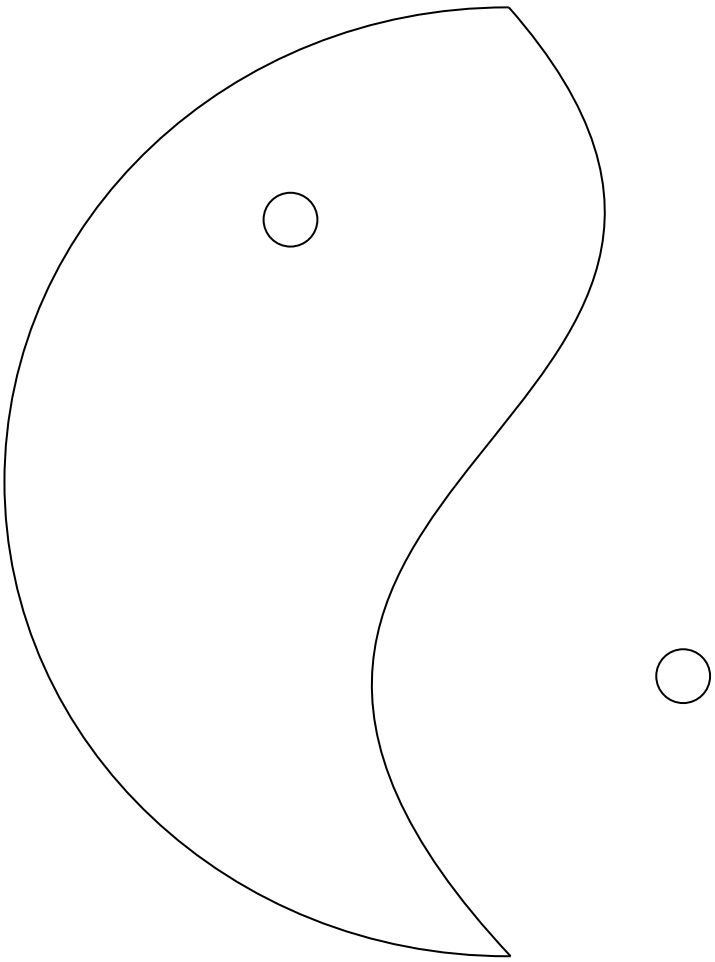
Obtuse

Wenqingqian



wenqq3@outlook.com

目录

第一章 C++

1.1 基础

1.1.1 C 和 C++ 区别

- C 语言是 C++ 的子集, C++ 可以很好兼容 C 语言. 但是 C++ 又有很多新特性, 如引用、智能指针、auto 变量等
- C++ 是面对对象的编程语言; C 语言是面对过程的编程语言
- C 语言有一些不安全的语言特性, 如指针使用的潜在危险、强制转换的不确定性、内存泄露等. 而 C++ 对此增加了不少新特性来改善安全性, 如 const 常量、引用、cast 转换、智能指针、try—catch 等等
- C++ 可复用性高, C++ 引入了模板的概念, 后面在此基础上, 实现了方便开发的标准模板库 STL. C++ 的 STL 库相对于 C 语言的函数库更灵活、更通用

1.1.2 C++ 从代码到可执行二进制文件的过程

1. 预编译: 这个过程主要的处理操作如下
   (a) 将所有的 #define 删除, 并且展开所有的宏定义
   (b) 处理所有的条件预编译指令, 如 #if、#ifdef
   (c) 处理 #include 预编译指令, 将被包含的文件插入到该预编译指令的位置
   (d) 过滤所有的注释
   (e) 添加行号和文件名标识
2. 编译: 这个过程主要的处理操作如下
   (a) 词法分析: 将源代码的字符序列分割成一系列的记号
   (b) 语法分析: 对记号进行语法分析, 产生语法树
   (c) 语义分析: 判断表达式是否有意义
   (d) 代码优化
   (e) 目标代码生成: 生成汇编代码
   (f) 目标代码优化
3. 汇编: 这个过程主要是将汇编代码转变成机器可以执行的指令
4. 将不同的源文件产生的目标文件进行链接, 从而形成一个可以执行的程序
   - 链接分为静态链接和动态链接
   - 静态链接, 是在链接的时候就已经把要调用的函数或者过程链接到了生成的可执行文件中, 就算你在去把静态库删除也不会影响可执行程序的执行; 生成的静态链接库, Windows 下以.lib 为后缀, Linux 下以.a 为后缀

