#include <iostream>

#include <vector>

#include <cstdlib>

#include <functional>

#include "./Programmer.h"

using namespace std;

void NewDeleteTest();

void DynamicCastTest();

void MemberPTest();

int main()

{

    NewDeleteTest();

    DynamicCastTest();

    MemberPTest();

    cout << "enter key" << endl;

    while (cin.get() != EOF) // ctrl + z

    {

    }

    return 0;

}

void NewDeleteTest()

{

    string \*sp = new string("1");

    delete sp;

    // 定位 new 表达式

    void \*mem = malloc(sizeof(string)); // 分配内存

    new (mem) string("abc");            // 在地址 mem 上构造对象

    cout << \*static\_cast<string \*>(mem) << endl;

}

// 当我们希望控制内存分配过程时，我们需要定义 operator delete （或者 operator delete[]）和 operator new （或者 operator new[]）函数

// operator delete （或者 operator delete[]）和 operator new （或者 operator new[]）具有8个重载版本，需要全部定义

// new 实际执行了三个步骤

// 1.调用 operator new （或者 operator new[]）分配内存

// 2.运行相应的构造函数构造对象

// 3.将对象分配给内存，返回指针

void \*operator new(size\_t size)

{

    cout << "new size:" << size << endl;

    // 使用 malloc 分配内存（#include <cstdlib>）

    if (void \*mem = malloc(size))

    {

        return mem;

    }

    else

    {

        throw bad\_alloc();

    }

}

// delete 实际执行了两个步骤

// 1.执行对象的析构函数

// 2.调用名为 operator delete（或者 operator delete[]）释放内存

void operator delete(void \*mem) noexcept

{

    cout << "delete:" << mem << endl;

    free(mem);

}

// 运行时类型识别（运行的时候识别）

// dynamic\_cast

void DynamicCastTest()

{

    FrontEnd frontEnd;

    Programmer \*programmerP = &frontEnd;

    // 使用 dynamic\_cast，Programmer 应至少有一个虚函数

    if (FrontEnd \*frontEndP = dynamic\_cast<FrontEnd \*>(programmerP))

    {

        cout << "programmerP is FrontEnd\*" << endl;

    }

}

// typeid，获取目标类型，#include <typeinfo>

// typeid的操作实在有限，只能用于判断类型是否相等

void TypeidTest()

{

    FrontEnd frontEnd;

    Programmer \*programmerP = &frontEnd;

    if (typeid(\*programmerP) == typeid(FrontEnd))

    {

        cout << "programmerP is FrontEnd\*" << endl;

    }

}

// 枚举

// 不限定作用域枚举（指定枚举成员可以不用加 color1）

// color1 a = red;

enum color1

{

    red,

    yellow

};

// 限定作用域枚举（指定枚举成员加 color2 作用域）

// color2 a = color2::red;

enum class color2

{

    red,

    yellow

};

// 成员指针

// 可以指向类非静态成员的指针

void MemberPTest()

{

    // 定义一个指向 Programmer 成员属性，属性类型是 string 的指针

    const string Programmer::\*name;

    // 将 name 指向 Programmer::Name

    // 如果 Programmer::Name 是 private，我们可以在 Programmer 声明一个静态函数，函数返回 &Programmer::Name 来获取 Programmer::Name 的地址

    name = &Programmer::Name;

    Programmer programmer1;

    programmer1.Name = "aaa";

    // 访问成员指针

    cout << programmer1.\*name << endl; // 输出 aaa

    // 成员函数指针，使用 auto 会更好

    std::string (Programmer::\*getName)() = &Programmer::GetName;

    // 访问成员函数

    cout << (programmer1.\*getName)() << endl; // 输出 aaa

    // 如下通过成员函数生成一个可调用对象

    // 成员函数的第一个参数是指向对象的指针，通过 string1.empty() 调用实际上是调用 empty(&string1)

    // 这也就是为什么如下的代码能通过

    function<bool(const string \*)> fcn = &string::empty;

    string s;

    cout << fcn(&s) << endl;

    ;

}

// 嵌套类

class TestClass1

{

public:

    // 嵌套类定义在一个类的内部，但嵌套类与外层类没有任何关系

    // 如果定义在 public 中外部可以访问该类成员

    // 如果定义在 protected 中，外层类，友元，派生可以访问

    // 如果定义在 private 中，外层类，友元可以访问

    class TestClass2

    {

    };

};

// 联合类型

// 组合几种类型的类型

// 每个时刻，只能有一个数据成员可以右值，其他数据成员都是未定义

// 如果有类类型，则必须有析构函数

union Token

{

    int Tokena;

    string Tokens;

    char Tokenc;

    ~Token(){

        Tokens.~basic\_string();

    };

};

void UnionTest(){

    // 定义，如果使用类去初始化，需要定义构造函数

    Token token = { 1 };

    token.Tokens = "aaa";

}

// 局部类，定义在函数内部的类

void FunClassTest(){

    class funclass1{};

    funclass1 fc1;

}