preinte.hpp 使用指南

（适用版本 v1.0.0.88）

Lukaß Zhang 编写

2023/8/28

一、基本使用

1. 格式控制

结构体 preinte::print_format 是 preinte 打印时必要的格式参数，其用于指示打印目标对象时的宽度、对齐和小数位数，其部分成员如下：

```
struct print_format {
    const char* name;         // 对象类型名
    size_t      width;        // 制表时的单元格宽度
    size_t      length;       // 制表时的表格列数
    size_t      digits;       // 保留的小数位数
    ALIGNMENT   alignment;    // 输出对齐格式
}
```

其含参构造函数如下：

```
print_format (
    const char*      _Name,                    // 对象类型名
    const size_t&    _Width,                   // 制表时的单元格宽度
    const size_t&    _Length,                  // 制表时的表格列数
    const size_t&    _Digits = 0,              // 保留的小数位数
    const ALIGNMENT& _Align  = ALIGNMENT::LEFT // 输出对齐格式
)
```

2. 打印内置类型

由于每次打印需要传入格式控制参数，十分繁琐。因此 preinte 为以下类型内置了格式参数：

类型	width	length	digits	alignment
int	7	10	0	RIGHT
unsigned int	7	10	0	RIGHT
short	6	10	0	RIGHT
unsigned short	6	10	0	RIGHT
float	9	10	2	RIGHT
double	9	10	2	RIGHT
long	10	10	0	RIGHT
unsigned long	10	10	0	RIGHT
long long	10	10	0	RIGHT
unsigned long long	10	10	0	RIGHT
long double	10	10	2	RIGHT
bool	3	25	0	RIGHT
char*	80	1	0	LEFT
std::string	80	1	0	LEFT

由于逐一设置了相对应的 preinte::print_format，打印时不必单独传参。我们将上表所列类型称为内置类型。如要改变内置类型的打印格式，可以调用下述函数：

```
// 设置内置类型的单元格宽度
template <typename _Ty>
is_built_in_t<_Ty> print_tool::set_width (
    const size_t& _Width  // 新宽度
)

// 设置内置类型的单元格列数
template <typename _Ty>
is_built_in_t<_Ty> print_tool::set_length (
```

```
    const size_t& _Length // 新列数
)

// 设置内置类型的打印小数位数
template <typename _Ty>
is_built_in_t<_Ty> print_tool::set_digits (
    const size_t& _Digit  // 新小数位数
)

// 设置内置类型的打印对齐位置
template <typename _Ty>
is_built_in_t<_Ty> print_tool::set_alignment (
    const print_format::ALIGNMENT& _Align   // 新对齐格式
)
```

打印内置类型时，仅需要调用 preinte::print 函数，传入必要参数即可：

1）打印一般值：

```
template <typename _Ty>      // _Ty 为内置类型
is_built_in_t<_Ty> print (
    const _Ty&  _Val,        // 对象
    const char* _Name        // 对象名称
)
```

2）打印一维数组

```
template <typename _Ty, size_t _Len>    // _Ty 为内置类型
is_built_in_t<_Ty> print (
    const _Ty   (&_Arr)[_Len],          // 数组指针
    const char* _Name                   // 数组名称
)

template <typename _Ty>                 // _Ty 为内置类型
is_built_in_t<_Ty> print (
    _Ty*  const   _Ptr,                 // 数组指针
    const size_t& _Len,                 // 数组长度
    const char*   _Name                 // 数组名称
)
```

3）打印二维数组

```
template <typename _Ty, size_t _Row, size_t _Col>  // _Ty 为内置类型
is_built_in_t<_Ty> print (
    const _Ty   (&_Arr)[_Row][_Col],               // 数组指针
    const char* _Name                              // 数组名称
)

template <typename _Ty>                            // _Ty 为内置类型
is_built_in_t<_Ty> print (
    _Ty** const   _Ptr,                            // 数组指针
    const size_t& _Row,                            // 数组的行数
    const size_t& _Col,                            // 数组的列数
```

```
    const char*   _Name                                 // 数组名称
)
```

4）打印 C 风格字符串

```
void print (
    char* const _Ptr,          // 字符串指针
    const char* _Name          // 字符串名称
)
```

5）打印 C 风格字符串数组

```
template <size_t _Len_Arr, size_t _Len_Str>     // _Ty 为内置类型
void print (
    const char  (&_Arr)[_Len_Arr][_Len_Str],    // 数组指针
    const char* _Name                           // 数组名称
)
```