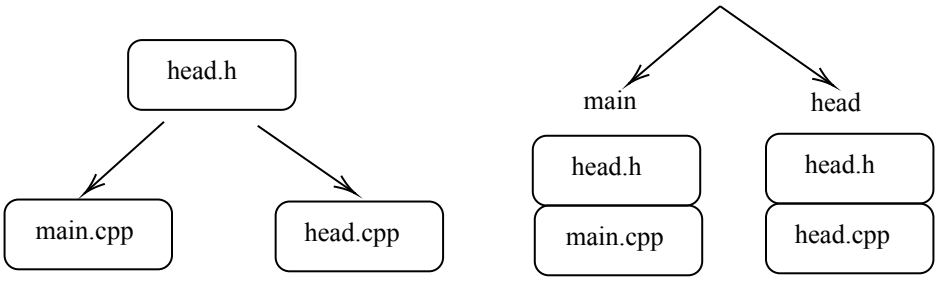
- 而动态链接，是在链接的时候没有把调用的函数代码链接进去，而是在执行的过程中，再去找要链接的函数，生成的可执行文件中没有函数代码，只包含函数的重定位信息，所以当你删除动态库时，可执行程序就不能运行生成的动态链接库，Windows 下以.dll 为后缀，Linux 下以.so 为后缀

1.1.3 static 关键字

1. 全局静态变量和局部静态变量：初始化的静态变量会在数据段分配内存，未初始化的静态变量会在 BSS 段分配内存. 直到程序结束，静态变量始终会维持前值. 只不过全局静态变量和局部静态变量的作用域不一样
2. 静态函数：静态函数只能在本源文件 (该翻译单元) 中使用
3. 类中的静态成员变量：静态数据成员，隐藏在类作用域中的全局变量 (可以通过类名 (Class::) 或类对象访问). 类中的 static 静态数据成员拥有一块单独的存储区，而不管创建了多少个该类的对象. 所有这些对象的静态数据成员都共享这一块静态存储空间
4. 类中的静态成员函数：静态成员函数也是类的一部分，而不是对象的一部分
   只能访问静态数据成员：当调用一个对象的非静态成员函数时，系统会把该对象的起始地址赋给成员函数的 this 指针. 而静态成员函数不属于任何一个对象，因此 C++ 规定静态成员函数没有 this 指针. 既然它没有指向某一对象，也就无法对一个对象中的非静态成员进行访问

1.1.4 extern 关键字与链接性

在 C++ 中，翻译单元由实现文件及直接或间接包含的所有标头组成

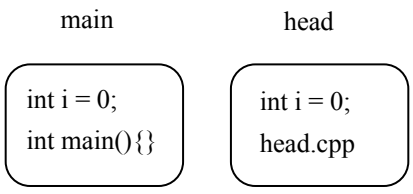


每个翻译单元由编译器单独编译，最终将得到的.o 文件链接得到可执行文件. 如上图所示程序结构，在链接时会出现如下问题：

假如在 head.h 文件中定义一个外部链接性变量.

```
#ifndef HEAD_H
#define HEAD_H
int i = 0;
#endif
```

那么在链接时就会出错，原因是如上两个翻译单元相当于



在两个文件中都定义了具有外部链接性的变量 i.

在一个作用域内，变量能且只能被定义一次，但是可以被多次声明 (one define rule)

定义

```
// 不能在局部作用域内使用
extern int v = 0;
int v;
```

声明

```
void func();
extern void func();
extern int v;
```

如果需要在多个文件, 指个多翻译单元中使用同一变量, 就必须把定义与声明分离. 变量的定义只能出现在一个文件中, 而在其它用到该变量的文件中对其声明.

所以往往头文件中只放一些声明而在另外的文件中进行定义.

常见的内部链接类型

1. const 全局变量
2. static 全局变量
3. static 函数
4. inline[1] 函数/变量[2]
5. 类内定义的成员函数
6. 类内定义的数据成员 (即非静态数据成员)

常见的外部链接类型

1. 全局变量与函数 (非 const、static、inline)
2. 类外定义的数据成员 (即静态数据成员在类外的定义)
3. 类外定义的成员函数
4. extern const T v = INIT;

其它可视为无链接

所有文件中的外部链接性变量会传递给链接器得到一张导出符号表. 记录本编译单元定义, 并且可提供给其它单元使用的符号及在本单元对应的地址.

所有文件中的声明会传递给链接器得到一张未解决符号表, 记录本编译单元有声明但不在本单元定义的符号及其对应的地址, 显然在导出符号表中不能存在相同的符号.

C11 标准关于 static 和 extern 的内容:

1. 若在 extern 声明标识符之前的可见范围内存在对该标识符的声明, 则该标识符的链接与先前声明相同. 若无先前声明, 或声明无链接, 则该标识符具有外部链接
2. 如果在同一翻译单元内, 同一标识符同时出现内外部链接, 则行为未定义

上述原文:

For an identifier declared with the storage-class specifier extern in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

1.1.5 指针与引用

1.1.5.1 指针与引用的区别

1. 引用就是常量指针.

[1] inline 现在表示在链接时遇到不同编译单元出现了相同签名的函数时只保留一份, 不在是内联的意思.

[2] inline variable is C++17 extension

2. 引用必须初始化, 且不能改变指向.

3. 无法对引用进行取地址, 或者说只能取到引用指向的对象的地址.

4. 引用的解引用操作由编译器自动进行.

1.1.5.2 void* 指针

指针一般有三个含义

1. 指明数据的位置, 体现在指针的值.

2. 表示数据的大小, 例如 int 指针表示四个字节为一组的数据, 体现在指针的步长、自身的加减法计算.

