PyObject

题目描述

刷新

在Python中，类型是动态绑定的。而在C++中所有实例在使用前都需要先声明类型。 那么我们可以在C++中实现类似Python的灵活性吗?

我们尝试构建这样一个类 `PyObject` 。它可以存储任意类型的数据，并且可以在运行时动态改变数据的类型。

一个很有用的知识是类型转换操作符的重载。如下例中所示。

```
class A {
public:
    operator int() const { return 1; }
};
```

如果这个时候我们有一个 `A` 的实例 `a` ，那么我们可以这样使用它：

```
int b = a;
```

这样的操作会调用 `A` 中的 `operator int()` 方法，将 `a` 转换为 `int` 类型。 这可以帮助我们在实现 `PyObject` 的时候实现十分灵活的类型转换。

题目要求

给定 `main.cpp` 、 `Makefile` 不可修改，你需要实现 `PyObject.h` ，以完成下面的四个任务。文件可以在这里(/staticdata/problem/2156.hXCXyyYpniSsNGne.pub/v9MzMeyP4kEejwdR.download.zip/download.zip)下载。

任务一

`PyObject` 需要实现动态类型绑定，且可以随时获取其值（此处仅要求 `int` ， `double` , `char` ）。具体来讲，以下代码应当可以正常运行：

```
#include "PyObject.h"
#include <iostream>

int main() {
    PyObject p;
    p = 1;
    p = 'c';
    char c = p;
    std::cout << c << std::endl;
    p = 1.0;
    int u = 2;
    p = u;
    p = 1ll;
}
```

同时在获得赋值的时候应当输出 `"PyObject got a value"` ，下同。

任务二

我们不应当满足于基本的内建类型，我们还希望它更强大一点，可以存储任意类型的数据，比如一个其他的类的对象。对应的任务代码如下：

```
#include "PyObject.h"
#include <iostream>

class Test {
public:
    int data;
};

int main() {
    PyObject p;
    Test t;
    t.data = 5;
    p = t;
    std::cout << ((Test) p).data << std::endl;
}
```

任务三

对于基本类型和复杂类型，我们希望 `PyObject` 可以有不同的响应方式。

对于基本类型，我们希望 `PyObject` 在任何情况下复制原值，并且从此不受原值的干扰，即原值改变后 `PyObject` 保存的值不变。

而对于复杂类型，我们希望避免不必要的复制，所以此时 `PyObject` 仅保存其引用，此时原值的变化会导致 `PyObject` 保存的值也发生变化。对应的任务代码如下：

```
#include "PyObject.h"
#include <iostream>

class Test {
public:
    int data;
};

int main() {
    PyObject p;
    Test t;
    t.data = 5;
    p = t;
    std::cout << ((Test) p).data << std::endl;
    t.data = 6;
    std::cout << ((Test) p).data << std::endl;
    char c = 'c';
    p = c;
    c = 'd';
    std::cout << (char) p << std::endl;
}
```

任务四

更进一步地，我们希望 `PyObject` 对值负责。所谓负责，指的是 `PyObject` 需要确保在自己生命周期内且被赋其他值以前，该值一定可以访问。而在 `PyObject` 的生命周期结束或者被赋其他值以后，该值得到了析构。即可以认为 `PyObject` 获得了该值的**所有权**。

对于基本类型，我们希望 `PyObject` 在**任何情况**下复制原值，并且前后互相不干扰；所有权完全独立。

对于复杂类型，如果赋值方式是：

1. 左值： `PyObject` 不获得所有权，仅保存引用。
2. 右值： `PyObject` 获得所有权，对其**负责**。
3. `PyObject`：如果对方拥有所有权，所有权转移，旧 `PyObject` 丧失所有权，新 `PyObject` 获得所有权；如果对方是引用，则新 `PyObject` 获得对原对象的引用。
4. `PyObject *`：不获得所有权，保存指向原对象的引用。

