plist: A Python-like List Library

陈仁泽 1700012774

简介

- plist是一个基于C++11标准的C++库，实现了一种类似python中list的泛型数据结构。
- 同一个plist可以容纳各种不同的类型，并支持对其容纳的不同元素的一些通用的操作，如输出、比较等（当然，如果一些类型不支持某些操作，会在执行操作时抛出异常）。
- 支持C++中的vector的所有操作；同时支持python中list的各种操作，包括但不限于index、count、sort、用加法进行列表连接、用乘法进行列表复制、列表切片等。
- 注：和python的list最大的不同是，plist的拷贝是深拷贝，而python中的list的拷贝是浅拷贝，也就是plist中存放的是值，而python的list存放的是引用。这是C++和python的语言层面的差异导致的。如果要实现类似python的list的浅拷贝的效果，可以在plist中存放`std::shared_ptr`。

使用示例

先声明一些必要的工具，用于之后的演示：

```
struct UserType {
    int x;
    friend std::ostream &operator<<(std::ostream &os, const UserType &ut) {
        return os << "User(" << ut.x << ")";
    }
};

template<typename T>
void println(const T &t) {
    std::cout << t << std::endl;
}
```

创建列表

```
crz::plist l1{1, 1.2, std::string("???"), UserType{1}}; // 用初始化列表构造
println(l1); // [1, 1.2, ???, User(1)]

crz::plist l2(3); // 构造含有3个空元素的列表
println(l2); // [None, None, None]

crz::plist l3(2, l1); // 构造含有2个l1的列表
println(l3); // [[1, 1.2, ???, User(1)], [1, 1.2, ???, User(1)]]

std::vector<int> vi{1, 2, 3};
crz::plist l4(vi.begin(), vi.end()); // 用迭代器构造列表
println(l4); // [1, 2, 3]

crz::plist l5 = l4; // 拷贝构造
println(l5); // [1, 2, 3]

crz::plist l6 = std::move(l5); // 移动构造
println(l5); // []
println(l6); // [1, 2, 3]
```

添加元素

```
crz::plist l;
println(l); // []

l.push_back(1);
println(l); // [1]

l.insert(l.begin(), "wow");
println(l); // [wow, 1]

l.append(UserType{2}); // 等同于push_back
println(l); // [wow, 1, User(2)]
```

移除元素

```
crz::plist l{1, 1.2, std::string("hello"), std::string("hello"), UserType{3}};
println(l); // [1, 1.2, hello, hello, User(3)]

l.erase(l.begin());
println(l); // [1.2, hello, hello, User(3)]

l.pop_back();
println(l); // [1.2, hello, hello]

l.remove(std::string("hello")); // 删除所有该元素
println(l); // [1.2]
```

列表连接和复制

```
crz::plist l{1, 1.2};
println(l); // [1, 1.2]

l += {"wow", UserType{3}};
println(l); // [1, 1.2, wow, User(3)]
println(crz::plist{"???", 2.2} + l); // [???, 2.2, 1, 1.2, wow, User(3)]
println(2 * l); // [1, 1.2, wow, User(3), 1, 1.2, wow, User(3)]
println(l * 2); // [1, 1.2, wow, User(3), 1, 1.2, wow, User(3)]

l.extend(l);
println(l); // [1, 1.2, wow, User(3), 1, 1.2, wow, User(3)]
```

列表内嵌

```
crz::plist l{"禁止套娃"};
for (int i = 0; i < 5; ++i)
    l = {"禁止", l};
println(l); // [禁止, [禁止, [禁止, [禁止, [禁止, [禁止套娃]]]]]]
```

