通用工具

1. pairs

class pair 可以将两个值视为一个单元，比较常见的使用案例是 map 和 multimap 通过 pair 来管 理 key/value 结构。

pair 实现

pair 被定义为 struct 意味着所有成员都是 public 的。其中的 copy constructor 是 template 形式，是因为可能发生隐式类型转换。

pair 之间的比较

STL 为 pair 提供了管用的操作符，如果两个 pair 对象内的所有元素都相等，那么这两个 pair 会被 视为相等。比较顺序是先 first 后 second。

make_pair

make_pair 可以无需写出类型就生成一个 pair 对象。这一点在进行函数调用时最方便使用。

```
std::pair<int, float>(42, 7.77);
std::make_pair(42, 7.77); \\ 更简单
```

2. class auto_ptr

auto_ptr 是一种智能型指针，可以帮助程序员防止 “被异常抛出时发生资源泄露”。

auto_ptr 的设计动机

以下是一个通常带有资源变量的函数写法。

```
void f() {
  A* ptr = new A();
  ...
  delete ptr;
}
```

上面这段代码可能出现的问题是，程序员很有可能会忘记 delete 操作，导致资源没有被释放而产生资源 泄露。除了普通资源泄露问题之外，有一个更大的隐患在于当异常发生时，如果发生异常更容易忘记进行资 源释放。

```
void f() {
  A* ptr = new A();
  try {

  } catch(...) {
```

```
    delete ptr; // 发生异常时也要释放资源
    throw;
  }
  delete ptr;
}
```

智能型指针可以在 scoop 结束时自动销毁，就和局部变量一样，这一点很好的解决了资源泄露的问题。

```
void f() {
  std::auto_ptr<A> ptr(new A);
  ...
}
```

但是需要注意，auto_ptr<> 不允许使用一般指针惯用的赋值操作进行初始化。

```
std::auto_ptr<A> ptr1(new A);
std::auto_ptr<A> ptr2 = ptr1; // 错误
```

auto_ptr 拥有权转移

auto_ptr 所界定的是一种严格的拥有权观念，那么不要让同一个对象为初值将两个 auto_ptr 初始化。 这个条件还会导致一个问题：auto_ptr 的 copy constructor 和 assignment 操作符会将所有权交 出而非简单的进行数据复制。

```
std::auto_ptr<A> ptr1(new A);
std::auto_ptr<A> ptr2(ptr1); // ptr2 拥有了 ptr1 原来所拥有的数据，而 ptr1 指向了
null
```

起点和终点

因为 auto_ptr 的拥有权可以进行转移，所以可以有两种特殊用法：

1. 数据终点 如果某个函数作为数据终点出现，它以 by value 方式接收一个 auto_ptr 并不再传出。

```
void sink(std::auto_ptr<A>);
```

2. 数据起点 如果某个函数作为数据起点，它会以 by value 的方式返回一个 auto_ptr，这意味着产生了一个数据。

```
std::auto_ptr<A> f() {
  std::auto_ptr<A> ptr(new A);
  ...
```

```
  return ptr;
}
```

这样做，也不会造成数据泄露，如果使用传统的 new 进行数据分配就很容易造成数据的泄露。

缺陷

auto_ptr 的语义本身就具有拥有权，如果无意转交该拥有权，就不要使用 auto_ptr，否则可能会产生 灾难。

```
template <class T>
void bad_print(std::auto_ptr<T> p) {
  if (p.get() == nullptr) {
    std::cout << "nullptr";
  } else {
    std::cout << *p;
  }
}

std::auto_ptr<int> p(new int);
*p = 42;
bad_print(p);
*p = 18; // 错误，因为拥有权已经被交出
```

也许你会想要使用 reference 来操作，但是对于 auto_ptr 而言，这种操作时没有被定义的。如果你 不想交出拥有权又想使用 auto_ptr，将它声明为 const 的，让所有问题暴露在编译期。

```
const std::auto_ptr<int> p(new int);
*p = 42;
bad_print(p); // 编译错误，因为不能将 const 转向 non-const reference
*p = 18;
```

const auto_ptr 减小了 “不经意转移拥有权” 的风险。这里的 const 并不意味着你不可以更改 auto_ptr 中的数据，而是指你不可以更改 auto_ptr 的拥有权。

> [!tip] 如果想要规避 auto_ptr 的拥有权带来的问题，可以使用 unique_ptr + move() 来替代，这也是 为什么 auto_ptr 被遗弃了。

auto_ptrs 作为成员之一

在 class 中使用 auto_ptr 可以避免资源泄露，即便在初始化期间抛出异常，auto_ptr 也会自动释放 避免资源泄露。

auto_ptrs 错误运用

1. auto_ptr 之间不能共享拥有权。(如果你需要共享拥有权，你可以使用 shared_ptr)
2. 并不存在针对 array 而设计的 auto_ptr。auto_ptr 是通过 delete 来释放的，而非 delete[] 因此不能指向 array。你也可以使用 shared_ptr 来指向数组，只需要为其提供一个 删除器即可。

```
shared_ptr<int> ptr(new int[100], [](int* p) { delete[] p; });
```

