做法

平面容器是 Boost.Container 库的一部分。我们已经在前面的秘笈中看到了如何使用它的一些容器。在这个秘笈中，我们将使用一个 flat_set 关联式容器。

1. 只需要包含一个头文件：

```
#include <boost/container/flat_set.hpp>
```

2. 之后，就可以随意构建平面容器：

```
boost::container::flat_set<int> set;
```

3. 为元素预留空间：

```
set.reserve(4096);
```

4. 填充容器：

```
for (int i = 0; i < 4000; ++i) {
    set.insert(i);
}
```

5. 现在，就可以像用 std::set 那样使用它：

```
//5.1
assert(set.lower_bound(500) - set.lower_bound(100) ==
  400);

//5.2
set.erase(0);

//5.3
set.erase(5000);

//5.4
assert(std::lower_bound(set.cbegin(), set.cend(),
  900000) == set.cend());

//5.5
assert(
    set.lower_bound(100) + 400
    ==
    set.find(500)
);
```

工作原理

第 1 步和第 2 步都是很简单的，但需要注意第 3 步。它是使用关联式容器和 std::vector 最重要的步骤之一。

boost::container::flat_set 类在向量中有序地存储值，这意味着任何插入或删除元素都花费线性时间复杂度 0(N)，就像 std::vector 一样。这是一种必要的牺牲。但是，我们获得了每个元素使用几乎三分之一的内存，更多对处理器缓存友好的存储，以及随机访问迭代器等好

处。在第 5 步中，5.1，我们得到由调用 lower_bound 成员函数返回的两个迭代器之间的距离。在平面集合中获取距离花费恒定的时间复杂度 0(1), 而 std::set 的迭代器执行相同的操作花费线性时间 0(N)。在 5.1 的情况下，使用 std::set 得到距离比在平面集合容器中获得距离会慢 400 倍。

回到第 3 步。若不预留内存，插入元素会慢数倍，并且内存的使用效率更低。std::vector 类分配所需的内存块，并在该内存块中就地构建元素。当我们插入一些元素而没有预留内存时，预分配的内存块中很可能没有剩余空间，这样 std::vector 会分配两倍于先前分配的内存块。此后，std::vector 将从第一个块复制或移动元素到第二个块，删除第一个块中的元素，释放第一个块。只有在此之后，插入才会发生。在插入过程中，这种复制和释放可能多次出现，从而大幅度降低速度。

如果你知道 std::vector 或任何平面容器必须存储的元素的个数，就在插入前为这些元素预留空间。这条规则没有例外！

第 4 步是很简单的，我们在这里插入元素。请注意，我们插入有序的元素。这不是必需的，但建议这么做以加速插入。在 std::vector 的末端插入元素比在中间位置或在开头插入要廉价得多。

在第 5 步中，5.2 和 5.3 除了执行速度外，没有多大差别。擦除元素的规则与插入它们几乎是相同的，所以请参阅前一段的解释。

也许我正在告诉你关于容器的简单的事情，但我已经看到了一些使用 C++11 功能的非常流行的产品，它们具有大量的优化但在使用 STL 容器方面是蹩脚的，尤其是 std::vector。

在第 5 步中，5.4 显示由于随机访问迭代器，使用 boost::container::flat_set 时，std::lower_bound 函数会运行得比使用 std::set 时更快。

在第 5 步中，5.5 也展示了随机访问迭代器的好处。请注意，在这里我们并没有使用 std::find 函数。这是因为该功能需要花费线性时间复杂度 0(N)，而 find 成员函数花费对数时间复杂度 0(log(N))。

还有更多

应当在何时使用平面容器，又应在何时使用平常的容器呢？嗯，这由你决定，下面是来自 Boost.Container 官方文档的一个差异列表，它将帮助你做决定：

- 比标准关联容器更快的查找。
- 比标准关联容器快得多的迭代。
- 对于小对象，消耗更少的内存（如果使用 shrink_to_fit，大型对象也是）。
- 改进的缓存性能（数据存储在连续的内存中）。
- 不稳定的迭代器（插入和删除元素时，迭代器失效）。

- 不能存储不可复制和不可移动的值类型。
- 比标准关联容器更弱的异常安全性（在移动删除和插入的值时复制 / 移动构造函数可能抛出一个异常）。
- 比标准关联容器更慢的插入和删除（特别对于不可移动类型）。

