C++ notice

C++ 类成员未声明访问修饰符时默认为private

C++ 值未初始化是默认为0 或者NULL 即’\0’ 空字符。

C++ 字符串末尾自动插入 ‘\0’null 字符来作为结尾

使用参数列表来参数对象：

构造方法 在“：”后 成员变量（形式参数）完成赋值

C::C( double a, double b, double c): X(a), Y(b), Z(c)

{

....

}

拷贝构造函数 使用类的对象在创建新的对象。

编译器会自行定义一个拷贝构造，如果有指针并且动态分配（先 new 指针分配内存再赋值的）动态分配指针内存 时需要在析构时delete 指针 释放内存

classname (const classname &obj) {

// 构造函数的主体

}

形参使用对象同样调用了拷贝构造方法，结束时调用了析构方法，

友元函数 friend

声明在类中， 不属于类的成员方法（调用不是类的对象， 而是别的调用）， 可以方位类的所有成员

内敛函数 inline

函数内敛 编译器把函数副本放在每个调用函数的地方，并且函数的定义只能有一行 否则编译器会忽略 inline

c++ this 指针

隐式存在于 类中，所有成员方法都可以访问this notice 友元不能方位this 因为他不是成员函数

c++类的静态成员

静态成员在所有类的对象上只有一个副本

可以通过：： 来在类的外部访问静态对象并且修改值，

静态函数成员：

静态函数可以直接通过类来调用只需要：：范围解析就可以访问，但是静态函数只能访问类的静态成员，不能访问其他成员

并且它不能访问this ， this 需要确定的对象。

C++继承 is – a 的关系 让派生类可以拥有父类的一切（private 部分无法访问但是拥有，构造析构 拷贝都没有 ， 运算符重载也没有 ， 友元也没有）

class derived-class: access-specifier base-class

继承类型：

继承也可以声明声明访问限制符， 同样默认值是private，几乎所有继承都使用public，

以下为继承修饰带来的效果

公有继承（public）：当一个类派生自公有基类时，基类的公有成员也是派生类的公有成员，基类的保护成员也是派生类的保护成员，基类的私有成员不能直接被派生类访问，但是可以通过调用基类的公有和保护成员来访问。

保护继承（protected）： 当一个类派生自保护基类时，基类的公有和保护成员将成为派生类的保护成员。

私有继承（private）：当一个类派生自私有基类时，基类的公有和保护成员将成为派生类的私有成员。

也就是派生会把基类的方法移动到派生对于的范围下

c++ 支持多继承 以逗号分隔，使用修饰符以及类名来表达

C++ 中的函数重载

根据参数进入同名的不同方法

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。您不能仅通过返回类型的不同来重载函数。

C++ 中的运算符重载

您可以重定义或重载大部分 C++ 内置的运算符。这样，您就能使用自定义类型的运算符。

重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

//demo

Box operator+(const Box&);

//return type + “operator” keywords + “运算符”；

一元运算符： 只有一个对象 ： -- , ++ , - ,!, 对一个对象计算，

二元运算符： 两个对象：二元运算符需要两个参数，下面是二元运算符的实例。我们平常使用的加运算符（ + ）、减运算符（ - ）、乘运算符（ \* ）和除运算符（ / ）都属于二元运算符。就像加(+)运算符。

下面的实例演示了如何重载加运算符（ + ）。类似地，您也可以尝试重载减运算符（ - ）和除运算符（ / ）。

C++多态类层次关系： 继承 来实现多态。- 根据实例对象来掉层级对应的方法。

Notice！：c++为静态连接早绑定的面向对象语言，方法在编译时已经绑定好了所以需要virtual关键字来实现。

虚函数

虚函数 是在基类中使用关键字 virtual 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数。

我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为动态链接，或后期绑定。

纯虚函数 virtual int area() = 0;