打印包含内置类型的 STL 容器时，也仅需调用 preinte::print 函数。其中支持的 STL 容器在下表列出，其常用的函数重载使用请参见。

std::vector	std::deque	std::set	std::unordered_set
std::multiset	std::unordered_multiset	std::list	std::array
std::stack	std::queue	std::priority_queue	std::map
std::unordered_map	std::multimap	std::unordered_multimap	

【示例 1-1】修改内置类型的格式并打印内置类型的值：

```
<<<
#include "preinte.hpp"

int main() {
    // 打印一般值
    int d = 233;
    preinte::print(d, "d");

    double lf = 3.1415926;
    preinte::print(lf, "lf");
    // 将 double 的打印时小数位数设为 3
    preinte::print_tool::set_digits<double>(4);
    preinte::print(lf, "lf");

    // 打印在栈上的 int 数组
    int arr[5] = {-1, 0, 1, 2, 3};
    preinte::print(arr, "arr");

    // 打印在堆上的 short 数组
    short* shorts = new short[5] {5, 6, 7, 8, 9};
    preinte::print(shorts, 5, "shorts");
    delete[] shorts;

    // 打印 int 二维数组
    int ints[4][6] = {
        {1, 2, 3, 4, 5, 6},
```

```
        {7, 8, 9, 10, 11, 12},
        {13, 14, 15, 16, 17, 18},
        {19, 20, 21, 22, 23, 24}
    };
    preinte::print(ints, "ints");

    // 打印C风格字符串
    char cs[] = "zyh is a beautiful girl.";
    preinte::print(cs, "s");

    // 打印std::string
    std::string str = "I'm isolated.";
    preinte::print(str, "str");

    // 打印C风格字符串数组
    char csarr[3][32] = {0};
    sprintf(csarr[0], " - i love you.");
    sprintf(csarr[1], " - 6.");
    sprintf(csarr[2], " - okay, i'll remake.");
    preinte::print(csarr, "csarr");
}


>>>
@0x63fdec int d = 233
@0x63fde0 double lf = 3.14
@0x63fde0 double lf = 3.1416
@0x63fdc0 int arr[5]:
```

0 - 4	-1	0	1	2	3

```
@0x751930 short shorts[5]:
```

0 - 4	5	6	7	8	9

```
@0x63fd60 int ints[4][6]:
```

\	0	1	2	3	4	5
0	1	2	3	4	5	6
1	7	8	9	10	11	12
2	13	14	15	16	17	18
3	19	20	21	22	23	24

```
@0x63fd40 char* s = "zyh is a beautiful girl."
@0x63fd20 std::string str = "I'm isolated."
@0x63fcc0 char csarr[3][32]:
```

0	- i love you.
1	- 6.
2	- okay, i'll remake.

【示例 1-2】修改内置类型的格式并打印包含内置类型的 STL 容器:

```
<<<
#include "preinte.hpp"

int main() {
    // 将 double 的打印时小数位数设为 3
    preinte::print_tool::set_digits<double>(3);
    // 将 double 的打印格式设为左对齐
    preinte::print_tool::set_alignment<double>(preinte::print_format::ALIGNMENT::LEFT);
    // 将 double 的打表宽度设为 5
    preinte::print_tool::set_width<double>(6);
    // 打印 std::vector
    std::vector<double> vec_lf = {1.25, 2.829, 3.105, 4.2867, 5.212, 6.151, 7.037, 8.0445,
9.5651, 10.232};
    preinte::print(vec_lf, "vec_lf");

    // 将 int 的打表列数设为 10
    preinte::print_tool::set_length<int>(10);
    // 将 int 的打表宽度设为 5
    preinte::print_tool::set_width<int>(5);
    // 打印 std::array
    std::array<int, 20> arr_d = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20};
    preinte::print(arr_d, "arr_d");

    // 打印 std::set
    std::set<long, std::greater<>> set_ld = {25673, 98524, -12464, 43567};
    preinte::print(set_ld, "set_ld");

    // 将 std::string 的打表宽度设为 16
    preinte::print_tool::set_width<std::string>(16);
    // 打印 std::map
    std::map<int, std::string> map = {
        {520, " I love you."},
        {6, " Go away."},
        {7535, " Genshin Impact"}
    };
    preinte::print(map, "map");

    // 打印 std::stack
    std::stack<float> stack;
    stack.push(2.34f);
    stack.push(-5.92f);
    stack.push(4.59f);
    preinte::print(stack, "stack");
```

```
    // 打印 std::priority_queue
    std::priority_queue<int> priority_queue;
    priority_queue.push(-9);
    priority_queue.push(14);
    preinte::print(priority_queue, "priority_queue");
}


>>>
@0x67fcf0 std::vector<double> vec_lf[.size() = 10]:
```