数组索引与切片

```
crz::plist l{1, 1.2, "wow", std::string("???"), UserType{3}};
// l[2]
println(l[2]); // wow
// l[-2]
println(l[-2]); // ???
// l[1:3]
println(l[{1, 3}]); // [1.2, wow]
// l[1:-1]
println(l[{1, -1}]); // [1.2, wow, ???]
// l[:]
println(l[{{}, {}}]); // [1, 1.2, wow, ???, User(3)]
// l[-1:]
println(l[{-1, {}}]); // [User(3)]
// l[:1]
println(l[{{}, 1}]); // [1]
// l[::-1]
println(l[{{}, {}, -1}]); // [User(3), ???, wow, 1.2, 1]
// l[::2]
println(l[{{}, {}, 2}]); // [1, wow, User(3)]
// l[1:3:-1]
println(l[{1, 3, -1}]); // []
// l[3:1:1]
println(l[{3, 1, 1}]); // []
```

元素查找

```
crz::plist l{1, 2, std::string("???"), 2, std::string("!!!"), 2};
// 查找元素第一次出现的索引
println(l.index(std::string("???"))); // 2
println(l.index(15)); // -1
// 元素出现的次数
println(l.count(2)); // 3
println(l.count(std::string("..."))); // 0
```

列表反转

```
crz::plist l{"灵梦", "早苗", "魔理沙"};
println(l); // [灵梦, 早苗, 魔理沙]
l.reverse();
println(l); // [魔理沙, 早苗, 灵梦]
```

列表排序

```
crz::plist l{std::string("Reimu"), std::string("Marisa"), std::string("Sakuya"),
             std::string("Sanae")};
println(l); // [Reimu, Marisa, Sakuya, Sanae]
l.sort();
println(l); // [Marisa, Reimu, Sakuya, Sanae]
l.sort(true); // 逆序排序
println(l); // [Sanae, Sakuya, Reimu, Marisa]
l.sort(false, [](const std::string &s) {
    return std::make_pair(s.length(), s); // 先比长度，再比字典序
});
println(l); // [Reimu, Sanae, Marisa, Sakuya]

crz::plist il;
int len = l.size();
for (int i = 0; i < len; ++i)
    il.push_back(i); // il中保存l的索引
println(il); // [0, 1, 2, 3]
il.sort(false, [&](int i) { return l[i]; }); // 根据索引对应的l中的元素对索引排序
println(il); // [2, 0, 3, 1]
```

其他列表操作

```
crz::plist l{1, 2, 3, 4};
println(l.map([](int x) { return x + 1; })); // [2, 3, 4, 5]
println(l.map([](int x) { return std::to_string(x); })
         .map([](const std::string &s) { return s + "..."; })); // [1..., 2..., 3..., 4...]
l.for_each([](int &x) { x += 1; }).for_each([](int x) { println(x); });
println(l); // [2, 3, 4, 5]
println(l.map([](int x) { return x + 1; })
         .filter([](int x) { return x < 5; })); // [3, 4]
```

默认输出

```
struct A {
};
crz::plist l{A{}};
// 没有重载<<操作符，则输出其对象id。id结构为类型的typeinfo加上对象的地址。
println(l); // [Z19test_default_outputvE1A0xbf00b0]
println(l[0].id()); // Z19test_default_outputvE1A0xbf00b0
// 将l[0]容器内的值窃走
auto a = std::move(l[0]);
println(l); // [None]
println(l[0].id()); // None
```

提取元素（类型转换）

```
crz::plist l{1, 1.2, "???"};
println(l); // [1, 1.2, ???]
// 必须通过显示转换对容器内的值进行直接读写
// 因为虽然对象行为可以在运行期动态确定（比如<<操作符的行为），但类型信息必须在编译期静态确定
l[0].cast<int &>() = 2;
println(l); // [2, 1.2, ???]
const char *s = l.back().cast<const char *>();
println(s); // ???
```

元素类型判断

```
crz::plist pl;
// 随机生成列表元素，无法在编译期知道列表某个位置的元素的确切的底层类型
for (int i = 0; i < 10; ++i) {
    if (std::rand() % 2) {
        pl.push_back(233);
    } else {
        pl.push_back(std::string("???"));
    }
}
for (auto &x: pl) {
    // 用type()函数比较类型的typeinfo（C++的RTTI机制）
    if (x.type() == typeid(int)) {
        println(x.cast<int>());
    } else if (x.type() == typeid(std::string)) {
        println(x.cast<const std::string &>());
    } else {
        throw std::runtime_error("???");
    }
}
```

