C++内存分配的方式（重要）

（1） 从静态存储区域/全局区分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在
。例如全局变量，static变量。

（2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存
储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3） 从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自
己负责在何时用free或delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

```
4、文字（只读）常量区 –常量字符串就是放在这里的。 程序结束后由系统释放
5、代码段–存放函数体的二进制代码，直接的操作数也是存储在这个位置的。如int a=4;。
```

malloc和new的区别（重要）

1. malloc与free是C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。
2. new 不止是分配内存，而且会调用类的构造函数，同理delete会调用类的析构函数，而malloc则只分配内存，不会进行初始化类成员的工作，同样free也不会调用析构函数
3. **new 和 malloc效率上**
   (1) new可以认为是malloc加构造函数的执行。new能通过placement new自动调用对象的构造函数，malloc不会

   (2) new出来的指针是直接带类型信息的，而malloc返回的都是void指针，需要进行强制类型转换

   (3) new失败时会调用new_handler处理函数,malloc不会,失败时返回NULL

new的执行过程（重要）

1. 通过operator new申请内存，这里的new实际上是像加减乘除一样的操作符，因此也是可以重载的。

   delete也有delete operator和operator delete之分，后者也是可以重载的。并且，如果重载了operator new，就应该也相应的重载operator delete，这是良好的编程习惯。

2. 使用placement new调用构造函数（简单类型忽略此步）

   ```
   //placement new的用法
    char s[sizeof(A)];
    A* p = (A*)s;
    new(p) A(3); //p->A::A(3);
    p->Say();
   ```

   如果是像上面那样在栈上使用了placement new，则必须手工调用析构函数，这也是显式调用析构函数的唯一情况：

```
p->~A();
```

3. 成功返回内存指针，失败调用new_hander抛出bad_alloc异常 或者exit()或类似的函数，使程序结束
   更加深入前往->https://www.cnblogs.com/fnlingnzb-learner/p/8515183.html
4. 示例代码更方便理解

```
try{
    Complex* pc = new Complex(1,2); //new一个对象
    1.void* mem=operator new(sizeof(Complex));//申请内存 内部会调用malloc
    2.pc = static_cast<Complex*>(mem);//类型转换
    3.pc->Complex::Complex(1,2);//构造函数
}catch(std::bad_alloc){

}
```

内存对齐（重要）

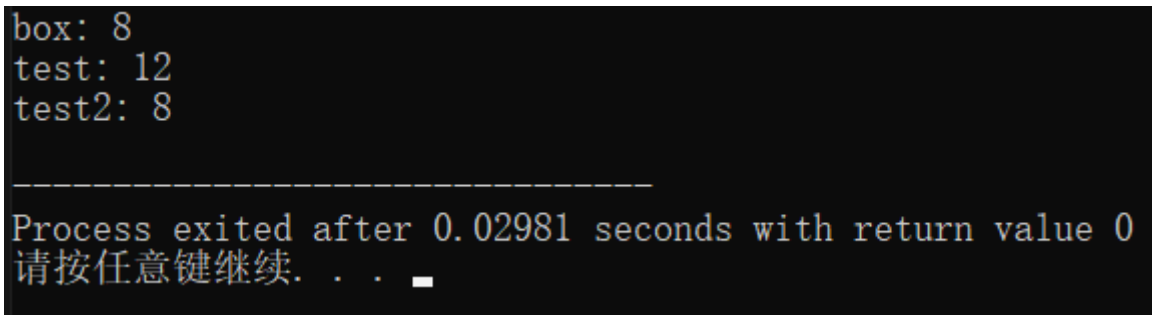
深入探索c++对象模型 p104

C++空类的内存大小为1字节，为了保证其对象拥有彼此独立的内存地址。非空类的大小与类中非静态成员变量和**虚函数表的多少**有关。而值得注意的是，类中非静态成员变量的大小与编译器内存对齐的设置有关。

成员变量在类中的内存存储并不一定是连续的。它是按照编译器的设置，按照**内存块来存储的**，这个内存块大小的取值，就是内存对齐。

```
class Box
{
private:
    int w;//4byte
    int h;//4byte
};
class test {
private :

    char c='1';//1byte
    int i;//4byte
    short s=2;//2byte
};
class test2 {
private:
    int i;//4byte
    char c = '1';//1byte
    short s = 2;//2byte
};
```



可以看到。类test和test2的成员变量完全一样，只是定义顺序不一样，却造成了2个类占用内存大小不一样。而这就是编译器内存对齐的缘故。

规则

伪指令#pragma pack (n)，编译器将按照n个字节对齐；

1、第一个数据成员放在offset为0的地方，以后每个数据成员的对齐按照**#pragma pack**指定的数值和这个数据成员自身长度中，比较小的那个进行。

2、在数据成员完成各自对齐之后，**类(结构或联合)本身也要进行对齐**，对齐将按照#pragma pack指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。

很明显#pragma pack(n)作为一个预编译指令用来设置多少个字节对齐的。值得注意的是，n的缺省数值是按照编译器自身设置，一般为8，合法的数值分别是1、2、4、8、16。

即编译器只会按照1、2、4、8、16的方式分割内存。若n为其他值，是无效的。

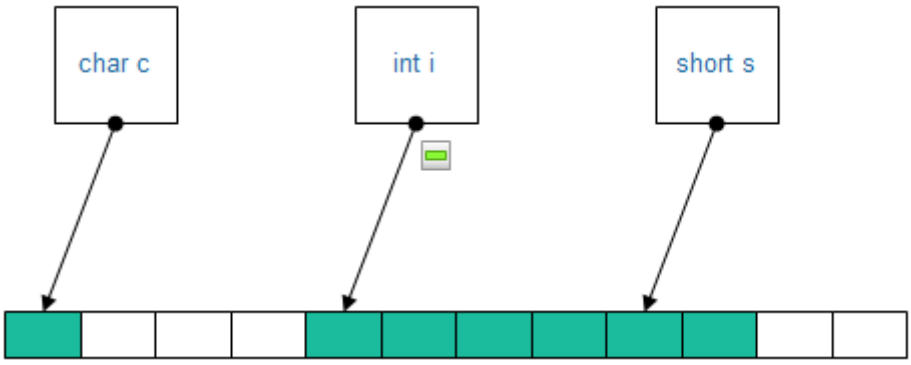
```
#include<iostream>
using namespace std;
#pragma pack(1)//设定为 1 字节对齐
class test {
private :

    char c='1';//1byte
    int i;//4byte
    short s=2;//2byte
};

class test2 {
private:
    int i;//4byte
    char c = '1';//1byte
    short s = 2;//2byte
};
int main(){
    cout << sizeof(test) << endl;
    cout << sizeof(test2) << endl;
    return 0;
}
```