0 - 9	1.250	2.829	3.105	4.287	5.212	6.151	7.037	8.044	9.565	10.232

```
@0x67fca0 std::array<int> arr_d[.size() = 20]:
```

0 - 9	1	2	3	4	5	6	7	8	9	10
10-19	11	12	13	14	15	16	17	18	19	20

```
@0x67fc70 std::set<long> set_ld[.size() = 4]:
```

0 - 3	98524	43567	25673	-12464

```
@0x67fc40 std::map<int, STD::string> map[.size() = 3]:
```

\	key	value
1	6	Go away.
2	520	I love you.
3	7535	Genshin Impact

```
@0x67fbf0 std::stack<float> stack[.size() = 3]:
```

top	4.59	...

```
@0x67fbd0 std::priority_queue<int> priority_queue[.size() = 2]:
```

top	14	...

3. 打印自定义类型

preinte 也可以打印自定义类型。preinte 会对该类型进行可打印校验，即其必须实现对 std::ostream 的 operator<<重载，示例如下：

```
_Ty& operator<<(std::ostream& _Cout, const _Ty& _Val) {
    // 打印的实现
    return _Cout;
}
```

其中，请保证打印的实现中**仅使用一次** `std::cout << ...`，否则可能会引发制表混乱等问题。为此，preinte 附带了用于方便字符串拼接的函数 preinte::str_tool::concat，其使用说明详见。

由于没有内置自定义类型的格式参数，因此在调用 preinte::print 时还需用户手动传入格式参数：

1）打印一般值：

```
template <typename _Ty>
is_printable_t<_Ty> print (
    const _Ty&          _Val,       // 对象
```

```
    const print_format& _Format,     // 打印格式
    const char*         _Name        // 对象名称
)
```

2）打印一维数组：

```
template <typename _Ty, size_t _Len>
is_built_in_t<_Ty> print (
    const _Ty           (&_Arr)[_Len],   // 数组指针
    const print_format& _Format,         // 打印格式
    const char*         _Name            // 数组名称
)
template <typename _Ty>
is_printable_t<_Ty> print (
    _Ty* const          _Ptr,            // 数组指针
    const print_format& _Format,         // 打印格式
    const size_t&       _Len,            // 数组长度
    const char*         _Name            // 对象名称
)
```

3）打印二维数组：

```
template <typename _Ty, size_t _Row, size_t _Col>
is_built_in_t<_Ty> print (
    const _Ty           (&_Arr)[_Row][_Col],    // 数组指针
    const print_format& _Format,                // 打印格式
    const char*         _Name                   // 数组名称
)

template <typename _Ty>
is_printable_t<_Ty> print (
    _Ty** const         _Ptr,                   // 数组指针
    const print_format& _Format,                // 打印格式
    const size_t&       _Row,                   // 数组的行数
    const size_t&       _Col,                   // 数组的列数
    const char*         _Name                   // 对象名称
)
```