异常行为

```
struct A {
};
{
    crz::plist l{A{}, A{}, A{}};
    try {
        l.count(A{}); // A未定义==运算符，无法进行查找
    } catch (crz::bad_comparison &e) {
        println(e.what()); // bad comparison: Z19test_some_exceptionvE1A ==
Z19test_some_exceptionvE1A
    }
}
{
    crz::plist l{1, 2, std::string("???")};
    try {
        l.sort(); // 导致不同类型元素比较
    } catch (crz::bad_comparison &e) {
        println(e.what()); // bad comparison:
NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE < i
    }
    try {
        l.sort(false, [](int i) { return i; }); // 会将std::string转换为int
    } catch (crz::bad_pcell_cast &e) {
        println(e.what()); // bad pcell cast: from
NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE to i
    }
}
{
    crz::plist l(2);
    l[0] = 1;
    println(l); // [1, None]
    try {
        auto s = l[0].cast<std::string>(); // 将int转换为std::string
        println(s);
    } catch (crz::bad_pcell_cast &e) {
        println(e.what()); // bad pcell cast: from i to
NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
    }
    try {
        auto i = l[1].cast<int>(); // 将空值转换为int
        println(i);
    } catch (crz::bad_pcell_access &e) {
        println(e.what()); // bad pcell access
    }
}
```

API

pcell

```
class pcell {

    // 对象持有者的基类，定义了一些虚函数用于运行时的动态操作
    struct holder;

    // 真正的对象持有者
    template<typename T>
    struct holder_impl : public holder;

    template<typename T>
    using deref_type = typename std::remove_reference<T>::type;

    template<typename T>
    using decay_type = typename std::decay<T>::type;

    template<typename T>
    using base_type = decay_type<deref_type<T>>;

public:
    pcell() = default;

    // 隐式构造函数，可以进行其他类型到pcell的隐式转换
    template<typename T, typename B = base_type<T>,
            typename = typename std::enable_if<!std::is_same<B, pcell>::value>::type>
    pcell(T &&t);

    // 拷贝构造函数，调用clone函数来动态地拷贝值
    pcell(const pcell &rhs);

    // 移动构造函数，直接交换指针
    pcell(pcell &&rhs) noexcept;

    ~pcell();

    // 拷贝运算符。rhs构造时会根据情况调用拷贝构造和或移动构造，因此只要实现一个拷贝运算即可。
    pcell &operator=(pcell rhs);

    // 对于非pcell对象的拷贝
    template<typename T, typename B = base_type<T>,
            typename = typename std::enable_if<!std::is_same<B, pcell>::value>::type>
    pcell &operator=(T &&t);

    // 清除容器
    void reset() noexcept

    // 交换
    void swap(pcell &rhs) noexcept;

    // 判断容器内是否包含着对象
    bool has_value() const noexcept;
```

```
// 返回容器包含对象的类型的type_info
const std::type_info &type() const noexcept;

// 返回保存的对象的id
std::string id() const;

// 返回容器内的对象的值
// 目标的底层类型不为pcell
template<typename T, typename B = base_type<T>, typename = void,
        typename = typename std::enable_if<
                !std::is_same<B, pcell>::value
        >::type>
T cast();

// 目标的底层类型不为pcell且目标不为非常量引用
template<typename T, typename B = base_type<T>, typename = void,
        typename = typename std::enable_if<
                !std::is_same<B, pcell>::value &&
                (!std::is_reference<T>::value ||
                 std::is_const<deref_type<T>>::value)
        >::type>
T cast() const;

// 目标的底层类型为pcell
template<typename T, typename B = base_type<T>,
        typename = typename std::enable_if<
                std::is_same<B, pcell>::value
        >::type>
T cast();

// 目标的底层类型为pcell且目标不为非常量引用
template<typename T, typename B = base_type<T>,
        typename = typename std::enable_if<
                std::is_same<B, pcell>::value &&
                (!std::is_reference<T>::value ||
                 std::is_const<deref_type<T>>::value)
        >::type>
T cast() const;

// 判断容器内的对象是否为给定类型（其实可以直接用type_info判断）
template<typename T>
bool is() const;

// 显式类型转换函数
template<typename T>
explicit operator T() const;

// 重载<<运算；如果容器内为空，则输出"None"
friend std::ostream &operator<<(std::ostream &os, const pcell &cell);

// 重载==和!=；当前仅当两者都为空或都含有相同的值视作相等，其他情况视作不相等（并不会抛出异常）
// 不过，如果两者类型相同，但其类型没有重载==，则会抛出异常
```

```
    friend bool operator==(const pcell &a, const pcell &b);
    friend bool operator!=(const pcell &a, const pcell &b);

    // 重载< > <= >=；注意如果其中一方为空或者两者类型不相同则会抛出异常
    // 两者类型相同但没有重载相应运算符，也会抛出异常
    friend bool operator<(const pcell &a, const pcell &b);
    friend bool operator>(const pcell &a, const pcell &b);
    friend bool operator<=(const pcell &a, const pcell &b);
    friend bool operator>=(const pcell &a, const pcell &b);
private:

    template<typename T, typename B = base_type<T>>
    holder_impl<B> *get_holder_impl() const;
};
```

