surroundPoints=getSurroundPoints(curPoint,isIgnoreCorner);
        for(auto &target:surroundPoints)
        {
            //2,对某一个格子，如果它不在开启列表中，加入到开启列表，设置当前格为其父节点，计算
F G H
```

```
            if(!isInList(openList,target))
            {
                target->parent=curPoint;

                target->G=calcG(curPoint,target);
                target->H=calcH(target,&endPoint);
                target->F=calcF(target);

                openList.push_back(target);
            }
            //3，对某一个格子，它在开启列表中，计算G值，如果比原来的大，就什么都不做，否则设置它的父节点为当前点,并更新G和F
            else
            {
                int tempG=calcG(curPoint,target);
                if(tempG<target->G)
                {
                    target->parent=curPoint;

                    target->G=tempG;
                    target->F=calcF(target);
                }
            }
            Point *resPoint=isInList(openList,&endPoint);
            if(resPoint)
                return resPoint; //返回列表里的节点指针，不要用原来传入的endpoint指针，因为发生了深拷贝
        }
    }

    return NULL;
}

std::list<Point *> Astar::GetPath(Point &startPoint,Point &endPoint,bool isIgnoreCorner)
{
    Point *result=findPath(startPoint,endPoint,isIgnoreCorner);
    std::list<Point *> path;
    //返回路径，如果没找到路径，返回空链表
    while(result)
    {
        path.push_front(result);
        result=result->parent;
    }
    return path;
}

Point *Astar::isInList(const std::list<Point *> &list,const Point *point) const
{
    //判断某个节点是否在列表中，这里不能比较指针，因为每次加入列表是新开辟的节点，只能比较坐标
    for(auto p:list)
        if(p->x==point->x&&p->y==point->y)
            return p;
    return NULL;
}

bool Astar::isCanreach(const Point *point,const Point *target,bool isIgnoreCorner) const
```

```
{
    if(target->x<0||target->x>maze.size()-1
        ||target->y<0&&target->y>maze[0].size()-1
        ||maze[target->x][target->y]==1
        ||target->x==point->x&&target->y==point->y
        ||isInList(closeList,target)) //如果点与当前节点重合、超出地图、是障碍物、或者在关闭列表中，返回false
        return false;
    else
    {
        if(abs(point->x-target->x)+abs(point->y-target->y)==1) //非斜角可以
            return true;
        else
        {
            //斜对角要判断是否绊住
            if(maze[point->x][target->y]==0&&maze[target->x][point->y]==0)
                return true;
            else
                return isIgnoreCorner;
        }
    }
}

std::vector<Point *> Astar::getSurroundPoints(const Point *point,bool isIgnoreCorner) const
{
    std::vector<Point *> surroundPoints;

    for(int x=point->x-1;x<=point->x+1;x++)
        for(int y=point->y-1;y<=point->y+1;y++)
            if(isCanreach(point,new Point(x,y),isIgnoreCorner))
                surroundPoints.push_back(new Point(x,y));

    return surroundPoints;
}



int main()
{
    //初始化地图，用二维矩阵代表地图，1表示障碍物，0表示可通
    vector<vector<int>> maze={
        {1,1,1,1,1,1,1,1,1,1,1,1},
        {1,0,0,1,1,0,1,0,0,0,0,1},
        {1,0,0,1,1,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,1,0,0,1,1,1},
        {1,1,1,0,0,0,0,0,1,1,0,1},
        {1,1,0,1,0,0,0,0,0,0,0,1},
        {1,0,1,0,0,0,0,1,0,0,0,1},
        {1,1,1,1,1,1,1,1,1,1,1,1}
    };
    Astar astar;
    astar.InitAstar(maze);

    //设置起始和结束点
    Point start(1,1);
    Point end(6,10);
    //A*算法找寻路径
```

```
    list<Point *> path=astar.GetPath(start,end,false);
    //打印
    for(auto &p:path)
        cout<<'('<<p->x<<','<<p->y<<')'<<endl;

    system("pause");
    return 0;
}
```