问题分析

（1）对于类test的内存空间是这样的：



内存分配过程：

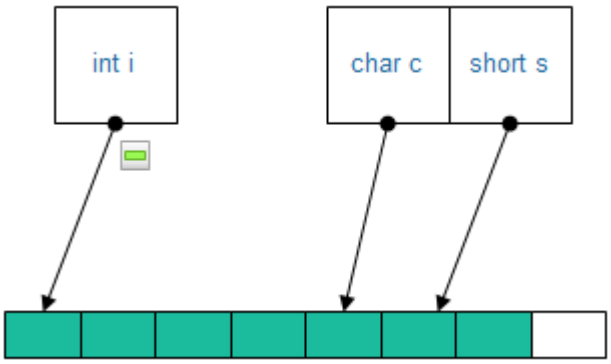
1、char和编译器默认的内存缺省分割大小比较，char比较小，分配一个字节给它。

2、int和编译器默认的内存缺省分割大小比较，int比较小，占4字节。只能空3个字节，重新分配4个字节。

3、short和编译器默认的内存缺省分割大小比较，short比较小，占2个字节，分配2个字节给它。

4、对齐结束类本身也要对齐，所以最后空余的2个字节也被test占用。

（2）对于类test2的内存空间是这样的：

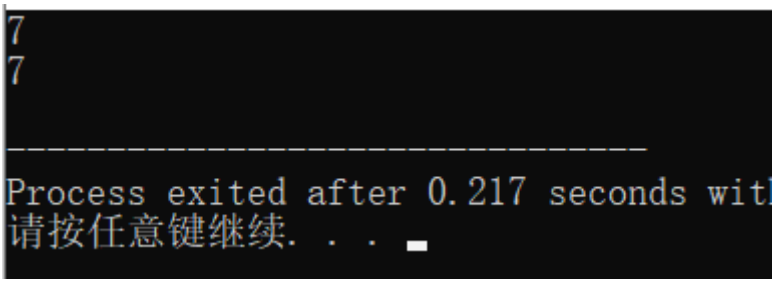


1、int和编译器默认的内存缺省分割大小比较，int比较小，占4字节。分配4个字节给int。

2、char和编译器默认的内存缺省分割大小比较，char比较小，分配一个字节给它。

3、short和编译器默认的内存缺省分割大小比较，short比较小，此时前面的char分配完毕还余下3个字节，足够short的2个字节存储，所以short紧挨着。分配2个字节给short。

4、对齐结束类本身也要对齐，所以最后空余的1个字节也被test占用。



最后当我们把编译器的内存分割大小设置为1后，类中所有的成员变量都紧密的连续分布。

为什么需要内存对齐：

因为cpu按照固定的字节读入，如果不对齐cpu需要多次读入会降低cpu效率

原博客

- 作者：钱书康

- 欢迎转载，请保留此段声明。
- 出处：http://www.cnblogs.com/zrtqsk/

inline函数（重要）

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。所以，如果函数体代码过长或者函数体重有循环语句，if语句或switch语句或递归时，不宜用内联

关键字inline 必须与函数定义体放在一起才能使函数成为内联，仅将inline 放在函数声明前面不起任何作用。内联函数调用前必须声明。即

```
void Foo(int x, int y);
inline void Foo(int x, int y) // inline 与函数定义体放在一起
{
    ...
}
```

inline 是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。

内联和宏的比较

宏代码本身不是参数，但使用起来像函数。编译预处理用拷贝宏代码的方式取代函数调用，省去了参数压栈、生成汇编语言的CALL调用、返回参数、执行return等过程，从而调高了速度。**使用宏代码最大的缺点是容易出错，预处理器在拷贝宏代码时常常产生意想不到的边际效应。**

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

在执行语句时，

```
result = MAX(i, j) + 2 ;
```

时,会被解释成

```
result = (i) > (j) ? (i) : (j) + 2 ;
```

使用宏，无法访问类的私有成员,但是也有一些奇技淫巧可以做到，虽然基本不会用，C++ 通过内联机制，既具备宏代码的效率，又增加了安全性

缺点:

1.内联函数具有一定的局限性，内联函数的函数体一般来说不能太大，如果内联函数的函数体过大，一般的编译器会放弃内联方式，而采用普通的方式调用函数。(换句话说就是，你使用内联函数，只不过是向编译器提出一个申请，编译器可以拒绝你的申请）这样，内联函数就和普通函数执行效率一样了。

2.nline说明对编译器来说只是一种建议，编译器可以选择忽略这个建议。比如，你将一个长达1000多行的函数指定为inline，编译器就会忽略这个inline，将这个函数还原成普通函数，因此并不是说把一个函数定义为inline函数就一定会被编译器识别为内联函数，具体取决于编译器的实现和函数体的大小。

内联函数和宏定义的区别

内联函数和宏的区别在于，宏是由预处理器对宏进行替代，而内联函数是通过编译器控制来实现的。而且内联函数是真正的函数，只是在需要用到的时候，内联函数像宏一样的展开，所以取消了函数的参数压栈，减少了调用的开销。你可以象调用函数一样来调用内联函数，而不必担心会产生于处理宏的一些问题。内联函数与带参数的宏定义进行下比较，它们的代码效率是一样，但是内联欢函数要优于

宏定义，因为内联函数遵循的类型和作用域规则，它与一般函数更相近，在一些编译器中，一旦关联上内联扩展，将与一般函数一样进行调用，比较方便。

另外，宏定义在使用时只是简单的文本替换，并没有做严格的参数检查，也就不能享受C++编译器严格类型检查的好处，另外它的返回值也不能被强制转换为可转换的合适的类型，这样，它的使用就存在着一系列的隐患和局限性。

智能指针（重要）

c++primer plus 6th page667

https://blog.csdn.net/albertsh/article/details/82286999

虚函数及工作原理（重要）

当子类重新定义了父类的虚函数后，**当父类的指针指向子类对象的地址时**，[即B b; A a = &b;] 父类指针根据赋给它的不同子类指针，动态的调用子类的该函数，且这样的函数调用发生在运行阶段，而不是发生在编译阶段，称为动态联编。

```
Base *b=new Derived();
Derived *d=new Derived();
b只能访问 Base的成员，因为能否访问取决于指针的类型而不是分配的类型
```

底层机制

为每个类对象添加一个隐藏成员，隐藏成员中保存了一个指向函数地址数组的指针，称为虚表指针（vptr），这种数组成为虚函数表（virtual function table, vtbl），即，**每个类使用一个虚函数表，每个类对象用一个虚表指针。**

看下面两种情况：

1. 如果派生类重写了基类的虚方法，该派生类虚函数表将保存重写的虚函数的地址，而不是基类的虚函数地址。
2. 如果基类中的虚方法没有在派生类中重写，那么派生类将继承基类中的虚方法，而且派生类中虚函数表将保存基类中未被重写的虚函数的地址。

   https://www.cnblogs.com/zkfopen/p/11061414.html

在c++中，如果类中含有虚函数表。编译器就会对该类产生一个虚函数表。

1. 表里有很多项，每一项都是一个指针，每个指针指向的是这个类的各个虚函数的入口地址。
2. 虚函数表项里，第一个表项很特殊，指向的不是虚函数的入口地址，他指向的是类所关联的tpye_info对象。

   从节省内存的角度上说，应该是一个类一个，同一个类的不同对象拥有相同虚函数表。我们用代码来试一试

为什么构造函数不能为虚函数（重要）

1.从存储空间角度

虚函数对应一个vtable，这大家都知道，可是这个vtable其实是存储在对象的内存空间的。问题出来了，如果构造函数是虚的，就需要通过 vtable来调用，可是对象还没有实例化，也就是内存空间还没有，无法找到vtable，所以构造函数不能是虚函数。