plist

```
// python-like list，继承自vector以复用其大部分的函数
class plist : public std::vector<pcell> {
    class pslice;
public:
    using std::vector<pcell>::vector;

    // 直接索引访问，支持负数索引。返回对应pcell的引用
    pcell &operator[](int i);
    const pcell &operator[](int i) const;

    // 切片索引访问，返回一个新的plist
    plist operator[](pslice sl) const;

    // 列表连接
    friend plist operator+(const plist &a, const plist &b);
    plist &operator+=(const plist &pl);

    // 列表拷贝
    friend plist operator*(size_t time, const plist &pl);
    friend plist operator*(const plist &pl, size_t time);
    plist &operator*=(size_t time);

    // 列表输出
    friend std::ostream &operator<<(std::ostream &os, const plist &pl);

    // 列表转换为字符串
    explicit operator std::string() const;

    // python API
    size_t count(const pcell &pc) const;
    int index(const pcell &pc) const;
    void append(const pcell &pc);
    void extend(const plist &pl);
    void remove(const pcell &pc);
    void reverse();
```

```
    // 排序函数。其中key是一个一元函数（类型为A => B），和python中的一样。rvs代表是否逆序排序
    template<typename F = std::function<const pcell &(const pcell &)>>
    void sort(bool rvs = false, F key = [](const pcell &x) -> const pcell & { return x; });

    // 其他常用的列表操作
    template<typename F>
    plist &for_each(F trans);

    template<typename F>
    plist map(F mapping);

    template<typename F>
    plist filter(F pred);
};
```

实现方法简介

先是基本的实现思路

pcell：类型安全（大概吧）的泛型容器

- 泛型对象的值的持有者：
  - `holder`是抽象基类，定义了一系列访问底层对象的虚方法（比如拷贝、比较、输出等）。
  - `holder_impl<T>`是继承自`holder`的模板类，保存着类型为T的值，实现了基类定义的虚方法。
- pcell中保存着`holder`类的指针，利用其定义的虚方法来动态地执行特定的行为（拷贝、比较、输出等）。利用`dynamic_cast`来进行动态的类型转换。

实际上pcell的实现思路和C++17的`std::any`类似。

plist：保存pcell的vector

- plist继承自`std::vector<pcell>`，这样可以复用大量的标准库代码。
- 在此基础上实现一些其他功能。
- 注意：类的构造函数需要显式地复用：

```
using std::vector<pcell>::vector;
```