C++ 接口（抽象类）

如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用 "= 0" 来指定的

C++ 动态内存

了解动态内存在 C++ 中是如何工作的是成为一名合格的 C++ 程序员必不可少的。C++ 程序中的内存分为两个部分：

// 局部变量？

栈：在函数内部声明的所有变量都将占用栈内存。

//

堆：这是程序中未使用的内存，在程序运行时可用于动态分配内存。

很多时候，您无法提前预知需要多少内存来存储某个定义变量中的特定信息，所需内存的大小需要在运行时才能确定。

在 C++ 中，您可以使用特殊的运算符为给定类型的变量在运行时分配堆内的内存，这会返回所分配的空间地址。这种运算符即 new 运算符。

如果您不需要动态分配内存，可以使用 delete 运算符，删除之前由 new 运算符分配的内存

动态申请内存可能返回null 所以需要校验

// 包换了申请内存以及判断非空

if( !(pvalue = new double ))

{

cout << "Error: out of memory." <<endl;

exit(1);

}

delete 可以在任意地方调用

delete pvalue；

对象也是可以用new 来申请内存

数组的动态内存分配：

char\* pvalue = NULL; // 初始化为 null 的指针

pvalue = new char[20]; // 为变量请求内存

delete [] pvalue; // 删除 pvalue 所指向的数组

int ROW = 2;

int COL = 3;

double \*\*pvalue = new double\* [ROW]; // 为行分配内存

// 为列分配内存

for(int i = 0; i < COL; i++) {

pvalue[i] = new double[COL];

}

for(int i = 0; i < COL; i++) {

delete[] pvalue[i];

}

delete [] pvalue;

c++ 命名空间：

定义了范围，寻找方法的范围， c++ 函数可以不依赖类， namespace 定义

namespace namespace\_name {

// 代码声明

}

using 指令向当前文件添加命名空间范围。后续方法都会默认使用该命名范围下的方法。

**C++ 模板**

泛型编程基础

函数模板 ：

template <class type> ret-type func-name(parameter list)

{

// 函数的主体

}

类模板：

template <class type> class class-name {

.

.

.

}

# C++ 预处理器

# 开始的命令 编译前处理的代码

#define 宏定义

#define macro-name replacement-text

简单的文本替换。

函数宏 函数名 函数体

#define MIN(a,b) (((a)<(b)) ? a : b)

## 条件编译

#ifndef NULL

#define NULL 0

#endif

区间内代码不会编译 所以与被注释同样效果

#if 0

不进行编译的代码

#endif

c++信号

## signal() 函数

C++ 信号处理库提供了 **signal** 函数，用来捕获突发事件。以下是 signal() 函数的语法：

void (\*signal (int sig, void (\*func)(int)))(int);

这个函数接收两个参数：第一个参数是一个整数，代表了信号的编号；第二个参数是一个指向信号处理函数的指针。

让我们编写一个简单的 C++ 程序，使用 signal() 函数捕获 SIGINT 信号。不管您想在程序中捕获什么信号，您都必须使用 **signal** 函数来注册信号，并将其与信号处理程序相关联。看看下面的实例：

c++可以传入函数指针来作为参数

c++多线程

创建线程

POSIX 线程：

#include <pthread.h>

pthread\_create (thread, attr, start\_routine, arg)

|  |  |
| --- | --- |
| **参数** | **描述** |
| thread | 指向线程标识符指针。 |
| attr | 一个不透明的属性对象，可以被用来设置线程属性。您可以指定线程属性对象，  也可以使用默认值 NULL。 |
| start\_routine | 线程运行函数起始地址，一旦线程被创建就会执行。 |
| arg | 运行函数的参数。它必须通过把引用作为指针强制转换为 void 类型进行传递  。如果没有传递参数，则使用 NULL。 |

一个线程会知道起始地址- 一个方法指针， 参数通过（void \*）强转传入 也就是转化为NULL 在转换回数据（不知道为什么是这样）

## 终止线程

使用下面的程序，我们可以用它来终止一个 POSIX 线程：

#include <pthread.h>

pthread\_exit (status)

demo

//文件名：test.cpp

#include <iostream>

#include <cstdlib>

#include <pthread.h>

using namespace std;

#define NUM\_THREADS 5

void \*PrintHello(void \*threadid)

{

// 对传入的参数进行强制类型转换，由无类型指针变为整形数指针，然后再读取

int tid = \*((int\*)threadid);

cout << "Hello Runoob! 线程 ID, " << tid << endl;

pthread\_exit(NULL);