遗憾的是，在 C++11 中没有平面容器。Boost 的平面容器快速且有很多的优化，并且不使用虚函数。来自 Boost.Containers 的类通过 Boost.Move 模拟支持右值引用，所以即使在 C++03 编译器中，也可以随意地使用它们。

参见

- 参阅秘笈 72 获得有关 Boost.Container 的更多信息。
- 秘笈 10 会包含在 C++03 兼容编译器中模拟右值引用的基础知识。
- Boost.Container 的官方文档包含了很多关于 Boost.Container 的有用信息和每个类的完整参考。该文档位于 http://www.boost.org/doc/libs/1_53_0/doc/html/container.html。

第 10 章

收集平台和编译器信息

不同的项目和公司有不同的编码要求。有些会禁止异常或者 RTTI 而有些会禁止 C++11。如果你希望编写可用于广泛的项目的可移植代码，本章就是为你写的。

要使你的代码尽可能地快，并使用最新的 C++ 特性吗？你绝对会需要一个工具来检测编译器功能。

有些编译器具有独特的功能，可以大大简化你的工作。如果你针对的是单个编译器，可以节省很多时间并且使用这些功能。不需要从头开始实现它们！

本章专门介绍用于检测编译器、平台和 Boost 功能的不同的辅助宏。这些宏被广泛应用于 Boost 库，并且其对于编写能够使用任何编译标志的可移植代码是必不可少的。

秘笈 74　检测 int128 支持

一些编译器支持扩展算术类型，如 128 位的浮点数或整数。我们快速浏览一下如何使用 Boost 来使用它们。我们将创建一个接受 3 个参数的方法，并返回这些参数的乘积。

准备

只需有 C++ 基础知识。

做法

要使用 128 位的整数，需要什么呢？一些显示它们可用的宏和一些具有跨平台的可移植类型名称的类型定义。

1. 只需要一个头文件：

```
#include <boost/config.hpp>
```

2. 现在需要检测 int128 支持：

```
#ifdef BOOST_HAS_INT128
```

3. 添加一些 typedef 并且实现方法如下：

```
typedef boost::int128_type int_t;
typedef boost::uint128_type uint_t;

inline int_t mul(int_t v1, int_t v2, int_t v3) {
    return v1 * v2 * v3;
}
```

4. 对于不支持 int128 类型的编译器，可能需要 int64 类型的支持：

```
#else // BOOST_NO_LONG_LONG

#ifdef BOOST_NO_LONG_LONG
#error "This code requires at least int64_t support"
#endif
```

5. 现在，需要为没有 int128 支持的编译器使用 int64 来提供一些实现：

```
struct int_t { boost::long_long_type hi, lo; };
struct uint_t { boost::ulong_long_type hi, lo; };

inline int_t mul(int_t v1, int_t v2, int_t v3) {
    // 一些手写数学代码
    //...
}

#endif // BOOST_NO_LONG_LONG
```

工作原理

头文件 <boost/config.hpp> 包含很多描述平台和编译器功能的宏。在这个例子中，使用 BOOST_HAS_INT128 检测对 128 位的整数的支持，并且用 BOOST_NO_LONG_LONG 检测对 64 位整数的支持。

正如我们从这个例子可以看到的，Boost 有 64 位有符号和无符号整数的 typedef：

```
boost::long_long_type
boost::ulong_long_type
```

它还具有 128 位有符号和无符号整数的 typedef：

```
boost::int128_type
boost::uint128_type
```

还有更多

C++11 通过内置 long long int 和 unsigned long long int 类型支持 64 位的类型。遗憾的是，并不是所有的编译器都支持 C++11，所以 BOOST_NO_LONG_LONG 将对你有用。128 位的整数不是 C++11 的一部分，所以从 Boost 的 typedef 和宏是编写可移植代码的唯一方式。

参见

- 阅读秘笈 75 获得有关 Boost.Config 的更多信息。
- 阅读 Boost.Config 的官方文档获得有关它的能力的更多信息，该文档位于 http://www.boost.org/doc/libs/1_53_0/libs/config/doc/html/index.html。
- Boost 中有一个库，使用它可以构造无限精度的类型。参阅 Boost.Multiprecision 库，其位于 http://www.boost.org/doc/libs/1_53_0/libs/multiprecision/doc/html/index.html。

