# 第⑧章

# 融入实际应用

对于 C++ 这样的语言来说，除了具有普适、通用等特性外，在一些实际应用方面也是不容忽视的，比如最为典型的，也是非英语国家程序员体会最深的，与字符编码相关的种种问题。在 C++11 中，我们看到语言标准终于也对 Unicode 做了较多深入的支持，以切实地为不同语言服务。此外，以前在 C++ 编程中的各种讨人喜欢的编译器扩展，也开始逐渐被 C++11 收编或者规范化。而有了标准的支持，这些特性也将更加好用。本章中，读者可以了解 C++11 在这些方面做出的务实而有效的改进。

## 8.1 对齐支持

☞ 类别：部分人

### 8.1.1 数据对齐

在了解为什么数据需要对齐之前，我们可以回顾一下打印结构体的大小这个 C/C++ 中的经典案例。首先来看看代码清单 8-1 所示的这个例子。

**代码清单 8-1**

```
#include <iostream>
using namespace std;

struct HowManyBytes{
    char    a;
    int     b;
};

int main() {
    cout << "sizeof(char): " << sizeof(char) << endl;
    cout << "sizeof(int): " << sizeof(int) << endl;
    cout << "sizeof(HowManyBytes): " << sizeof(HowManyBytes) << endl;

    cout << endl;
    cout << "offset of char a: " << offsetof(HowManyBytes, a) << endl;
    cout << "offset of int b: " << offsetof(HowManyBytes, b) << endl;
    return 0;
```

```
}

// 编译选项 :g++ -std=c++11 8-1-1.cpp
```

在代码清单 8-1 中，结构体 HowManyBytes 由一个 char 类型成员 a 及一个 int 类型成员 b 组成。编译运行代码清单 8-1 所示的例子，我们在实验机平台上会得到如下结果：

```
sizeof(char): 1
sizeof(int): 4
sizeof(HowManyBytes): 8

offset of char a: 0
offset of int b: 4
```

可以看到，在我们的实验机上，a 和 b 两个数据的长度分别为 1 字节和 4 字节，不过当我们使用 sizeof 来计算 HowManyBytes 这个结构体所占用的内存空间时，看到其值为 8 字节。其中似乎多出来了 3 字节没有使用的空间。

出现这个现象主要是由于数据对齐要求导致的。通常情况下，C/C++ 结构体中的数据会有一定的对齐要求。在这个例子中，可以通过 offsetof 查看成员的偏移的方式来检验数据的对齐方式。这里，成员 b 的偏移是 4 字节，而成员 a 只占用了 1 字节内存空间，这意味着 b 并非紧邻着 a 排列。事实上，在我们的平台定义上，C/C++ 的 int 类型数据要求对齐到 4 字节，即要求 int 类型数据必须放在一个能够整除 4 的地址上；而 char 要求对齐到 1 字节。这就造成了成员 a 之后的 3 字节空间被空出，通常我们也称因为对齐而造成的内存留空为填充数据（padding data）。

在 C++ 中，每个类型的数据除去长度等属性之外，都还有一项“被隐藏”属性，那就是对齐方式。对于每个内置或者自定义类型，都存在一个特定的对齐方式。对齐方式通常是一个整数，它表示的是一个类型的对象存放的内存地址应满足的条件。在这里，我们简单地将其称为对齐值。

对齐的数据在读写上会有性能上的优势。比如频繁使用的数据如果与处理器的高速缓存器大小对齐，有可能提高缓存性能。而数据不对齐可能造成一些不良的后果，比较严重的当属导致应用程序退出。典型的，如在有的平台上，硬件将无法读取不按字对齐的某些类型数据，这个时候硬件会抛出异常（如 bus error）来终止程序。而更为普遍的，在一些平台上，不按照字对齐的数据会造成数据读取效率低下。因此，在程序设计时，保证数据对齐是保证正确有效读写数据的一个基本条件。

虽然从语言设计者的角度而言，将对齐方式掩盖起来会使得语言更具有亲和力。但通常由于底层硬件的设计或用途不同，以及编程语言本身在基本（内置）类型的定义上的不同，相同的类型定义在不同的平台上会有不同的长度，以及不同的对齐要求。虽然系统设计者常常会在应用程序二进制接口中（Application Binary Interface，ABI）详细规定在特定平台上数