打印 STL 容器时另需传入格式控制参数，具体用法参见示例 1-3。

【示例 1-3】打印自定义类型并打印包含其类型的 STL 容器：

```
<<<
#include "preinte.hpp"

// 自定义类型point
struct point {
    int x, y;

    // 重载operator<，用于std::map 等容器内部的大小比较
    bool operator<(const point p) const {
        return x * x + y * y < p.x * p.x + p.y * p.y;
    }
```

```cpp
};

// 实现对std::ostream的operator<<重载
std::ostream& operator<<(std::ostream& cout, const point& p) {
    // 使用str_tool::concat拼接字符串，避免多次调用<<引发制表混乱
    return cout << preinte::str_tool::concat('(', p.x, ", ", p.y, ')');
}

// point的打印格式
preinte::print_format format {
    /* _Name   = */ "point",                                  // 类型名称
    /* _Width  = */ 9,                                        // 制表时的单元格宽度
    /* _Length = */ 3,                                        // 制表时的表格列数
    /* _Digits = */ 0,                                        // 保留的小数位数
    /* _Align  = */ preinte::print_format::ALIGNMENT::RIGHT // 输出对齐格式
};

int main() {
    // 打印一般值
    point p {-2, 4};
    preinte::print(p, format, "p");

    // 打印一维数组
    point array[5] = {
        {1, 3},  {2, 4}, {6, 8},
        {-4, 7}, {5, -7}
    };
    preinte::print(array, format, "points");

    // 打印std::deque
    std::deque<point> deque = {
        {1, 5},
        {3, 4}
    };
    preinte::print(deque, format, "deque");

    // 打印std::map<int, point>
    std::map<int, point> map_ip = {
        {5, {5, -3}},
        {4, {-9, 6}}
    };
    preinte::print(map_ip, format, "map_ip");

    // 打印std::map<point, int>，其中point需提前实现operator<(point)
    std::map<point, int> map_pi = {
        {{5, -3}, 5},
        {{-9, 6}, 4}
```

```
    };
    preinte::print(map_pi, format, "map_ip");

    // 打印 std::map<point, point>
    std::map<point, point> map_pp = {
        {{2, 4}, {1, 3}},
        {{4, 6}, {3, 5}}
    };
    // 当 std::map 的键值类型均不为内置类型时，print 需依次传入键值类型的打印格式参数
    preinte::print(map_pp, format, format, "map_pp");
}


>>>
@0x67fde0 point p = (-2, 4)
@0x67fdb0 point points[5]:
```

0 - 2	(1, 3)	(2, 4)	(6, 8)
3 - 4	(-4, 7)	(5, -7)	

@0x67fd60 std::deque<point> deque[.size() = 2]:

0 - 1	(1, 5)	(3, 4)

@0x67fd30 std::map<int, point> map_ip[.size() = 2]:

\	key	value
1	4	(-9, 6)
2	5	(5, -3)

@0x67fd00 std::map<point, int> map_ip[.size() = 2]:

\	key	value
1	(5, -3)	5
2	(-9, 6)	4

@0x67fcd0 std::map<point, point> map_pp[.size() = 2]:

\	key	value
1	(2, 4)	(1, 3)
2	(4, 6)	(3, 5)

二、链表打印

preinte::print 还可以打印含有简单数据的链表。结构体 preinte::linked_node_format 用以描述一个含简单结点的链表。它的部分成员如下：

```
template <typename _Node_Ty, typename _Data_Ty, typename = is_printable_t<_Data_Ty>>
struct linked_node_format {
    const char*  node_name; // 结点名称
    print_format format;    // 打印格式
    std::function<_Data_Ty& (_Node_Ty*)> data_func; // 数据获取函数
    std::function<_Node_Ty* (_Node_Ty*)> next_func; // 遍历函数
}
```

有参构造函数如下：

```
// 数据类型内置，使用内置print_format
linked_node_format (
    const char* _Node_Name,          // 结点名称
    const std::function<_Data_Ty& (_Node_Ty*)>& _Data_Func, // 数据获取函数
    const std::function<_Node_Ty* (_Node_Ty*)>& _Next_Func  // 遍历函数
)

// 使用给定print_format
linked_node_format (
    const char*         _Node_Name,  // 结点名称
    const print_format& _Format,     // 打印格式
    const std::function<_Data_Ty& (_Node_Ty*)>& _Data_Func, // 数据获取函数
    const std::function<_Node_Ty* (_Node_Ty*)>& _Next_Func  // 遍历函数
)
```

其中，`_Data_Func` 用于获取给定结点中的数据，`_Next_Func` 用于获取给定结点的下个结点。

【示例 2-1】打印结点数据为内置类型的链表：

```
<<<
#include "preinte.hpp"

// 结点类型
struct linked_int {
    int         data;
    linked_int* next;
};

// 链表格式
preinte::linked_node_format<linked_int, int> int_node_format (
    "int_node",
    [](linked_int* p) -> int&        { return p->data; },
    [](linked_int* p) -> linked_int* { return p->next; }
);

int main() {
    // 创建链表
    linked_int* head  = new linked_int;
```

```
    linked_int* temp1 = new linked_int;
    head->next  = temp1;
    head->data  = 2;
    temp1->data = 5;
    linked_int* temp2 = new linked_int;
    temp2->data = 7;
    temp1->next = temp2;
    temp2->next = nullptr;

    // 打印链表
    preinte::print(head, int_node_format, "int_list");

    delete head;
    delete temp1;
    delete temp2;
}

>>>
@0xf31930 int_node int_list[length = 3]:
```