3. 表示数据如何被解释, 例如 float 和 int 都是四字节, 但解释结果完全不同, 体现在指针解引用的结果.

void* 指针为第一种, 只指明了数据的位置. 以 void* 的视角来看内存空间也仅仅是内存空间, 没办法访问内存空间中所存的对象.

对于 void* 类型指针与其它类型指针的转换, C++ 有如下两个规定.

1. 不允许从 void* 类型到其它类型的隐式转换, 允许从其它类型到 void* 类型的隐式转换, 常见于函数传参.

2. 字面量 0 可以隐式转换成任意类型的空指针常量.

1.1.5.3 nullptr

在 C 中, 常用 NULL 来表示空指针

```
#define NULL ((void*)0)
```

然而在 C++ 中不允许从 void* 隐式转换成其它类型的指针, 所以定义了一种新的指针类型, 指针空值类型 std::nullptr_t , nullptr 是它的一个实例, 当对指针进行初始化时:

```
int *p = nullptr;
int *p = 0;
```

两者之间没有区别.

但在重载情况下:

```
void func(int n);
void func(int *p);
```

nullptr 会匹配第二个, 而 0 实际上两个都能匹配, 在本实验环境中, 会匹配第一项且没有二义性问题.

若再定义一个

```
void func(char *p);
```

则 nullptr 会出现二义性错误

至于空指针具体指向哪个地址由编译器管理, 可能是内存中的 0 号地址, 即指针数据全是 0, 也有可能不是.

1.1.5.4 野指针

wild pointer 也叫悬挂指针, dangling pointer

有三种情况会造成野指针:

1. 指针未初始化.

2. 指针指向的对象生命周期结束, 如将一个函数体内的局部变量地址传递给外部指针.

3. 指针释放.

当指针被 delete 之后, 指针值变为无效, 继续对该指针进行操作会出现 heap use after free 错误.

事实上, 当一个指针被 delete 之后, 指针指向的堆地址空间被释放, 随时可以被分配给其它对象, 但在不同的机器上会有不同的结果.

几种特殊情况, 指 delete 之后继续使用指针不报错的情况:

1. 指针的数据, 即它指向的地址发生改变, 对它进行解引用会得到 0.

2. 地址不改变, 解引用会得到 0.

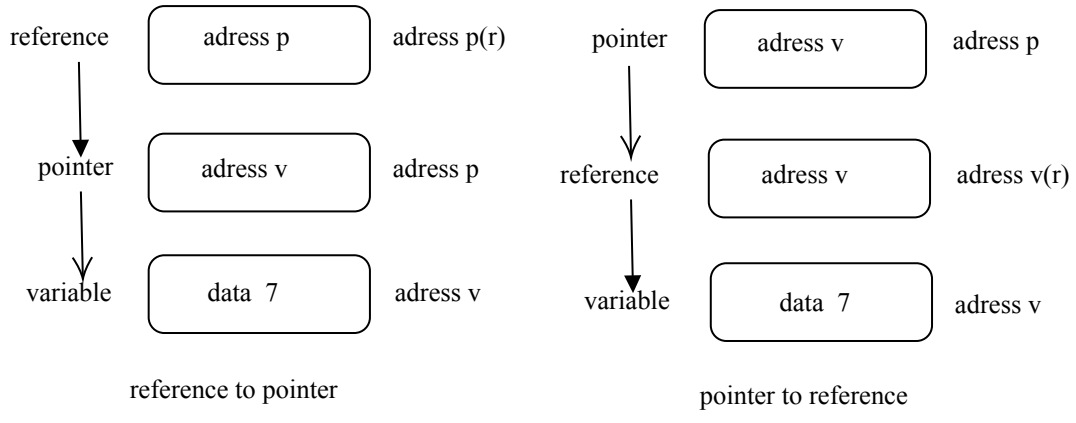
3. 地址不改变, 该内存地址的值也未被擦除, 解引用得到跟 delete 之前一样的结果.

准确来讲, delete 的实际意义是将指针指向的内存空间释放, 以便能分配给其它对象, 但指针本身是否依旧指向那块内存, 那块内存的数据是否被擦除, 将由编译器或者操作系统决定.TODO: 具体决定方式存疑.

给野指针赋 nullptr 初值避免麻烦.

1.1.5.5 指向引用的指针, 指向指针的引用

对引用地址的获取行为会得到引用指向的对象的地址



测试程序见CPP_BASIC_POINTER_1

1.1.6 malloc 和 new 区别

1.2 面向对象

1.2.1 多态的实现原理和应用场景

1.3 STL

1.3.1 线程安全的实现及标准容器库的线程安全性

1.3.2 sort 算法是怎么实现的

1.4 C++11

第二章 Operating System

2.1 Linux

2.1.1 Linux 常见命令

第三章 Computer Networking

3.1 TCP/IP

3.1.1 TCP/IP 协议中的三次握手和四次挥手

第四章 Socket

4.1 IO 多路复用

第五章 Data Structure

第六章 Database