重载与重写

https://www.cnblogs.com/zhangjxblog/p/8723291.html

重写的基类中被重写的函数必须有virtual修饰。

tcp和udp （重要）

计算机网络 （谢希仁）

骨骼蒙皮动画（SkinnedMesh）是什么

骨骼蒙皮动画的出现解决了关节动画的裂缝问题。骨骼动画的基本原理可概括为：在骨骼控制下，通过顶点混合动态计算蒙皮网格的顶点，而骨骼的运动相对于其父骨骼，并由动画关键帧数据驱动。一个骨骼动画通常包括骨骼层次结构数据，网格(Mesh)数据，网格蒙皮数据(skin info)和骨骼的动画(关键帧)数据。

法线贴图是什么

在进行物体的光照计算的时候，需要考虑物体表面的法线向量，而我们根据物体表面的细节对法线向量进行修改，这样就会获得更复杂的效果。这种每个fragment使用各自的法线，替代一个面上所有fragment使用同一个法线的技术叫做法线贴图（normal mapping）或凹凸贴图（bump mapping

方法中const的作用

前面一个const表明返回值不能修改，后一个const表明方法只能读取类的成员变量，不能给成员变量赋值。

指针和引用的区别（重要）

1. 指针存的是地址，指向内存的一个存储单元，而引用是和原变量实质是同一个对象，引用类似一种别名
2. 引用在初始化后就不能再修改了，指针还可以修改
3. 有const指针，但是没有const引用
4. 指针可以有多级，但是引用只能是一级（int **p；合法 而 int &&a是不合法的）
5. 指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化；
6. 指针和引用的自增(++)运算意义不一样；引用和指针自增的不同在于引用是指值自增，而指针的自增指的是指向下一段内存地址

指针和数组的区别（重要）

指针	数组
保存数据的地址	保存数据
间接访问数据，首先获得指针的内容，然后将其作为地址，从该地址中提取数据	直接访问数据，
通常用于动态的数据结构	通常用于固定数目且数据类型相同的元素
通过Malloc分配内存，free释放内存	隐式的分配和删除
通常指向匿名数据，操作匿名函数	自身即为数据名

malloc的底层实现原理

https://www.cnblogs.com/zpcoding/p/10808969.html

1）当开辟的空间小于 128K 时，调用 brk（）函数，malloc 的底层实现是系统调用函数 brk（），其主要移动指针 _enddata(此时的 _enddata 指的是 Linux 地址空间中堆段的末尾地址，不是数据段的末尾地址)

2）当开辟的空间大于 128K 时，mmap（）系统调用函数来在虚拟地址空间中（堆和栈中间，称为“文件映射区域”的地方）找一块空间来开辟。

进程间的通信方式（重要）

https://blog.csdn.net/qq_42052956/article/details/111499122

动画序列和动画蒙太奇有什么区别

我认为动画序列就是许多动画帧的序列，而动画蒙太奇在序列的基础上可以设置通知事件，当动画播放到某个通知时就可以触发相应的事件。我们可以通过蓝图或者c++控制动画蒙太奇，动画蒙太奇可以组合多个动画序列

大小端存储

小端：是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低。

大端：是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；

A9

高字节就是指16进制数的前8位（权重高的8位），如上例中的A。
低字节就是指16进制数的后8位（权重低的8位），如上例中的9。

堆栈和内存增长方向问题：

堆：生长方向是向上的，也就是向着内存地址增加的方向。通常我们在画内存四区图时，堆的开口是向上的。

栈：它的生长方式是向下的，是向着内存地址减小的方向增长。栈的开口是向下的，上面的底部是栈底，下面的开口是栈顶。

C++内存布局6大区和5大区

https://blog.csdn.net/Code_beeps/article/details/89608929

https://blog.csdn.net/hummingbird0/article/details/89713211

类继承和对象组合

一：继承

继承是Is a 的关系，比如说Student继承Person,则说明Student is a Person。继承的优点是子类可以重写父类的方法来方便地实现对父类的扩展。

继承的缺点有以下几点：

①：父类的内部细节对子类是可见的。

②：子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为。

③：如果对父类的方法做了修改的话（比如增加了一个参数），则子类的方法必须做出相应的修改。所以说子类与父类是一种高耦合，违背了面向对象思想。

二：组合

组合也就是设计类的时候把要组合的类的对象加入到该类中作为自己的成员变量。

组合的优点：

①：当前对象只能通过所包含的那个对象去调用其方法，所以所包含的对象的内部细节对当前对象时不可见的。

②：当前对象与包含的对象是一个低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码。

③：当前对象可以在运行时动态的绑定所包含的对象。可以通过set方法给所包含对象赋值。

组合的缺点：①：容易产生过多的对象。②：为了能组合多个对象，必须仔细对接口进行定义。

由此可见，组合比继承更具灵活性和稳定性，所以在设计的时候优先使用组合。只有当下列条件满足时才考虑使用继承：

- 子类是一种特殊的类型，而不只是父类的一个角色
- 子类的实例不需要变成另一个类的对象
- 子类扩展，而不是覆盖或者使父类的功能失效

多重继承中的二义性问题

即 继承的多个父类中有多个相同的方法或成员会造成二义性

可以通过三种方法解决

1.使用域运算符::指定是哪个父类的方法或成员

```
e.Base1::a();
e.Base2::a();
```

2.使用同名覆盖原则

在子类中重写父类相同的成员和方法

3.使用虚基类

虚基类的声明是在派生类的声明过程中进行的，其声明的一般形式为：
class<派生类名>:virtual<派生方式><基类名>

class Base1:virtual public Base*//virtual*

new和malloc的区别（重要）

1.new是C++关键字，需要编译器支持；malloc是库函数，需要头文件支持。

2.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。而malloc则需要显式地指出所需内存的尺寸。

3.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。而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。

4.new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。而malloc是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

5.C++允许自定义operator new 和 operator delete 函数控制动态内存的分配。

6.new做两件事，分别是分配内存和调用类的构造函数，而malloc只是分配和释放内存。new操作符从自由存储区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。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。自由存储区不等于堆，如上所述，布局new就可以不位于堆中。