0 - 2	2	5	7

【示例 2-2】打印结点数据不为内置类型的链表：

```
<<<
#include "preinte.hpp"

// 自定义 point 类型
struct point {
    int x, y;
};

// 实现对 std::ostream 的 operator<<重载
std::ostream& operator<<(std::ostream& cout, const point& p) {
    return cout << preinte::str_tool::concat('(', p.x, ", ", p.y, ')');
}

// point 的打印格式
preinte::print_format point_format { "point", 9, 3, 0,
preinte::print_format::ALIGNMENT::RIGHT };

// 结点类型
struct linked_point {
    point         data;
    linked_point* next;
};

// 链表格式
```

```cpp
preinte::linked_node_format<linked_point, point> point_node_format (
    "point_node",
    point_format,
    [](linked_point* p) -> point&        { return p->data; },
    [](linked_point* p) -> linked_point* { return p->next; }
);

int main() {
    // 创建链表
    linked_point* head  = new linked_point;
    linked_point* temp1 = new linked_point;
    head->next  = temp1;
    head->data  = {2, 4};
    temp1->data = {-1, 3};
    linked_point* temp2 = new linked_point;
    temp2->data = {7, 4};
    temp1->next = temp2;
    temp2->next = nullptr;

    // 打印链表
    preinte::print(head, point_node_format, "point_list");

    delete head;
    delete temp1;
    delete temp2;
}
```

```
>>>
@0xec1930 point_node point_list[length = 3]:
```

0 - 2	(2, 4)	(-1, 3)	(7, 4)

三、控制台制表

preinte 提供了模板类型 preinte::table 用于控制台制表。以下是它和部分成员的定义：

```
template <typename _Colty1, typename... _Coltys>
class table {
public:
    using _Rowty = row<_Colty1, _Coltys...>;        // 行类型别名
    nullable_object<std::string>        title;      // 表格标题
    array<nullable_object<std::string>> headers;    // 列标头
}
```

其中_Colty1 和_Coltys 为表格的列元素类型。例如 preinte::table<int, double, std::string>的第一列类型为 int，第二列为 double，第三列为 std::string。由于各列的类型的格式参数内置情况不统一，因此在构造时要依照列序依次传入非内置类型的格式参数。preinte::table 的构造函数如下：

```
// 所有列的格式参数均内置
table ()

// 存在一列的格式参数没有内置，其余均内置
table (
    const print_format& _Format
)

// 存在多列的格式参数没有内置，其余均内置
table (
    const std::initializer_list<print_format>& _Formats
)
```

表格默认没有数据行，添加数据行需使用 add_row()函数，删除数据行需使用 remove_row(const size_t&)函数，清除所有行需使用 clear()函数。使用方括号[]便可以实现对指定数据行的访问：例如获取表格对象 tab 的第 5 行（起始为 0 行）数据，可以通过 tab[5]实现。对数据行使用函数 column<size_t>()即可访问该行中的指定列元素：例如访问表格对象第 3 行第 4 列元素（起始为 0 列），可以通过 tab[3].column<4>()实现。表格中的各元素默认为空。当元素为空时，打印表格时将填充空格。若欲将元素置空，则仅需调用其 clear()函数。

表格标题默认为空。当标题为空时，打印表格时忽略标题；否则将居中打印表格标题。若设置表格 tab 的标题为"my chart"，则 tab.title = "my chart"；将 tab 的标题重新置空，则 tab.title.clear()。

表格各列标头默认为空。当各列均无标头时，打印表格时将忽略标头；否则逐一居中打印，并将加粗下框线。若将表格 tab 的第 3 列标头置为"happy"，则 tab.headers[3] = "happy"；若将其置空，则 tab.headers[3].clear()。

preinte::table 的部分成员使用了 preinte::array 和 preinte::nullable_object，关于其具体说明，请分别参见*杂项-preinte::array*和*杂项-preinte::nullable_object*。