据长度、数据对齐方式的相关信息，但是这两者存在着平台差异性则是不争的事实。在 C++ 语言中，我们可以通过 sizeof 查询数据长度，但 C++ 语言却没有对对齐方式有关的查询或者设定进行标准化，而语言本身又允许自定义类型、模板等诸多特性。编译器无法完全找到正确的对齐方式，这会在使用时造成困难。让我们来看一下代码清单 8-2 所示的这个例子。

**代码清单 8-2**

```
#include <iostream>
using namespace std;

// 自定义的 ColorVector，拥有 32 字节的数据
struct ColorVector {
    double r;
    double g;
    double b;
    double a;
};

int main() {
    // 使用 C++11 中的 alignof 来查询 ColorVector 的对齐方式
    cout << "alignof(ColorVector): " << alignof(ColorVector) << endl;
    return 1;
}

// 编译选项 :clang++ -std=c++11 8-1-2.cpp
```

在代码清单 8-2 所示的例子中，我们使用了 C++11 标准定义的 alignof 函数来查看数据的对齐方式。编译运行代码清单 8-2，我们可以看到 ColorVector 在实验机上依然是对齐到 8 字节的地址边界上。

```
alignof(ColorVector): 8
```

这与我们设计 ColorVector 的原意是不同的。现在的计算机通常会支持许多向量指令，而 ColorVector 正好是 4 组 8 字节的浮点数数据，很有潜力改造为能直接操作的向量数据。这样一来，为了能够高效地读写 ColorVector 大小的数据，我们最好能将其对齐在 32 字节的地址边界上。

在代码清单 8-3 所示的例子中，我们利用 C++11 新提供的修饰符 alignas 来重新设定 ColorVector 的对齐方式。

**代码清单 8-3**

```
#include <iostream>
using namespace std;

// 自定义的 ColorVector，对齐到 32 字节的边界
struct alignas(32) ColorVector {
```

```
    double r;
    double g;
    double b;
    double a;
};

int main() {
    // 使用 C++11 中的 alignof 来查询 ColorVector 的对齐方式
    cout << "alignof(ColorVector): " << alignof(ColorVector) << endl;
    return 1;
}

// 编译选项 :g++ -std=c++11 8-1-3.cpp
```

编译运行代码清单 8-3 所示的代码，我们会得到如下结果：

```
alignof(ColorVector): 32
```

正如我们在代码清单 8-3 中所看到的，指定数据 ColorVector 对齐到 32 字节的地址边界上，只需要声明 alignas(32) 即可。接下来我们会详细讨论 C++11 对对齐的支持。

### 8.1.2 C++11 的 alignof 和 alignas

如同我们在上一节中看到的，C++11 在新标准中为了支持对齐，主要引入两个关键字：操作符 alignof、对齐描述符（alignment-specifier）alignas。

操作符 alignof 的操作数表示一个定义完整的自定义类型或者内置类型或者变量，返回的值是一个 std::size_t 类型的整型常量。如同 sizeof 操作符一样，alignof 获得的也是一个与平台相关的值。我们可以看看代码清单 8-4 所示的例子。

**代码清单 8-4**

```
#include <iostream>
using namespace std;

class InComplete;
struct Completed{};

int main(){
    int a;
    long long b;
    auto & c = b;
    char d[1024];

    // 对内置类型和完整类型使用 alignof
    cout << alignof(int) << endl          //  4
        << alignof(Completed) << endl;    //  1
```

```
    // 对变量、引用或者数组使用 alignof
    cout << alignof(a) << endl          //  4
         << alignof(b) << endl          //  8
         << alignof(c) << endl          //  8，与 b 相同
         << alignof(d) << endl;         //  1，与元素要求相同

    // 本句无法通过编译，Incomplete 类型不完整
    // cout << alignof(Incomplete) << endl;
}

// 编译选项 :g++ -std=c++11 8-1-4.cpp
```

使用 alignof 很简单，基本上没有什么特别的限制。不过在代码清单 8-4 中，类型定义不完整的 class InComplete 是无法通过编译的。其他的规则则基本跟大多数人想象的相同：引用 c 与其引用的数据 b 对齐值相同，数组的对齐值由其元素决定。