7.new内存分配失败时，会抛出bac_alloc异常。malloc分配内存失败时返回NULL。

8.内存泄漏对于new和malloc都能检测出来，而new可以指明是哪个文件的哪一行，malloc不可以。

C++内存泄漏与避免（重要）

定义：指程序申请内存后，无法释放已申请的内存空间，久而久之系统就没有内存可以再分配了。

valgrind避免泄漏

https://blog.csdn.net/weixin_44718794/article/details/107071169

UE4的反射系统（了解）

https://blog.csdn.net/u012999985/article/details/52902065

反射就是 在程序运行期间，动态的获取类或者对象的属性，是程序在运行时进行自检的一种能力，检查自己的C++类，函数，成员变量，结构体等等

java中的反射

在java中，反射的原理就是通过类的字节码文件(class文件)反向获取该类或者对象中的属性，既然是通过字节码获取，这就需要JVM的操作了。

反射的优点：

1，可以在程序运行的过程中，操作这些对象。
2，可以解耦，提高程序的可扩展性。

UE4的反射

在UE4中我们会在变量和函数前面加上uproperty或者ufunction可以实现各种各样的功能，这个就是通过ue4的反射系统来实现的

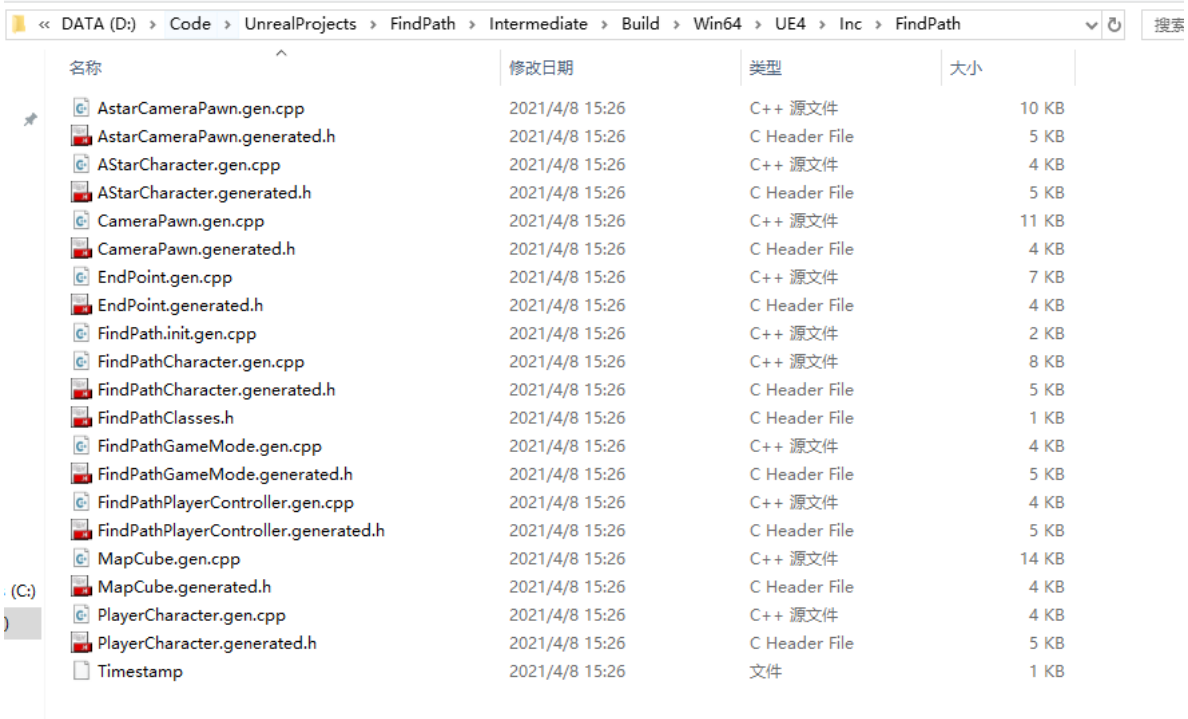
有了反射系统就是可以更自由的控制这些结构，让他在我们想出现的地方出现，让他在我们想使用的地方使用。我们可以使用这个类进行网络复制，执行垃圾回收，让他和蓝图交互等等。

反射系统的基本原理

项目工程中有两个文件很重要，“.generate.h”以及“.generate.cpp”。

“.generate.h”文件在每一个类的声明前面都会被包含到对应的头文件里面。而“.generate.cpp”对于一整个项目只会有一个。他们是通过Unreal Build Tool(UBT) 虚幻编译工具和UnrealHeaderTool（UHT）虚幻头文件分析工具来生成的。

“.generate.h”里面是宏，而且包含一个非常庞大的宏，这个宏把所有和反射相关的方法（包括定义）和结构体连接到一起。而“.generate.cpp”里面是许多的函数定义，UnrealHeaderTool根据你在头文件里面使用的宏（UFUNCTION等）自动的生成这个文件，所以这个文件并不需要你去修改，也不允许修改。



虚幻头文件分析工具（UHT） 是支持UObject系统的自定义解析和代码生成工具。代码编译分两个阶段进行：

1. 调用UHT。它将解析C++头文件中引擎相关类元数据，并生成自定义代码，以实现诸多UObject相关的功能。
2. 调用普通C++编译器，以便对结果进行编译。

UBT属性通过扫描头文件，记录任何至少有一个反射类型的头文件的模块。如果其中任意一个头文件从上一次编译起发生了变化，那么 UHT就会被调用来利用和更新反射数据。UHT分析头文件，创建一系列反射数据，并且生成包含反射数据的C++代码（也就是".generate.cpp"）以及各种辅助函数与thunk函数（".generate.h"）。

UE4的垃圾回收系统（了解）

ue4中我们创建一个物体之后不管他，那他会被垃圾回收吗？答：不会，一般没有被引用的物体会被回收，但是他不会被回收，因为他被他所在的关卡所引用

结构体和联合体的区别（重要）

联合体
用途：使几个不同类型的变量共占一段内存(相互覆盖)

联合体中的变量共用一个内存位置，长度为联合体中最大的变量长度

结构体是一种构造数据类型
用途：把不同类型的数据组合成一个整体-------自定义数据类型

结构体变量所占内存长度是各成员占的内存长度的总和。

1. struct和union都是由多个不同的数据类型成员组成, 但在任何同一时刻, union中只存放了一个被选中的成员, 而struct的所有成员都存在。在struct中，各成员都占有自己的内存空间，它们是同时存在的。一个struct变量的总长度等于所有成员长度之和。在Union中，所有成员不能同时占用它的内存空间，它们不能同时存在。Union变量的长度等于最长的成员的长度。
2. 对于union的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了, 而对于struct的不同成员赋值是互不影响的。

大型游戏地图无缝加载如何做到的

有一个实现的方式是场景预加载。

比如说我们有一个100平方公里的正方形大地图，那我们把它分为100个1平方公里的区域，当玩家角色进入某个区域时，我们就一并加载这个区域周围相邻的8个区域。例如玩家进入55区域，那我们为其加载44，45，46，54，56，64，65，66这8个区域，玩家进入这相邻8个区域的时候无需加载即可进入，同时再预加载其他相邻区域，释放不相邻区域的资源即可。

override关键字

override保留字表示当前函数重写了基类的虚函数。

目的：

1.在函数比较多的情况下可以提示读者某个函数重写了基类虚函数（表示这个虚函数是从基类继承，不是派生类自己定义的）；

2.强制编译器检查某个函数是否重写基类虚函数，如果没有则报错。

```
class Base {
virtual void f();
};

class Derived : public Base {
void f() override; // 表示派生类重写基类虚函数f
void F() override;//错误：函数F没有重写基类任何虚函数
};
```