【示例 3-1】打印表：

```
<<<
#include "preinte.hpp"

int main() {
    // 设置内置类型的打印格式
    preinte::print_tool::set_width<std::string>(10);
    preinte::print_tool::set_alignment<int>(preinte::print_format::ALIGNMENT::LEFT);

    // 建表
```

```
    preinte::table<int, std::string> tab;
    // 设置标题
    tab.title = "English for Numbers";
    // 设置标头
    tab.headers[0] = "number";
    tab.headers[1] = "English";
    // 新建行
    tab.add_row();
    tab[0].column<0>() = 1;
    tab[0].column<1>() = "one";
    // 新建行
    tab.add_row();
    tab[1].column<0>() = 2;
    tab[1].column<1>() = "two";
    // 新建行
    tab.add_row();
    tab[2].column<0>() = 3;
    tab[2].column<1>() = "three";
    // 打印表
    tab.print();
}

>>>
English for Numbers
```

number	English
1	one
2	two
3	three

【示例 3-2】打印表:

```
<<<
#include "preinte.hpp"
#include <cmath>

// 自定义 point 类型
struct point {
    int x, y;
};

// 实现对 std::ostream 的 operator<< 重载
std::ostream& operator<<(std::ostream& cout, const point& p) {
    return cout << preinte::str_tool::concat('(', p.x, ", ", p.y, ')');
}
```

```cpp
// point 的打印格式
preinte::print_format point_format { "point", 9, 3, 0,
preinte::print_format::ALIGNMENT::RIGHT };

int main() {
    // 建表，按列类型次序传入非内置类型的打印格式
    preinte::table<point, point, double> tab({point_format, point_format});
    // 设置标题
    tab.title = "Distance Between Two Points";
    // 设置标头
    tab.headers[0] = "point 1";
    tab.headers[1] = "point 2";
    tab.headers[2] = "distance";
    // 新建行
    tab.add_row();
    tab[0].column<0>() = {2, 4};
    tab[0].column<1>() = {3, 5};
    tab[0].column<2>() = sqrt(2);
    // 新建行
    tab.add_row();
    tab[1].column<0>() = {0, 0};
    tab[1].column<1>() = {5, 5};
    tab[1].column<2>() = 5 * sqrt(2);
    // 新建行，仅对其第 2 列赋值
    tab.add_row().column<2>() = 0;
    // 打印测试
    std::cout << ">> Test 1:\n";
    tab.print();
    // 删除第 2 行
    tab.remove_row(2);
    // 将第 1 行第 2 列置空
    tab[1].column<2>().clear();
    // 打印测试
    std::cout << ">> Test 2:\n";
    tab.print();
}
```

```
>>>
>> Test 1:
  Distance Between Two Points
```

point 1	point 2	distance
(2, 4)	(3, 5)	1.41
(0, 0)	(5, 5)	7.07
		0.00

```
>> Test 2:
```

Distance Between Two Points

point 1	point 2	distance
(2, 4)	(3, 5)	1.41
(0, 0)	(5, 5)	

四、获取访问受限的容器内容

preinte 内置了函数 preinte::peep_tool::peep 用于获取访问受限的容器内容，诸如 std::stack，std::queue 和 std::priority_queue。以下该函数的声明：

```
// 获取 std::stack 内容
template <typename _Ty, typename _Container>
std::vector<_Ty> peep_tool::peep (
    const STD::stack<_Ty, _Container>& _Stack
)

// 获取 std::queue 内容
template <typename _Ty, typename _Container>
std::vector<_Ty> peep_tool::peep (
    const STD::queue<_Ty, _Container>& _Queue
)

// 获取 std::priority_queue 内容
template <typename _Ty, typename _Container, typename _Pr>
std::vector<_Ty> peep_tool::peep (
    const STD::priority_queue<_Ty, _Container, _Pr>& _Queue
)
```

函数返回容器内部元素的拷贝。

【示例 4】获取并打印访问受限的容器内容：

```
<<<
#include "preinte.hpp"

int main() {
    // 获取并打印 std::stack 内容
    std::stack<int> stack;
    stack.push(-1);
    stack.push(-2);
    stack.push(-3);
    auto stack_items = preinte::peep_tool::peep(stack);
    preinte::print(stack_items, "stack_items");

    // 获取并打印 std::queue 内容
    std::queue<double> queue;
    queue.push(2.89);
    queue.push(5.12);
    queue.push(-9.78);
    auto queue_items = preinte::peep_tool::peep(queue);
    preinte::print(queue_items, "queue_items");

    // 获取并打印 std::priority_queue 内容
    std::priority_queue<float> priority_queue;
    priority_queue.push(5.76f);
    priority_queue.push(9.96f);
```

```
    priority_queue.push(0.04f);
    auto priority_queue_items = preinte::peep_tool::peep(priority_queue);
    preinte::print(priority_queue_items, "priority_queue_items");
}

>>>
@0x67fd60 std::vector<int> stack_items[.size() = 3]:
```