}

int main ()

{

pthread\_t threads[NUM\_THREADS];

int indexes[NUM\_THREADS];// 用数组来保存i的值

int rc;

int i;

for( i=0; i < NUM\_THREADS; i++ ){

cout << "main() : 创建线程, " << i << endl;

indexes[i] = i; //先保存i的值

// 传入的时候必须强制转换为void\* 类型，即无类型指针

rc = pthread\_create(&threads[i], NULL,

PrintHello, (void \*)&(indexes[i]));

if (rc){

cout << "Error:无法创建线程," << rc << endl;

exit(-1);

}

}

pthread\_exit(NULL);

}

**连接和分离线程**

**demo**

pthread\_join (threadid, status)

pthread\_detach (threadid)

#include <iostream>

#include <cstdlib>

#include <pthread.h>

#include <unistd.h>

using namespace std;

#define NUM\_THREADS 5

void \*wait(void \*t)

{

int i;

long tid;

tid = (long)t;

sleep(1);

cout << "Sleeping in thread " << endl;

cout << "Thread with id : " << tid << " ...exiting " << endl;

pthread\_exit(NULL);

}

int main ()

{

int rc;

int i;

pthread\_t threads[NUM\_THREADS];

pthread\_attr\_t attr;

void \*status;

// 初始化并设置线程为可连接的（joinable）

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);

for( i=0; i < NUM\_THREADS; i++ ){

cout << "main() : creating thread, " << i << endl;

rc = pthread\_create(&threads[i], NULL, wait, (void \*)i );

if (rc){

cout << "Error:unable to create thread," << rc << endl;

exit(-1);

}

}

// 删除属性，并等待其他线程

pthread\_attr\_destroy(&attr);

for( i=0; i < NUM\_THREADS; i++ ){

rc = pthread\_join(threads[i], &status);

if (rc){

cout << "Error:unable to join," << rc << endl;

exit(-1);

}

cout << "Main: completed thread id :" << i ;

cout << " exiting with status :" << status << endl;

}

cout << "Main: program exiting." << endl;

pthread\_exit(NULL);

}

## Web 浏览

为了更好地了解 CGI 的概念，让我们点击一个超链接，浏览一个特定的网页或 URL，看看会发生什么。

* 您的浏览器联系上 HTTP Web 服务器，并请求 URL，即文件名。
* Web 服务器将解析 URL，并查找文件名。如果找到请求的文件，Web 服务器会把文件发送回浏览器，否则发送一条错误消息，表明您请求了一个错误的文件。
* Web 浏览器从 Web 服务器获取响应，并根据接收到的响应来显示文件或错误消息。

然而，以这种方式搭建起来的 HTTP 服务器，不管何时请求目录中的某个文件，HTTP 服务器发送回来的不是该文件，而是以程序形式执行，并把执行产生的输出发送回浏览器显示出来。

公共网关接口（CGI），是使得应用程序（称为 CGI 程序或 CGI 脚本）能够与 Web 服务器以及客户端进行交互的标准协议。这些 CGI 程序可以用 Python、PERL、Shell、C 或 C++ 等进行编写。

C++ STL 教程

在前面的章节中，我们已经学习了 C++ 模板的概念。C++ STL（标准模板库）是一套功能强大的 C++ 模板类，提供了通用的模板类和函数，这些模板类和函数可以实现多种流行和常用的算法和数据结构，如向量、链表、队列、栈。

C++ 标准模板库的核心包括以下三个组件：

|  |  |
| --- | --- |
| **组件** | **描述** |
| 容器（Containers） | 容器是用来管理某一类对象的集合。C++ 提供了各种不同类型的容器，比如 deque、list、vector、map 等。 |
| 算法（Algorithms） | 算法作用于容器。它们提供了执行各种操作的方式，包括对容器内容执行初始化、排序、搜索和转换等操作。 |
| 迭代器（iterators） | 迭代器用于遍历对象集合的元素。这些集合可能是容器，也可能是容器的子集。 |

容器相关 的三个头文件：主要操作数据集合

vector 向量：数组类集合 可以自动处理大小