> 注意：override只是C++保留字，不是关键字，这意味着只有在正确的使用位置，oerride才启“关键字”的作用，其他地方可以作为标志符（如：int override；是合法的）。

decltype关键字

decltype与auto关键字一样，用于进行编译时类型推导。
decltype实际上有点像auto的反函数，auto可以让你声明一个变量，而decltype则可以从一个变量或表达式中得到类型

```
int x = 3;
decltype(x) y = x;
```

和auto的区别

mutable 关键字

https://liam.page/2017/05/25/the-mutable-keyword-in-Cxx/

可以造成逻辑上修改const，外部观察，它是常量而不可修改；但是内部可以有非常量的状态。

比如函数 加了const表示 ，数据成员是不可修改的但是加了mutable后，可以允许在内部修改变量，但是对外还是不变的。

emplace关键字

vector的push_back，map的insert，set的insert。这些插入操作会涉及到两次构造，首先是对象的初始化构造，接着在插入的时候会复制一次，会触发拷贝构造。但是很多时候我们并不需要两次构造带来效率的浪费，如果可以在插入的时候直接构造，就只需要构造一次就够了。所以就有了emplace

volatile 关键字

https://www.cnblogs.com/likui360/p/6036601.html

=default

虚拟内存（重要）

物理内存的缺点

1. 内存空间利用率的问题

各个进程对内存的使用会导致内存碎片化，当要用malloc分配一块很大的内存空间时，可能会出现虽然有足够多的空闲物理内存，却没有足够大的连续空闲内存这种情况，东一块西一块的内存碎片就被浪费掉了

2. 读写内存的安全性问题

物理内存本身是不限制访问的，任何地址都可以读写，而现代操作系统需要实现不同的页面具有不同的访问权限，例如只读的数据等等

3. 进程间的安全问题

各个进程之间没有独立的地址空间，一个进程由于执行错误指令或是恶意代码都可以直接修改其它进程的数据，甚至修改内核地址空间的数据，这是操作系统所不愿看到的

4. 内存读写的效率问题

当多个进程同时运行，需要分配给进程的内存总和大于实际可用的物理内存时，需要将其他程序暂时拷贝到硬盘当中，然后将新的程序装入内存运行。由于大量的数据频繁装入装出，内存的使用效率会非常低

虚拟内存的优点

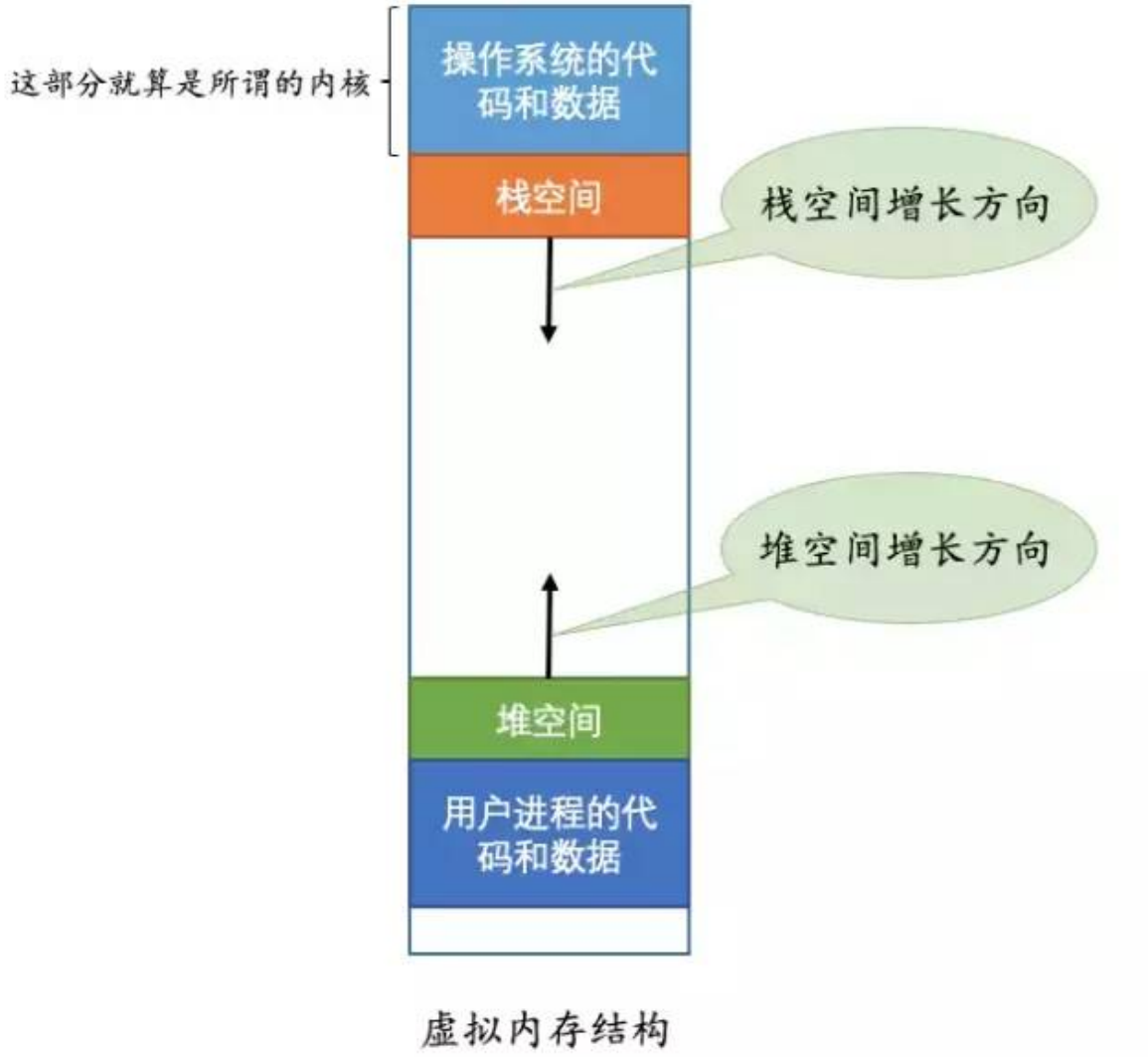
1. 它将主存看成是一个存储在磁盘空间上的地址空间的高速缓存，主存中只保存活动区域，并根据需要在磁盘和主存之间来回传送数据。
2. 它为内阁进程提供了一致的地址空间,简化了内存管理。
3. 它保护了每个进程的地址空间不被其他进程破坏。

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

虚拟地址空间：在32位的i386 CPU的地址总线的是32位的，也就是说可以寻找到4G的地址空间。我们的程序被CPU执行，就是在0x00000000到0xFFFFFFFF这一段地址中。高2G的空间为内核空间，由操作系统调用，低2G的空间为用户空间，由用户使用。

https://www.jianshu.com/p/13e337312651

CPU在寻址的时候，是按照虚拟地址来寻址，然后通过MMU(内存管理单元)将虚拟地址转换为物理地址。因为只有程序的一部分加入到内存中，所以会出现所寻找的地址不在内存中的情况（CPU产生缺页异常），如果在内存不足的情况下，就会通过页面调度算法来将内存中的页面置换出来，然后将在外存中的页面加入到内存中，使程序继续正常运行。