0 - 2	-1	-2	-3

```
@0x67fcf0 std::vector<double> queue_items[.size() = 3]:
```

0 - 2	2.89	5.12	-9.78

```
@0x67fcb0 std::vector<float> priority_queue_items[.size() = 3]:
```

0 - 2	9.96	5.76	0.04

五、杂项

1. preinte::str_tool::concat

返回其参数依次序拼接成的字符串。函数声明如下：

```
template <typename... _Tys>
std::string str_tool::concat (
    _Tys&&... _Contents
)
```

【示例 5-1】preinte::str_tool::concat 函数示例：

```
<<<
#include "preinte.hpp"

int main() {
    std::string en = "six";
    std::string sw = preinte::str_tool::concat(en, '=', 6);
    std::cout << sw;
}

>>>
six=6
```

2. preinte::print_tool::pause

为减少断点设置、便于调试，preinte 内置了 preinte::pause 函数，与 preinte::print 等函数配合使用，可以提升调试效率。以下为其函数声明：

```
void print_tool::pause (
    const std::string& _Tip = ""
)
```

【示例 5-2】preinte::print_tool::pause 函数示例：

```
<<<
#include "preinte.hpp"

int main() {
    std::vector<double> lfs;
    for (int i = 0; i < 3; i++) {
        lfs.push_back(rand());
        preinte::print(lfs, "lfs");
        // pause 和 concat 配合使用
        preinte::print_tool::pause(preinte::str_tool::concat("i = ", i));
    }
}

>>>
@0x63fdd0 std::vector<double> lfs[.size() = 1]:
```

0	41.00

```
********************************************************************************
```

```
[2023/08/23 22:55:28] i = 0
[2023/08/23 22:55:28] Program is paused. Please press 'Enter' to continue.
********************************************************************************
@0x63fdd0 std::vector<double> lfs[.size() = 2]:
```

0 - 1	41.00	18467.00

```
********************************************************************************
[2023/08/23 22:55:31] i = 1
[2023/08/23 22:55:31] Program is paused. Please press 'Enter' to continue.
********************************************************************************
@0x63fdd0 std::vector<double> lfs[.size() = 3]:
```

0 - 2	41.00	18467.00	6334.00

```
********************************************************************************
[2023/08/23 22:55:31] i = 2
[2023/08/23 22:55:31] Program is paused. Please press 'Enter' to continue.
********************************************************************************
```

3. 输出流换向

std::cout 的默认输出流指向控制台，在一些需求中，可能需要指向文件。preinte 简单地封装了一对函数用于便捷的切换输出流位置。以下是函数的声明:

```
// 将输出流指向给定文件
void print_tool::to_file (
    std::ofstream& _File_Stream
)

// 将输出流指向控制台
void print_tool::to_console ()
```

【示例 5-3】输出流换向示例:

```
<<<
#include "preinte.hpp"

int main() {
    // 打印到控制台
    std::vector<int> v = {1, 2, 3};
    std::cout << "1: Print to console.\n";
    preinte::print(v, "v");

    // 打印到文件
    std::ofstream file("D:\\try.txt");
    preinte::print_tool::to_file(file);
    v.push_back(4);
    std::cout << "2: Print to file.\n";
    preinte::print(v, "v");

    // 打印到控制台
    preinte::print_tool::to_console();
    v.push_back(5);
    std::cout << "3: Print to console.\n";
```

```
    preinte::print(v, "v");
}


>>> console:
1: Print to console.
@0x63fde0 std::vector<int> v[.size() = 3]:
```

0 - 2	1	2	3

```
3: Print to console.
@0x63fde0 std::vector<int> v[.size() = 5]:
```

0 - 4	1	2	3	4	5

```
>>> D:\try.txt:
2: Print to file.
@0x63fde0 std::vector<int> v[.size() = 4]:
```

0 - 3	1	2	3	4

4. 打表样式

preinte 内置了两种打表样式：默认的制表符样式和基础样式。切换这两种样式的函数如下：

```
// 简易样式
void print_tool::basic_style()

// 制表符样式
void print_tool::normal_style()
```

