小明的PVector

题目描述

刷新

注意：请合理安排考试时间，可以选择实现部分任务以获得部分分数

小明想最近了解到了可持久化数据结构（persistent data structure）的概念，因此想仿照着设计一个可持久化的vector容器，称为PVector。

Subtask1

简单来说，可持久化数据结构支持在修改之后仍然保留原来的历史版本。例如，任意修改PVector的操作将返回一个全新的PVector对象，而原有PVector不变。

```
// Subtask 1
PVector<int> a1;                  // Create an empty PVector, a1 = []
cout << a1 << endl;               // Output: []
auto a2 = a1.push_back(1);        // a2 = [1]
cout << a2 << endl;               // Output: [1]
int index = 0, value = 2;
auto a3 = a2.set(index, value);   // modify an element, a3 = [2]
cout << a3 << endl;               // Output: [2]
auto a4 = a2.push_back(3);        // a4 = [1, 3]
cout << a4 << endl;               // Output: [1, 3]
cout << a4[1] << endl;            // Output: 3
auto a5 = a4.push_back(5);        // a5 = [1, 3, 5]
cout << a5 << endl;               // Output: [1, 3, 5]
```

进一步，小明希望这些操作的内存与时间开销不要太大，**即对于每一个 `push_back` 或 `set` 操作，应该只记录修改，不能将整个PVector都复制一遍**。 为了测试他的代码，他使用自定义类型Point构造了 `PVector<Point>` 。每个Point包含x, y两个坐标，并且重载了输出流运算符（具体可以下载代码查看）。同时，对于Point类型，小明会检查所有Point对象的构造、析构次数。你需要满足：

- Point对象的构造次数不应该超过给定的参考值，在subtask1中是 `push_back` 和 `set` 的操作数量的4倍。
- Point对象的析构次数应该与构造次数相等，避免内存泄露。

当然，若你满足不了上述条件，我们也会有部分分，具体可以查看题目最后的评分标准。

Subtask2

在Subtask1的基础上，小明还想增加一个撤销功能。对于一个PVector，可以通过 `undo` 函数获得上一次修改前的版本。注意 `undo` 可以多次使用。如果已经是最初的，操作无效并输出 `cannot undo` 。（操作无效时请返回当前对象，不做任何操作。）

特别的， `undo` 操作不应该消耗构造次数，即要求：

- Point对象的构造次数不应该超过给定的参考值，在subtask2中是 `push_back` 和 `set` 的操作数量的4倍。

（如果你对以下代码的输出有疑惑，我们提供一张图片展现了测试代码中各个对象的关系，请在最后的链接中下载）

```
// Subtask 2
// Codes after Subtask 1
auto b4 = a5.undo();              // b4 = [1, 3]
auto b2 = b4.undo();              // b3 = [1]
auto b1 = b2.undo();              // b1 = []
b1.undo();                        // Output: cannot undo
auto b6 = b2.push_back(0);        // b6 = [1, 0]
```

Subtask3

最后，小明想添加一个终极功能：合并两个PVector的修改。如果对于同一个PVector `pv_origin` ，做出了不同的修改后分别得到了 `pv_a` 和 `pv_b` 。那么使用 `pv_merge = pv_a.update(pv_b)` 将得到综合两个修改的结果。具体来说：

- 从 `pv_origin` 到 `pv_b` 的所有操作将插入到 `pv_a` 的修改之后。这次 `update` 被认为是一次操作，即通过 `pv_merge.undo()` 可以回到 `pv_a` 的状态。特殊地，允许 `pv_origin` 是 `pv_a` 或者 `pv_b` 本身。
- 如果 `pv_a` 和 `pv_b` 不是从同一个PVector修改而来，那么操作无效，应输出 `cannot update: no origin found` 。（操作无效时请返回当前对象，不做任何操作。）
- 如果 `pv_a` 和 `pv_b` 的修改有冲突：即都在队尾插入了元素，或者都修改了同一个元素；那么操作无效，应输出 `cannot update: conflicts found` 。（操作无效时请返回当前对象，不做任何操作。）

注意，在判断修改是否有冲突时，可能有两种需要考虑的情况：

- 被合并的对象可能会经历过 `undo` 操作，此时被撤销的操作不应考虑在冲突里。举例来说：
  - 从 `pv_origin` 做出了修改A、B得到了 `pv_a` 。（其中A、B可能是 `push_back` , `set` , `update` 操作）。
  - 从 `pv_origin` 做出了修改D、E、F，再经历一次 `undo` 得到了 `pv_b` 。（其中D、E、F可能是 `push_back` , `set` , `update` 操作）。

特别的， update 操作**最多**消耗的构造次数应该和 update 涉及操作数量有关，即要求：

- Point对象的构造次数不应该超过给定的参考值，在subtask3中是 (push_back 和 set 的操作数量 + update 所涉及操作数量) * 4。
- 其中， update 所涉及的操作数量是指，从 pv_origin 到 pv_b 、 pv_a 之间的所有操作数量之和（操作中 push_back , set , update 都只计一次）。