虚拟内存结构

1、每次我要访问地址空间上的某一个地址，都需要把地址翻译为实际物理内存地址；
2、所有进程共享这整一块物理内存，每个进程只把自己目前需要的虚拟地址空间映射到物理内存上；
3、进程需要知道哪些地址空间上的数据在物理内存上，哪些不在（可能这部分存储在磁盘上），还有在物理内存上的哪里，这就需要通过页表来记录；
4、页表的每一个表项分两部分，第一部分记录此页是否在物理内存上，第二部分记录物理内存页的地址（如果在的话）；
5、当进程访问某个虚拟地址的时候，就会先去看页表，如果发现对应的数据不在物理内存上，就会发生缺页异常；
6、缺页异常的处理过程，操作系统立即阻塞该进程，并将硬盘里对应的页换入内存，然后使该进程就绪，如果内存已经满了，没有空地方了，那就找一个页覆盖，至于具体覆盖的哪个页，就需要看操作系统的页面置换算法是怎么设计的了。

调试打断点的原理

https://zhuanlan.zhihu.com/p/34003929

软件断点在x86系统中指令就是INT 3

调试处理程序会把自己附在 程序的进程上，（操作系统的INT 3中断处理器会寻找注册在该进程上的调试处理程序）通过程序pe文件的debug节找到调试信息，在调试信息中找到加断点行所在的机器代码，并把头一个字节用WriteProcessMemory()函数换成0xCC(INT 3)

平衡二叉树和红黑树的区别（重要）

红黑树的性质：

1 所有节点只有红色和黑色

2 红色节点的子节点必定为黑色（不存在相邻两个节点都为红色）

3 所有叶子节点为黑色

4 从一节点出发到叶子节点的所有路径上 黑色节点的个数是相同的

5 根节点为黑色

二叉排序树在进行查找的时候时间复杂度为O（h）h为树高，但是当遇到斜树时效率变成了O(n)，所以想要把树高维持到logn可以用AVL和RBT

比较

AVL 树比红黑树更加平衡，但AVL树在插入和删除的时候也会存在大量的旋转操作。所以当你的应用涉及到频繁的插入和删除操作，切记放弃AVL树，选择性能更好的红黑树；当然，如果你的应用中涉及的插入和删除操作并不频繁，而是查找操作相对更频繁，那么就优先选择 AVL 树进行实现。

红黑树的应用

大多数自平衡BST(self-balancing BST) 库函数都是用红黑树实现的，比如C++中的map 和 set （或者 Java 中的 TreeSet 和 TreeMap）。

红黑树也用于实现 Linux 操作系统的 CPU 调度。完全公平调度（Completely Fair Scheduler）使用的就是红黑树。

如何取到非静态类成员变量（重要）

对象的地址即数据成员所在的地址，想要对非静态成员进行存取操作，编译器需要把class object的起始地址加上数据成员的偏移位置（即根据不同类型，偏移大小不同？）

为了区别“一个指向class的第一个成员”和“没有指向任何成员”两种情况

详情：深入探索c++对象模型

类中的成员函数与inline（重要）

定义！在类中的**成员函数**默认都是**内联的**

如果在类中未给出成员函数定义，而又想内联该函数的话，那在类外要加上 inline，否则就认为不是内联的。

```
class A{
    public:void Foo(int x, int y) { } // 自动地成为内联函数
}
```

一致性哈希算法 写一下建树

解决hash冲突？负债均衡

https://www.bilibili.com/video/BV1ei4y1L7bd

c++ extern

它告诉编译器存在着一个变量或者一个函数，如果在当前编译语句的前面中没有找到相应的变量或者函数，也会在当前文件的后面或者其它文件中定义。

extern关键字的主要作用是扩大变量/函数的作用域，使得其它源文件和头文件可以复用同样的变量/函数

extern有两个作用

第一个,当它与"C"一起连用时，如: extern "C" void fun(int a, int b);则告诉编译器在编译fun这个函数名时按着C的规则去翻译相应的函数名而不是C++的，C++的规则在翻译这个函数名时会把fun这个名字变得面目全非

第二，当extern不与"C"在一起修饰变量或函数时，如在头文件中: extern int g_Int; 它的作用就是声明函数或全局变量的作用范围的关键字，其声明的函数和变量可以在本模块活其他模块中使用，记住它是一个声明不是定义!也就是说B模块(编译单元)要是引用模块(编译单元)A中定义的全局变量或函数时，它只要包含A模块的头文件即可,在编译阶段，模块B虽然找不到该函数或变量，但它不会报错，它会在连接时从模块A生成的目标代码中找到此函数。

内存碎片化以及解决方案（重要）

1. 内部碎片是由于采用固定大小的内存分区，当一个进程不能完全使用分给它的固定内存区域时就产生了内部碎片，通常内部碎片难以完全避免；
2. 外部碎片是由于某些未分配的连续内存区域太小，以至于不能满足任意进程的内存分配请求，从而不能被进程利用的内存区域。

段页式内存分配方式就是将进程的内存区域分为不同的段，然后将每一段由多个固定大小的页组成。通过页表机制，使段内的页可以不必连续处于同一内存区域，从而减少了外部碎片，然而同一页内仍然可能存在少量的内部碎片，只是一页的内存空间本就较小，从而使可能存在的内部碎片也较少。

代价：程序需要记录的是内存页ID，每次使用时，需要从内存页ID翻译成实际内存地址，多了一次转换。而且这种模式，会浪费一些内存

https://blog.csdn.net/tong5956/article/details/74937178

四种cast转换（重要）

https://zhuanlan.zhihu.com/p/265608625

static因为在向下进行转换时没有动态类型检查所有不安全

RTTI

https://www.jianshu.com/p/3b4a80adffa7

完美转发（重要）

http://shaoyuan1943.github.io/2016/03/26/explain-move-forward/

http://c.biancheng.net/view/7868.html

无论目的调用函数需要哪种类型的参数都可以正确调用到我们想要的那个函数里。C++11中实现完美转发是依靠的类型推导和引用折叠。

std::forward()与std::move()相区别的是，move()会无条件的将一个参数转换成右值，而forward()则会保留参数的左右值类型。

引用折叠规则就是函数接受参数形式与传入参数形式之间进行引用简化，具体编译器定义了这样一条规则

```
1. T& + & => T&
2. T&& + & => T&
3. T& + && => T&
4. T&& + && => T&&
```

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

//重载被调用函数，查看完美转发的效果
void otherdef(int & t) {
    cout << "lvalue\n";
}
void otherdef(const int & t) {
    cout << "rvalue\n";
}