秘笈 75　检测 RTTI 支持

一些公司和库对它们的 C++ 代码有特定的要求，如在没有运行时类型信息（Runtime type information RTTI）时才能编译成功。在这个小秘笈中，我们将了解如何检测出禁用的 RTTI，如何存储类型的相关信息，并在运行时对类型进行比较，即使没有 typeid。

准备

此秘笈需要使用 C++ 的 RTTI 的基础知识。

做法

检测禁用的 RTTI，存储类型相关信息，并在运行时比较类型，是被广泛应用于整个 Boost 库的技巧。这方面的例子是 Boost.Exception 和 Boost.Function。

1. 要做到这一点，首先需要包含以下头文件：

```
#include <boost/config.hpp>
```

2. 首先来看 RTTI 被启用，并且 C++11 的 std::type_index 类可用的情况：

```
#if !defined(BOOST_NO_RTTI) \
    && !defined(BOOST_NO_CXX11_HDR_TYPEINDEX)

#include <typeindex>
using std::type_index;

template <class T>
type_index type_id() {
    return typeid(T);
}
```

3. 否则，需要构建我们自己的 type_index 类：

```
#else

#include <cstring>

struct type_index {
    const char* name_;

    explicit type_index(const char* name)
        : name_(name)
```

```
    {}
};
inline bool operator == (const type_index& v1,
    const type_index& v2)
{
    return !std::strcmp(v1.name_, v2.name_);
}

inline bool operator != (const type_index& v1,
    const type_index& v2)
{
    // '!!' 用于抑制警告
    return !!std::strcmp(v1.name_, v2.name_);
}
```

4. 最后一步是定义 type_id 函数：

```
#include <boost/current_function.hpp>

template <class T>
inline type_index type_id() {
    return type_index(BOOST_CURRENT_FUNCTION);
}
#endif
```

5. 现在，可以对类型进行比较：

```
assert(type_id<unsigned int>() == type_id<unsigned>());
assert(type_id<double>() != type_id<long double>());
```

工作原理

如果 RTTI 被禁用，宏 BOOST_NO_RTTI 将被定义，而且当编译器没有 <typeindex> 头文件并且没有 std::type_index 类时，宏 BOOST_NO_CXX11_HDR_TYPEINDEX 将被定义。

上一节第 3 步的手写 type_index 结构只持有某个字符串的指针，这里没有什么真正有趣的东西。

我们来看看 BOOST_CURRENT_FUNCTION 宏。它返回当前函数的全名，包括模板参数、参数和返回类型。例如，type_id<double>() 将被表示为：

```
type_index type_id() [with T = double]
```

所以，对于任何其他类型，BOOST_CURRENT_FUNCTION 将返回不同的字符串，这就是为什么例子中的 type_index 变量并不等于它的原因。

还有更多

为得到完整的函数名称和 RTTI，不同的编译器有不同的宏。使用 Boost 的宏是可移植性最好的解决方案。宏 BOOST_CURRENT_FUNCTION 在编译时返回名称，所以这意味着最小的运行时开销。

参见

- 阅读即将介绍的秘笈了解 Boost.Config 的更多信息。
- 浏览 https://github.com/apolukhin/type_index 并参考那里的库，它使用这个秘笈的所有技巧来实现 type_index。
- 阅读 Boost.Config 的官方文档，该文档位于 http://www.boost.org/doc/libs/1_53_0/libs/config/doc/html/index.html。

秘笈 76　使用 C++11 外部模板加快编译速度

还记得你使用一些头文件中声明的复杂的模板类的某些情况吗？这样的类的例子有 boost::variant，其来自 Boost.Container 的容器，或 Boost.Spirit 解析器。当我们用这样的类或方法时，它们通常分别在使用它们的每个源文件中被编译（实例化），而重复的内容都在连接过程中扔掉了。在一些编译器中，这可能会导致编译速度缓慢。

如果有一些方式来告诉编译器在哪个源文件中实例化它就好了！

准备

此秘笈需要有模板的基础知识。

做法

这种方法被广泛应用于确实支持现代 C++ 标准库的编译器中。例如，GCC 发布时附带的 STL 库，使用这种技术来实例化 std::basic_string<char> 和 std::basic_fstream<char>。