按照以上思路基本可以实现大部分的功能，还有一部分功能的实现细节较为复杂，而且基本都涉及到模板元编程（TMP），故单独列出。

根据对象行为进行静态派发

同一个列表可以装不同类型的元素，每种类型支持的操作集合可能不同，这样会对实现造成一定阻碍，因为pcell支持的操作集合肯定是这些类型的操作集合的并集而不是交集（至少不完全是交集，毕竟所有可能类型的操作集合交集基本等于空集），那么就必须得为不支持某些操作的类型定义一些默认行为（比如输出默认值，或者抛出异常等），并且该行为的派发还得是静态的，否则会在编译时就出现问题。

下面以比较运算符为例来解释这种问题并说明解决方法。

- 某些类型没有重载小于运算符，就不能对列表进行默认排序。
- 但为了支持默认的排序功能，pcell肯定要重载小于运算符，该小于运算符调用holder的虚函数来动态派发给底层的holder_impl进行比较操作。
- 而对于`holder_impl<T>`的小于运算的实现肯定不能直接用类型T的比较操作实现，因为类型T不一定有定义比较操作。直接用T的比较操作会导致只有支持比较操作的T才能用来实例化`holder_impl<T>`。
- 因此我们要通过类型T的行为对`holder_impl<T>`的小于操作的实现进行派发：
  - 对于支持小于运算的T，直接实现为对象的小于比较。
  - 否则，实现为抛出一个异常。
- 该派发必须是静态的，并且能根据对象的行为进行判断，也就是不能直接用`if`等条件语句判断，也不方便用C++17的`constexpr if`来静态判断（在编译期反射功能出来前，用C++现有的技术很难将对象对某种行为的支持编码为if的判断条件）。
- 解决方法是利用模板偏特化，再结合`decltype`、`std::declval`、`std::void_t`（`std::void_t`是C++17的内容，不过实现起来非常简单）等TMP的常客。具体方法参见代码。可以参考cppreference关于`std::void_t`的示例来理解解决方法的思路。

函数参数类型的获取

主要是实现sort等函数对于形如(A => B)的用户自定义的映射函数的支持时，需要根据用户传入的函数判断函数的参数类型，将pcell强制转换到该类型。

其实可以让pcell重载形如`template<typename T> operator T() const;`的隐式类型转换函数，这样可以不用显式获取函数的参数类型进行显式类型转换。不过这种方法有一些缺陷：

- 如果函数的参数类型为引用（比如`const std::string&`），隐式转换会脱去引用修饰符和底层const（也就是对于`const std::string&`，pcell会被隐式转换成`std::string`），这样会导致大量不必要的拷贝开销。
- 上面一种方案可以用重载多个隐式转换函数解决（如额外重载`template<typename T> operator T&();` `template<typename T> operator const T&() const;`）。但是不同编译器的对此的处理规则不尽相同，用g++没问题，但用clang++时会出现问题，例如：
  - 如果目标类型是`int`，clang++认为三种类型转换函数都可行，从而报错。而g++只会匹配`operator T() const`。
- 允许pcell隐式转换为其他类型很容易引发歧义，在实践中并不是个好选择。

具体怎么获取参数类型，基本思路是利用函数萃取传入的函数（普通函数或函数对象）的参数类型列表，再利用一些TMP技巧判断传入的函数是普通的函数还是函数对象，以及从参数类型列表中抽取给定位置的参数类型。详情请直接参见代码。关于萃取函数对象的参数类型列表的方法可以参考stackoverflow上的一个问题。

除此之外，还有其他一些琐碎的细节，比如“隐式构造pcell时需要褪去传入参数类型T的引用修饰和const修饰获得类型B，构造的是`holder_impl<B>`而不是`holder_impl<T>`，但转发给`holder_impl<B>`构造函数的参数仍然必须是类型T”，“对于需要进一步转发的参数需要用右值引用接收”，诸如此类，不再赘述。详情请直接参见代码。