（如果你对以下代码的输出有疑惑，我们提供一张图片展现了测试代码中各个对象的关系，请在最后的链接中下载。）

```
// Subtask 3
// Codes after Subtask 2
auto c7 = a3.update(a5);        // c7 = [2, 3, 5]
auto c3 = c7.undo();            // c3 = [2]
auto c8 = a5.update(a3);        // c8 = [2, 3, 5]
auto c5 = c8.undo();            // c5 = [1, 3, 5]
auto c9 = a1.update(c8);        // c9 = [2, 3, 5]
auto c10 = c8.update(a1);       // c10 = [2, 3, 5]
auto c11 = c10.undo();          // c11 = [2, 3, 5]
auto c12 = a5.update(a3);       // c12 = [2, 3, 5]

PVector<int> other;
a3.update(other);                 // Output: cannot update: no origin found
a4.update(b6);                    // Output: cannot update: conflicts found
c10.update(b6);                   // Output: cannot update: conflicts found
a2.update(c9);                    // Output: cannot update: conflicts found
c12.update(c8);                   // Output: cannot update: conflicts found
```

提示

- 小明已经实现了部分代码，可以在下方链接中下载，你可以**基于小明的代码修改，也可以完全自己来实现**。他的实现思路如下：
  - 小明完全不会使用vector容器，而是使用装饰器模式来避免复制原有对象。具体来说，每层装饰器只记录修改的部分，通过重写覆盖 T get(int index) 函数来修改元素访问时返回的内容。
  - 由于PVector是一个对象，因此无法直接使用虚函数。他在PVector内部维护一个指针，指向真正的容器对象Data。
  - 传参和返回值中使用const T&有助于减少构造函数调用次数(但注意不能返回局部变量的引用)。
  - 小明目前还没考虑内存泄露问题，但预计可以使用智能指针解决。
- 为了简单考虑，我们对题目做出以下限制：
  - 你只用考虑 PVector<int> 和 PVector<Point> 。
  - **所有操作保证不会越界**。
- 本题的样例测试代码在 main_int.cpp ， main_point.cpp 中。我们提供 main_int.cpp 、 main_point.cpp 、 point.h 和 Makefile 。
- 特殊地， Makefile 支持测试部分subtask。如果你只想测试 subtask1 ，可以使用 make subtask1 命令。
- **为了辅助你的理解，我们提供一张图片展现了测试代码中各个对象的关系，请在以下链接中下载**。
- 链接中我们提供了完整的样例输出。
- 文件下载地址：下载链接 (/staticdata/1810.Vh2tXL5rwpWc3Iar.pub/8NQvHjI31bgLg0DP.%E5%BD%92%E6%A1%A3.zip/%E5%BD%92%E6%A1%A3.zip)。

提交格式

你只需提交 pvector.h 。我们会将你提交的文件和我们预先设置好的 main_int.cpp ， main_point.cpp 、 point.h 、 Makefile 一起编译运行。

评分标准

我们共有3个subtask：

- SUBTASK1（25分）：你需要实现 push_back , set , 流输出, 下标访问操作。保证所有 PVector 长度不超过100，操作不超过100。
- SUBTASK2（25分）：在SUBTASK1基础上，你需要实现 undo 操作。保证所有 PVector 长度不超过100，操作数量不超过100。
- SUBTASK3（50分）：在SUBTASK2基础上，你需要实现 update 操作。保证所有 PVector 长度不超过100，操作数量不超过100。

每个subtask中会有两个样例测试点，即下发的 main_int.cpp , main_point.cpp 。另外也有2个隐藏测试点，会相应的更改样例代码以及 point.h 代码进行测试。一般来说，如果正确实现了题目要求，**设计符合复杂度要求**，并能通过样例测试点（而不是通过某种方法直接输出标准答案），也应该能够通过隐藏测试点。（**若你认为存在问题，请及时联系监考老师。**）每个subtask分数分为4挡：

- 能通过 main_int.cpp 以及只包含int的隐藏测试点，获得该subtask 25%分数。
- 满足以上条件，并能通过 main_point.cpp 以及包含Point的隐藏测试点，获得该subtask 50%分数。
- 满足以上条件，并且构造数量符合要求，即不超过(push_back 和 set 的操作数量 + update 所涉及操作数量) * 4，获得该subtask 75%分数。
- 满足以上条件，并且没有内存泄露，获得该subtask 100%分数。

注意你不用同时通过3个子任务再提交，我们会将每一个子任务的代码拆开，分别编译。

考试100%为OJ评分。

时间限制：2s 内存限制：256M

- 包含Point的隐藏测试点。

其中：

- 前2个测试点均正确时将获得25% subtask得分。
- 后2个测试点最低得分为25时（代表构造函数次数超过限制），获得50% subtask得分。
- 后2个测试点最低得分为50时（代表出现内存泄露），获得75% subtask得分。
- 后2个测试点均正确时，获得100% subtask得分。

（注：由于OJ迁移的缘故，具体计分方式可能与上述描述不一致，即分数可能不准确，但各项评测均已准确实现）

Subtask中的每个测试点得分之和不等于该subtask总分属于正常现象。

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
表中没有数据		

递交答案

语言和编译选项 oop_custom

```
1
```

提交

文件请拖入编辑器中，或 上传文件