3. auto_ptr 并非适用于所有地方，比如不能引用计数。
4. auto_ptr 不满足 STL 容器对其元素的要求。因为 auto_ptr 在 复制/赋值 动作后，拥有权被交出，而非简单的复制操作。

auto_ptr 运用实例

```
#include <iostream>
#include <memory>

//! 千万不要写成：std::ostream& operator<<(std::ostream& os, const std::auto_ptr<T>
p)
template <class T>
std::ostream& operator<<(std::ostream& os, const std::auto_ptr<T>& p) {
  if (p.get() == nullptr) os << "null";
  else os << *p;
  return os;
}

int main() {
  std::auto_ptr<int> p(new int(42));
  std::auto_ptr<int> q;

  std::cout << "after initialization:" << std::endl;
  std::cout << "p: " << p << std::endl;
  std::cout << "q: " << q << std::endl;

  q = p;
  std::cout << "after assigning auto pointers:" << std::endl;
  std::cout << "p: " << p << std::endl;
  std::cout << "q: " << q << std::endl;

  *q += 13;
  p = q;
  std::cout << "after change and assigning:" << std::endl;
  std::cout << "p: " << p << std::endl;
  std::cout << "q: " << q << std::endl;

  std::cout << "================================" << std::endl;

  const std::auto_ptr<int> a(new int(20));
  const std::auto_ptr<int> b(new int(0));
  const std::auto_ptr<int> c;

  std::cout << "after initialization:" << std::endl;
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;
  std::cout << "c: " << c << std::endl;
```

```
  *b = *a;
  //! *c = *a; // 错误写法
  *a = -77;

  std::cout << "after assigning auto pointers:" << std::endl;
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;
  std::cout << "c: " << c << std::endl;

  //! a = b;
  //! c = b;
  std::auto_ptr<int>::element_type;

  std::cout << "================================" << std::endl;

  std::auto_ptr<int> e;
  e.reset(new int(12));
  std::cout << "after reset auto pointers:" << std::endl;
  std::cout << "e: " << e << std::endl;
}
```

auto_ptr 实作细目

auto_ptr

类型定义: `auto_ptr<T>::element_type` 构造函数: auto_ptr 提供了多种构造函数，并全都进行了 explicit 限定，这也是为什么没有办法进行 赋值初始化，因为没有办法隐式类型转换。 数值存取：

- `auto_ptr::get()`
- `auto_ptr::operator*()`
- `auto_ptr::operator->()` 数值操作：
- `auto_ptr::release()`: 放弃 auto_ptr 原先拥有的对象拥有权，并返回对象地址，如果没有对象返回 null 指针
- `auto_ptr::reset(T* ptr = 0)`: 以 ptr 重新初始化 auto_ptr

3. 数值极限

使用新式的 class numeric_limits<> (#include) 或者老式的 类型限制选项(#include)。

class numeric_limits<> 通过模板和特化来实现，通用性 template 为所有类别提供缺省值，特化版 本为不同的具体类型提供极值。

numeric_limits

```
#include <limits>
#include <string>
#include <iostream>
#include <climits>

void print(std::string s, auto a) {
  if (typeid(a) == typeid(bool)) {
```

```
    if (a == false) std::cout << s << ": false" << std::endl;
    else std::cout << s << ": true" << std::endl;
  } else {
    std::cout << s << ": " << a << std::endl;
  }
}

int main() {
  print("int 是否有极值", std::numeric_limits<int>::is_specialized);
  print("int 带有正负号", std::numeric_limits<int>::is_signed);
  print("int 是否是整数类别", std::numeric_limits<int>::is_integer);
  print("int 是否是精确值", std::numeric_limits<int>::is_exact);
  print("int 数值集合有限", std::numeric_limits<int>::is_bounded);
  print("int 正值相加可能因为溢出而回绕", std::numeric_limits<int>::is_modulo);
  print("int 最小值", std::numeric_limits<int>::min());
  print("int 最大值", std::numeric_limits<int>::max());
  print("int 位个数", std::numeric_limits<int>::is_specialized);
  print("int 底数", std::numeric_limits<int>::radix);
  print("int 底数的最小负整数指数", std::numeric_limits<int>::min_exponent);
  print("int 底数的最大负整数指数", std::numeric_limits<int>::max_exponent);
  print("int 最小间隔", std::numeric_limits<int>::epsilon());
  print("int 舍入风格", std::numeric_limits<int>::round_style);
  print("int 舍入误差", std::numeric_limits<int>::round_error());
  print("int 有正无穷大", std::numeric_limits<int>::has_infinity);
  print("int 正无穷大", std::numeric_limits<int>::infinity());
}
```

numeric_limits 的舍入风格

- round_toward_zero: 向零舍入
- round_to_neares: 像最接近的可表示值舍入
- round_toward_infinity: 向正无限舍入
- round_toward_neg_infinity: 向负无限舍入
- round_indeterminate: 无法确定