//实现完美转发的函数模板
template <typename T>
void function(T&& t) {
    otherdef(forward<T>(t));
}

int main()
{
    function(5);
    int  x = 1;
    function(x);
    return 0;
}
```

进程调度算法（重要）

操作系统p99

进程概述和内存分配（重要）

https://blog.csdn.net/liangjiabao5555/article/details/88991252

线程间和进程间通信方式（重要）

https://www.jianshu.com/p/eb3e3019e7b9

7种

C/C++程序从文本文件到可执行目标文件都做了什么（重要）

来源：深入理解计算机系统（CSAPP）

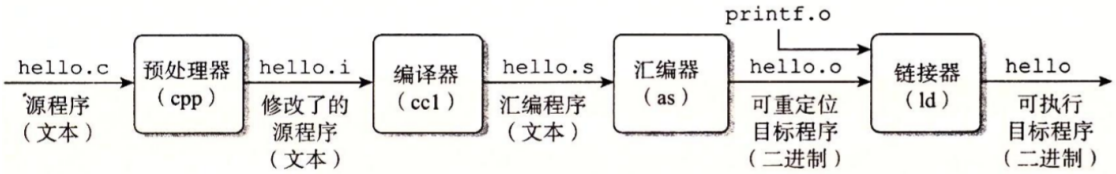


图 1-3　编译系统

1.预处理阶段

预处理器根据以字符#开头的命令，修改原始的c程序。比如#include<stdio.h>命令告诉预处理器读取系统头文件stdio.h的内容，并把它直接插入程序文本中。结果就得到了另一个c程序,通常是以.i作为文件扩展名(环境在linux下)

2.编译阶段

编译器(ccl)将文本文件hello.i翻译成文本文件hello.s 它包含了一个汇编语言程序。它为不同高级语言的不同编译器提供了通用的输出语言。

3.汇编 阶段

汇编器(as)将hello.s翻译成机器语言指令，把这些指令打包成文件hello.o中。hello.o文件是一个二进制文件

4.链接阶段

hello程序调用了printf函数，它是每个c编译器都提供的标准c库中的一个函数。printf函数存在于一个名为printf.o的单独的预编译好的目标文件中，而这个文件必须以某种方式合并到我们的hello,o程序中，链接器就负责处理这种合并。结果就得到了hello文件，它是一个可执行目标文件或简称为可执行文件，可以被加载到内存中，由系统执行。

---

1. gcc -E source_file.c
   -E，只执行到预编译。直接输出预编译结果。
2. gcc -S source_file.c
   -S，只执行到源代码到汇编代码的转换，输出汇编代码。

3. gcc -c source_file.c
   -c，只执行到编译，输出目标文件。
4. gcc (-E/S/c/) source_file.c -o output_filename
   -o, 指定输出文件名，可以配合以上三种标签使用。
   -o 参数可以被省略。这种情况下编译器将使用以下默认名称输出：
   -E：预编译结果将被输出到标准输出端口（通常是显示器）
   -S：生成名为source_file.s的汇编代码

-c：生成名为source_file.o的目标文件。

无标签情况：生成名为a.out的可执行文件。

动态链接库和静态链接库（重要）

静态库在编译时链接到程序中，动态库是运行时用到了才会链接到程序中

lib文件是必须在编译期就连接到应用程序中的，而dll文件是运行期才会被调用的。如果有dll文件，那么对应的lib文件一般是一些索引信息，具体的实现在dll文件中。如果只有lib文件，那么这个lib文件是静态编译出来的，索引和实现都在其中。

this指针存在的意义（重要）

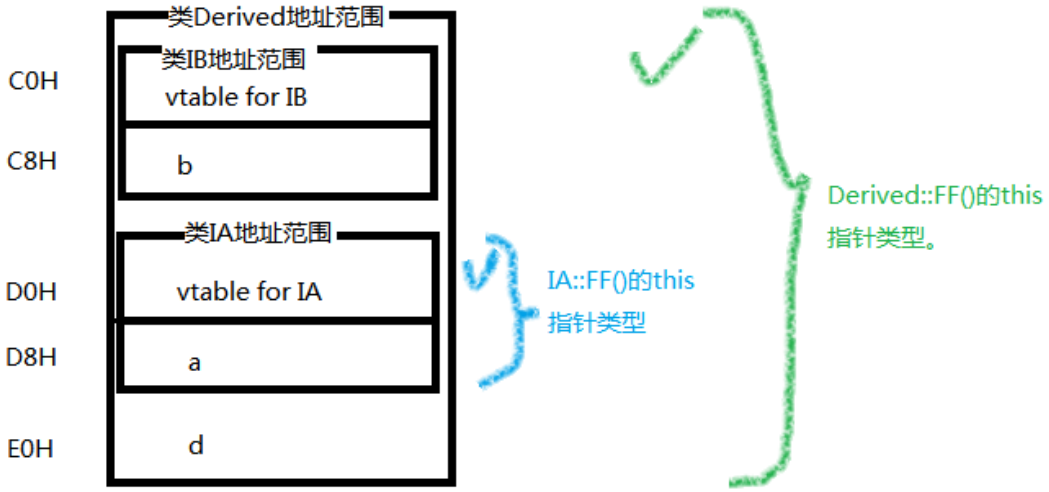
https://blog.csdn.net/w903414/article/details/108974260

1. 只能在“成员函数”中使用
2. `this`指针类型： **类类型 *const** （加const是为了保证，指针的指向不被更改）
3. `this`指针不存储在对象中，不影响对象大小，且始终指向当前对象
4. `this`指针是“成员函数”的第一个隐藏参数，由编译器自动给出
5. 主要是通过ecx寄存器来传递（并不是所有的）

对象内存分布（重要）

https://blog.csdn.net/weixin_30511039/article/details/95952988

主要看那个图



为什么long和int都是4字节

C++标准上只是说long至少要和int一样大，所有整数类型实现时要满足如下规范：

```
sizeof(char)  ==  1

sizeof(char)  <=  sizeof(short)

sizeof(short)  <=  sizeof(int)

sizeof(int)  <=  sizeof(long)

sizeof(long)  <=  sizeof(long long)
```

除了char和long long，其余的类型范围较灵活，都是平台相关的，与实现相关。如果要实现平台独立的话，在windows平台上，就有__intn可以使用，n代表位数。__int8 __int16 __int32 __int64。

int的字节一定是4字节吗

在32位和64位都是4字节，long在32位上是4字节，在64位上是8字节，

序列化和反序列化

序列化：对象转换为字节序列的过程，将数据分解成字节流，以便存储在文件中或在网络上传输。

反序列化：打开字节流并重构对象

为什么需要序列化和反序列化

当两个进程进行远程通信时，可以相互发送各种类型的数据，包括文本、图片、音频、视频等， 而这些数据都会以二进制序列的形式在网络上传送。

修改一个变量的过程（有点迷）

1.程序计数器PC给出访问数据单元的地址，

2.地址通过地址缓冲器送到地址总线，

3.再经过地址译码找到相应单元，

4.在cpu控制单元的控制下（比如读控制）将地址的数据送到数据总线，

5.再由数据总线送到数据缓冲器，经过内部总线送到指令寄存器

/// 先把数据从地址（通过地址缓冲器和地址译码）读到寄存器中，然后进行修改，再经过地址缓冲器和地址译码找到地址写回去？？

struct和class区别（重要）

1. 成员访问权限不同
2. 在继承体系中，struct默认public继承，class是private继承
3. 如果用class继承struct 默认是private**默认的防控属性取决于子类而不是基类**
4. class可以用于定义模板参数，struct不能

5. **当你觉得你要做的更像是一种数据结构集合的话，那么用struct。如果你要做的更像是一种对象的话，那么用class。**

锁上加锁

法线贴图的原理

非递归实现二叉树的镜像(使用栈和广搜的思路)

顶点着色器，如果我想点一下就让图上的三角形变成我想要的颜色，这个变量如何传递

---

memcpy和vector

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

int main()
{
    vector<int> v1={1,2,3};
    vector<int> v2(3);
    memcpy(&v2,&v1,3);
    for(auto i:v2)
        cout<<i<<" ";
    //使用memcpy时,被拷贝的对象里面存在动态内存.
    //比如:vector对象大小无法确定，memcpy不管这事直接拷贝sizeof大小的内存，
    //导致vector的内存结构破坏，vector析构时就出错了

//  char str1[]="123456";
//  char str2[10];
//  memcpy(str2,str1,3);
//  cout<<str2;
    return 0;
}
```