1. 要自己做这件事，需要包含以下头文件：

```
#include <boost/config.hpp>
```

2. 还需要包含一个头文件，其中包含一个我们希望减少其实例数的模板类，：

```
#include <boost/variant.hpp>
#include <boost/blank.hpp>
#include <string>
```

3. 以下是适用于支持 C++11 的外部模板的编译器的代码：

```
#ifndef BOOST_NO_CXX11_EXTERN_TEMPLATE

extern template class boost::variant<
    boost::blank,
    int,
    std::string,
    double
>;

#endif
```

4. 现在，需要在希望模板被实例化的源文件中添加下面的代码：

```
// 带有 'extern template' 的头文件
#include "header.hpp"

#ifndef BOOST_NO_CXX11_EXTERN_TEMPLATE
template class boost::variant<
    boost::blank,
    int,
    std::string,
    double
>;
#endif
```

工作原理

C++11 的关键字 extern template 只是告诉编译器，若没有一个明确的要求这么做，则不要实例化模板。

在第 4 步中的代码是一个要求，它明确地要求在这个源文件中实例化模板。

当编译器支持 C++11 的外部模板时，BOOST_NO_CXX11_EXTERN_TEMPLATES 宏被定义。

还有更多

外部模板不影响你的程序的运行时性能，而且可以显著降低一些模板类的编译时间。不要过度使用它们，它们对于小的模板类几乎是无用的。

参见

- 阅读本章其他秘笈以获得更多有关 Boost.Config 的信息。
- 阅读 Boost.Config 的官方文档获得未包括在本章的宏的相关信息，该文档位于 http://www.boost.org/doc/libs/1_53_0/libs/config/doc/html/index.html。

秘笈 77　使用更简单的方法编写元函数

第 4 章和第 8 章都描述了元编程。如果你试图使用这些章节中的技术，你可能已经注意到，编写一个元函数会花费很多的时间。因此，在编写一个可移植的实现前，使用更人性化的方法，如 C++11 constexpr 进行元函数的试验可能是一个好办法。

在这个秘笈中，我们将了解如何检测 constexpr 的支持。

准备

constexpr 函数是可以在编译时计算的函数。这是对于这个秘笈我们需要知道的所有东西。

做法

目前，只有少量编译器支持 constexpr 功能，因此进行试验可能需要一个良好的新的编

译器。我们来看如何检测编译器对 constexpr 功能的支持。

1. 仍从包含下面的头文件开始：

```
#include <boost/config.hpp>
```

2. 现在将使用 constexpr：

```
#if !defined(BOOST_NO_CXX11_CONSTEXPR) \
    && !defined(BOOST_NO_CXX11_HDR_ARRAY)

template <class T>
constexpr int get_size(const T& val) {
    return val.size() * sizeof(typename T::value_type);
}
```

3. 如果 C++11 功能缺失，打印出一个错误：

```
#else
#error "This code requires C++11 constexpr and std::array"
//这段代码需要 C++11 constexpr 和 std::array
#endif
```

4. 这就行了，现在可以随意地编写如下面的代码：

```
std::array<short, 5> arr;
assert(get_size(arr) == 5 * sizeof(short));

unsigned char data[get_size(arr)];
```

工作原理

当 C++11 constexpr 不可用时，BOOST_NO_CXX11_CONSTEXPR 宏被定义。

constexpr 关键字告诉编译器，如果函数的所有输入都是编译时常量，则该函数可以在编译时计算。C++11 对一个 constexpr 函数可以做什么施加了很多限制。C++14 将删除其中的一些限制。

当 C++11 的 std::array 类和 <array> 头文件不可用时，BOOST_NO_CXX11_HDR_ARRAY 宏被定义。

还有更多

但是，对于 constexpr，还有其他可用的有趣的宏，如下所示：

- BOOST_CONSTEXPR 宏扩展到 constexpr 或不扩展。
- BOOST_CONSTEXPR_OR_CONST 宏扩展到 constexpr 或 const。
- BOOST_STATIC_CONSTEXPR 宏与 static BOOST_ CONSTEXPR_OR_CONST 是一样的。

```
template <class T, T Value>
struct integral_constant {
    BOOST_STATIC_CONSTEXPR T value = Value;
```