4. 辅助函数

挑选较大值或较小值

`min(const T& a, const T& b)` 或者 `min(const T& a, const T& b, Compare comp)` `max(const T& a, const T& b)` 或者 `max(const T& a, const T& b, Compare comp)`

```
#include <algorithm>
#include <iostream>

bool int_ptr_less(int* a, int* b) {
  return *a < *b;
}

int main() {
```

```
  int* pa = new int(12);
  int* pb = new int(13);

  int* pmax = std::max(pa, pb, [](int* a, int* b) {
    return *a < *b;
  });
  // int* pmax = std::max(pa, pb, int_ptr_less);

  std::cout << "min(0, 1): " << std::min(0, 1) << std::endl;
  std::cout << "max(" << *pa << ", " << *pb << "): " << *pmax << std::endl;
}
```

两值呼唤

`swap(T& a, T& b)` 用来交换任意变量。需要注意，swap 以来 copy 构造函数和 assignment 操 作进行。其最大的优势在于，通过 模板特化 或者 函数重载 可以为更复杂的类型提供特殊实现版本。

```
#include <algorithm>

class MyContainer {
public:
  MyContainer(int nums) : numElems(nums) {
    if (nums == 0) elems = 0;
    else elems = new int[nums];
  }
  ~MyContainer() {
    delete[] elems;
  }

  void swap(MyContainer& x) {
    std::swap(elems, x.elems);
    std::swap(numElems, x.numElems);
  }
private:
  int* elems;
  int numElems;
};

inline void swap(MyContainer& c1, MyContainer& c2) {
  c1.swap(c2);
}

int main() {
  MyContainer a(12);
  MyContainer b(5);
  swap(a, b);
}
```

5. 辅助性的比较操作符

在 STL 中定义了 !=, >, <=, >= 四个操作符，只需要你在设计类时 重载 ==, < 就可以使用这几个 额外的操作符。

使用时只需要三步：

1. #include <utility>
2. using namespace std::rel_ops
3. 在自定义类中重载 operator< 和 operator==

```
#include <string>
#include <iostream>
#include <utility>

using namespace std::rel_ops;

class Rational {
public:
  Rational(int x, int y) : numerator(x), denominator(y) {}
  bool operator==(const Rational& rhs) const {
    return ((*this).numerator * rhs.denominator) == ((*this).denominator *
rhs.numerator);
  }
  bool operator<(const Rational& rhs) const {
    return ((*this).numerator * rhs.denominator) < ((*this).denominator *
rhs.numerator);
  }

  friend std::ostream& operator<<(std::ostream& os, const Rational& r);
private:
  int numerator;
  int denominator;
};

std::ostream& operator<<(std::ostream& os, const Rational& r) {
  os << r.numerator << "/" << r.denominator;
  return os;
}

int main() {
  Rational a(1, 2);
  Rational b(3, 4);

  std::cout << a << " == " << b << " is: " << (a == b ? "true" : "false") <<
std::endl;
  std::cout << a << " < " << b << " is: " << (a < b ? "true" : "false") <<
std::endl;
  std::cout << a << " != " << b << " is: " << (a != b ? "true" : "false") <<
std::endl;
  std::cout << a << " > " << b << " is: " << (a > b ? "true" : "false") <<
std::endl;
  std::cout << a << " >= " << b << " is: " << (a >= b ? "true" : "false") <<
std::endl;
```

```
    std::cout << a << " <= " << b << " is: " << (a <= b ? "true" : "false") <<
std::endl;
}
```

6. 头文件 <cstddef> 和 <cstdlib>

<cstddef>

- NULL: 空指针值。在c++ 中不正确，因为 NULL 被定义为 (void*)0，而 c++ 没有从 void* 到任何其他类型的自动转型操作。
- size_t: 无正负号类型，用来表示大小
- ptrdiff_t: 有正负号类型，用来表示指针之间的距离
- offsetof: 表示一个成员在 struct/union 中的偏移量

<cstdlib>

- exit(int status): 退出程序，销毁 static 对象，刷新缓冲区，关闭 IO 通道，调用 atexit 注册的函数，终止程序
- EXIT_SUCCESS: 程序正常结束
- EXIT_FAILURE: 程序不正常结束
- abort(): 退出程序，不做任何清理工作
- atexit(void (*function)()): 退出程序时调用某些函数