C++和C的区别（重要）

设计思想上：

C++是面向对象的语言，而C是面向过程的结构化编程语言

语法上：

C++具有封装、继承和多态三种特性

C++相比C，增加多许多类型安全的功能，比如强制类型转换、

C++支持范式编程，比如模板类、函数模板等

如何判断两个IP是否在同一个网段内

https://zhidao.baidu.com/question/567773789.html

静态函数可以是或者访问虚函数吗（重要）

虚函数需要具体的对象实例来访问，而静态函数不属于某个具体的对象

静态函数和全局函数的区别

多线程中栈和堆的权限

在多线程环境下，每个线程拥有一个栈和一个程序计数器。栈和程序计数器用来保存线程的执行历史和线程的执行状态，是线程私有的资源。其他的资源（比如堆、地址空间、全局变量）是由同一个进程内的多个线程共享。

多线程（重要）

https://blog.csdn.net/Primeprime/article/details/79080015?utm_medium=distribute.pc_relevant.none-task-blog-baidujs_baidulandingword-1&spm=1001.2101.3001.4242

临界区

“每个进程中访问临界资源的那段代码称为临界区(Critical Section)(临界资源是一次仅允许一个进程使用的共享资源)。每次只准许一个进程进入临界区,进入后不允许其他进程进入。不论是硬件临界资源,还是软件临界资源,多个进程必须互斥地对它进行访问。多个进程中涉及到同一个临界资源的临界区称为相关临界区。.”

临界区和互斥量的区别

1）互斥量是内核对象，所以它比临界区更加耗费资源，但是它可以命名，因此可以被其它进程访问

2）从目的是来说，临界区是通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。

互斥量是为协调共同对一个共享资源的单独访问而设计的。

自旋锁和互斥锁

自旋锁与互斥锁有点类似，只是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，**“自旋锁”的作用是为了解决某项资源的互斥使用。因为自旋锁不会引起调用者睡眠，所以自旋锁的效率远高于互斥锁。**

自旋锁的不足之处：

自旋锁一直占用着CPU，他在未获得锁的情况下，一直运行（自旋），所以占用着CPU，如果不能在很短的时间内获得锁，这无疑会使CPU效率降低。

自旋锁适用于锁使用者保持锁时间比较短的情况下。因为时间较段的话我们要保存进程线程的上下文，在切换时消耗资源较大

计算机网络五层结构（重要）

1.应用层

应用层的任务是通过应用进程间的交互来完成特定网络应用。应用层协议定义的是应用进程间通信和交互的规则。

不同的网络应用需要不同的协议，如万维网应用的HTTP协议，支持电子邮件的SMTP协议，支持文件传送的FTP协议等

2.运输层

运输层的任务是负责为两个主机中进程之间的通信提供通用的数据传输服务。应用进程利用该服务传送应用层 报文。

所谓通用，是指并不针对某个特定网络的应用。而是多种应用可以使用同一个运输层服务。

运输层主要使用以下两种协议：

传输控制协议TCP （提供面向连接的，可靠的数据传输服务，数据传输的单位是报文段）

用户数据报协议UDP(提供无连接的，尽最大努力交付，其数据传输的单位是用户数据报)

3.网络层

网络层为分组交换网上不同主机提供通信服务。网络层将运输层产生的报文段或用户数据报封装成分组和包进行传送。

4.数据链路层

两台主机间的数据传输，总是一段一段在数据链路上传送的，这就需要使用专门的链路层协议。在两个相邻节点间的链路上传送帧，每一帧包括数据和必要的控制信息（如同步信息，地址信息，差错控制等）

三个基本问题：封装成帧，透明传输，差错检测

5.物理层

在物理层上所传数据单位是比特。

3次握手/四次挥手

计算机网络p238

std::move和std::forward的原理

https://blog.csdn.net/ouyangjinbin/article/details/51064274

https://zhuanlan.zhihu.com/p/92486757

高斯模糊

https://zhuanlan.zhihu.com/p/358910882

卷积：

就是使用一个卷积核(kernel)对一张图像中的每个像素进行一系列操作。卷积核通常是一个四方形网格结构。（例如3X3的方形区域），该区域内每个方格都有一个权重值。当对图像中的某个像素进行卷积时，我们会把卷积核的中心放置于该图像上，依次计算核中每个元素和其覆盖的图像像素值的乘积并求和，得到的结果就是该位置的新像素值

模糊

模糊指我们对图像进行平滑化处理，那么怎么进行平衡化处理？就是我们需要用平滑滤波函数生成卷积核中对应的权重，然后对图像进行卷积处理，其中平滑滤波函数用到了高斯函数就是高斯模糊

为什么选用高斯函数?

距离某像素a越近的像素，它对a的影响越大，应当越重要，因此应该有越高的权重，这和高斯分布的函数图像是吻合的。

$$f(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{(x-v)^2}{2\sigma^2}}$$

STL中map中find和[ ]的区别

1.map::find 返回一个空迭代器（map::end）。

2.map::operator[] 将用 VALUE 默认的构造函数创建一个对象并插入到 map 中，将其返回。

STL各个容器的算法复杂度 （重要）

https://blog.csdn.net/xuefeiliuyuxiu/article/details/73555683

优先队列

https://www.coder.work/article/810830

图形学基础

https://zhuanlan.zhihu.com/p/79183044

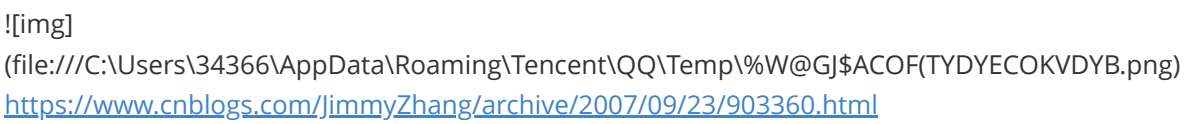
STL基础

https://blog.csdn.net/daaikuaichuan/article/details/80717222

C++ 获取类成员函数地址，并通过地址调用函数

![img]
(file:///C:\Users\34366\AppData\Roaming\Tencent\QQ\Temp\%W@GJ$ACOF(TYDYECOKVDYB.png)
https://blog.csdn.net/Simon798/article/details/113701038

委托和事件


https://www.cnblogs.com/JimmyZhang/archive/2007/09/23/903360.html