对于 `PyObject` 向 `PyObject` 以外的类的对象赋值：

1. `T`： 复制， `PyObject` 的所有权不改变。
2. `T &`： 返回一份引用， `PyObject` 的所有权不改变。

除此之外， `PyObject` 在获得引用后需要输出 `"Borrowing"` ，在获得所有权后输出 `"Owning"` 。

样例输入：

```cpp
#include <iostream>
#include <vector>
#include "PyObject.h"

class Test {
private:
    std::vector<int> data;
    int id;
    static int count;
public:
    Test(): id(count) {
        std::cout << "Test " << count << " created" << std::endl;
        count++;
    }

    Test(const Test &t): id(count), data(t.data) {
        std::cout << "Test " << count << " created by reference" << std::endl;
        count++;
    }

    Test(Test &&t) noexcept : id(t.id), data(std::move(t.data)) {
        std::cout << "Test " << id << " moved by rvalue reference" << std::endl;
    }

    void func(int a) {
        std::cout << "Test " << id << " calling func" << std::endl;
        data.push_back(a);
        std::cout << "data size: " << data.size() << std::endl;
    }

    static int getCount() {
        return count;
    }

    ~Test() {
        std::cout << "Test " << id << " destroyed" << std::endl;
        count--;
    }
};

int Test::count = 0;

int main() {
    PyObject p;
    p = std::move(*(new Test));
    ((Test &) p).func(1);
    Test t2 = p;
    ((Test &) p).func(10);
    Test &t3 = p;
    t3.func(100);
    t2.func(1000);
    char c = 'c';
    p = c;
    std::cout << Test::getCount() << std::endl;
    std::cout << (char) p << std::endl;
    p = t2;
    PyObject p1;
    PyObject p2;
    PyObject p3;
    p1 = &p;
    ((Test &) p1).func(1);
    ((Test &) p).func(10);
    p2 = p1;
    ((Test &) p2).func(100);
    p3 = p;
    p = 1;
    ((Test &) p3).func(1000);
    p3 = 1.0;
    std::cout << Test::getCount() << std::endl;
    return 0;
}
```

样例输出：

```
Test 0 created
PyObject got a value
Test 0 moved by rvalue reference
Owning
Test 0 calling func with 1
data size: 1
Test 1 created by reference
Test 0 calling func with 10
data size: 2
Test 0 calling func with 100
data size: 3
Test 1 calling func with 1000
data size: 2
PyObject got a value
Test 0 destroyed
1
c
PyObject got a value
Borrowing
PyObject got a value
Borrowing
Test 1 calling func with 1
data size: 3
Test 1 calling func with 10
data size: 4
PyObject got a value
Borrowing
Test 1 calling func with 100
data size: 5
PyObject got a value
Borrowing
PyObject got a value
Test 1 calling func with 1000
data size: 6
PyObject got a value
1
Test 1 destroyed
```

本题**没有输入**，正确完成以上四个任务并输出答案即可通过测试。

请将你的文件打包成一个 zip 格式的压缩包并上传。注意：你的文件应该在压缩包的根目录下，而不是压缩包的一个子文件夹下，换而言之，解压你提交的压缩包后，应该直接得到一系列 cpp 文件、h 文件等代码文件，而不是一个包含它们的文件夹。评测时，OJ会将提供的文件贴入你的目录下进行编译并执行。

提示：本题实现过程中可能会用到以下函数：

1. `std::forward`：`std::forward` 是 C++ 标准库提供的完美转发（Perfect Forwarding） 工具，用于在模板编程中保持参数的原始值类别（即左值/右值属性）。在函数模板中，当参数被传递时，如果直接使用 T&& 接收参数，参数的值类别会在后续传递中丢失。而使用 `std::forward` 即可确保参数的左值/右值属性不变。
2. `typename std::remove_cv::type`：`typename std::remove_cv::type` 是一个类型转换工具，用于从给定类型中移除顶层的 `const` 和 `volatile` 限定符（在 C++14 以上版本替换为 `std::remove_cv_t`）。在模板编程中使用 `typename std::remove_cv<T>::type` 的功能为：如果 `T` 是 `const` 或 `volatile` 修饰的类型（如 `const int`、`volatile char`、`const volatile double`），则返回移除限定符后的类型（`int`、`char`、`double`）。如果 `T` 未被 `const` 或 `volatile` 修饰，则直接返回 `T` 本身。

语言和编译选项

#	名称	编译器	额外参数	代码长度限制
0	oop_custom	`make`		1048576 B

递交历史

#	状态	时间
335955	Accepted	2025-04-25 16:49:26
335953	Wrong Answer	2025-04-25 16:44:07
335947	Runtime Error	2025-04-25 16:01:05

1

当前没有提交权限！