【示例 5-4】更换打表样式示例：

```
<<<
#include "preinte.hpp"

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    char csarr[3][32] = {0};
    sprintf(csarr[0], " - i love you.");
    sprintf(csarr[1], " - 6.");
    sprintf(csarr[2], " - okay, i'll remake.");

    // 基础样式
    std::cout << "Basic Style:\n";
    preinte::table_printer::basic_style();
    preinte::print(arr, "arr");
    preinte::print(csarr, "csarr");

    // 制表符样式
    std::cout << "Normal Style:\n";
    preinte::table_printer::normal_style();
    preinte::print(arr, "arr");
    preinte::print(csarr, "csarr");
```

```
}


>>>
Basic Style:
@0x63fdf0 int arr[5]:
+-------+-------+-------+-------+-------+-------+
| 0 - 4 |      1|      2|      3|      4|      5|
+-------+-------+-------+-------+-------+-------+
@0x63fd90 char csarr[3][32]:
+---+-------------------------------------------------------------------------------+
| 0 | - i love you.                                                                 |
+---+-------------------------------------------------------------------------------+
| 1 | - 6.                                                                          |
+---+-------------------------------------------------------------------------------+
| 2 | - okay, i'll remake.                                                          |
+---+-------------------------------------------------------------------------------+
Normal Style:
@0x63fdf0 int arr[5]:
```

0 - 4	1	2	3	4	5

@0x63fd90 char csarr[3][32]:

0	- i love you.
1	- 6.
2	- okay, i'll remake.

5. preinte::array

内存安全的定长数组，深拷贝赋值。其部分成员声明如下：

```
// 通过给定长度初始化空数组
array (
    const size_t& _Size
);

// 通过 C 风格数组深拷贝初始化
array (
    const _Ty*    _Arr,
    const size_t& _Size
)

// 下标访问
_Ty& operator[] (
    const size_t& _Id
)

// 获取长度
const size_t size()

// 隐式转换为 C 风格数组
inline operator _Ty* ()
```

【示例 5-5】preinte::array 使用示例：

```
<<<
#include "preinte.hpp"
int main() {
    // 给定长度初始化空数组
    preinte::array<int> arr1(3);
    arr1[0] = -1;
    arr1[1] = -2;
    arr1[2] = -3;
    preinte::print(arr1, "arr1");

    // 用初始化列表初始化数组
    preinte::array<int> arr2 = {1, 2, 3, 4, 5};
    preinte::print(arr2, "arr2");

    // 用 C 风格数组深拷贝初始化新数组
    int* heap = new int[4] {-2, 56, 98, 72};
    preinte::array<int> arr3(heap, 4);
    delete[] heap;
    preinte::print(arr3, "arr3");

    // 用现有数组深拷贝初始化新数组
    preinte::array<int> arr4 = arr3;
    preinte::print(arr4, "arr4");
}

>>>
@0x62fda0 preinte::array<int> arr1[.size() = 3]:
```

0 - 2	-1	-2	-3

```
@0x62fd80 preinte::array<int> arr2[.size() = 5]:
```

0 - 4	1	2	3	4	5

```
@0xf31da0 preinte::array<int> arr3[.size() = 4]:
```

0 - 3	-2	56	98	72

```
@0x62fd60 preinte::array<int> arr4[.size() = 4]:
```

0 - 3	-2	56	98	72

6. preinte::nullable_object

可空类型，深拷贝构造与赋值。其部分成员声明如下：

```
// 默认构造函数
nullable_object ()

// 给定对象构造
```

```
nullable_object (
    const _Ty& _Value
)

// 获取包含对象
const _Ty& object()

// 是否为空
bool is_null()

// 箭头运算符，执行对象的方法
_Ty* operator->()

// 析构对象，置空
void clear()
```

【示例 5-6】preinte::nullable_object 使用示例:

```
<<<
#include "preinte.hpp"

// 打印函数
void print_obj(const preinte::nullable_object<std::string>& _Obj) {
    std::cout << (_Obj.is_null() ? "null" : _Obj.object()) << '\n';
}

int main() {
    // 空对象
    preinte::nullable_object<std::string> obj;
    print_obj(obj);
    // 赋值后非空
    obj = "This is a test for nullable_object.";
    print_obj(obj);
    // 调用包含对象的函数
    obj->append("666");
    print_obj(obj);
    // 置空
    obj.clear();
    print_obj(obj);
}

>>>
null
This is a test for nullable_object.
This is a test for nullable_object.666
null
```