我们再来看看对齐描述符 alignas。事实上，alignas 既可以接受常量表达式，也可以接受类型作为参数，比如

```
alignas(double) char c;
```

也是合法的描述符。其使用效果跟

```
alignas(alignof(double)) char c;
```

是一样的。

**注意** 在 C++11 标准之前，我们也可以使用一些编译器的扩展来描述对齐方式，比如 GNU 格式的 __attribute__((__aligned__(8))) 就是一个广泛被接受的版本。

我们在使用常量表达式作为 alignas 的操作符的时候，其结果必须是以 2 的自然数幂次作为对齐值。对齐值越大，我们称其对齐要求越高；而对齐值越小，其对齐要求也越低。由于 2 的幂次的关系，能够满足严格对齐要求的对齐方式也总是能够满足要求低的对齐值的。

在 C++11 标准中规定了一个“基本对齐值”（fundamental alignment）。一般情况下其值通常等于平台上支持的最大标量类型数据的对齐值（常常是 long double）。我们可以通过 alignof(std::max_align_t) 来查询其值。而像我们在代码清单 8-3 中设定 ColorVector 对齐值到 32 字节（超过标准对齐）的做法称为扩展对齐（extended alignment）。不过即使使用了扩展对齐，也并非意味着程序员可以随心所欲。对于每个平台，系统能够支持的对齐值总是有限的，程序中如果声明了超过平台要求的对齐值，则按照 C++ 标准该程序是不规范的（ill-formed），这可能会导致未知的编译时或者运行时错误。因此程序员应该定义合理的对齐值，否则可能会遇到一些麻烦。

对齐描述符可以作用于各种数据。具体来说，可以修饰变量、类的数据成员等，而位域（bit field）以及用 register 声明的变量则不可以。我们可以看看 C++11 标准中的这个例子，如

代码清单 8-5 所示。

**代码清单 8-5**

```
alignas(double) void f(); // 错误：alignas 不能修饰函数
alignas(double) unsigned char c[sizeof(double)]; // 正确
extern unsigned char c[sizeof(double)];
alignas(float)
    extern unsigned char c[sizeof(double)]; // 错误：不同对齐方式的变量定义

// 编译选项 :clang++ 8-1-5.cpp -c -std=c++11
```

对于代码清单 8-5 所示的例子，标准给出了建议的答案（如注释所示）。C++11 标准建议用户在声明同一个变量的时候使用同样的对齐方式以免发生意外。不过 C++11 并没有规定声明变量采用了不同的对齐方式就终止编译器的编译。在编写本书时，clang++ 编译器对该例就没有终止编译，而是使用了最严格的对齐方式作为 c 的最终对齐方式。读者可以试试自己的编译环境，看一下编译器是如何处理的。

我们再来看一个例子，这个例子中我们采用了模板的方式来实现一个固定容量但是大小随着所用的数据类型变化的容器类型，如代码清单 8-6 所示。

**代码清单 8-6**

```
#include <iostream>
using namespace std;

struct alignas(alignof(double)*4) ColorVector {
    double r;
    double g;
    double b;
    double a;
};

// 固定容量的模板数组
template <typename T>
class FixedCapacityArray {
public:
    void push_back(T t) { /* 在 data 中加入 t 变量 */}
    // ...
    // 一些其他成员函数、成员变量等
    // ...
    char alignas(T) data[1024] = {0};
    //int length = 1024 / sizeof(T);
};

int main() {
    FixedCapacityArray<char> arrCh;
    cout << "alignof(char): " << alignof(char) << endl;
```

```
    cout << "alignof(arrCh.data): " << alignof(arrCh.data) << endl;

    FixedCapacityArray<ColorVector> arrCV;
    cout << "alignof(ColorVector): " << alignof(ColorVector) << endl;
    cout << "alignof(arrCV.data): " << alignof(arrCV.data) << endl;
    return 1;
}

// 编译选项 :clang++ 8-1-6.cpp -std=c++11
```

代码清单 8-6 修改自代码清单 8-3，在本例中，FixedCapacityArray 固定使用 1024 字节的空间，但由于模板的存在，可以实例化为各种版本。这样一来，我们可以在相同的内存使用量的前提下，做出多种类型（内置或者自定义）版本的数组。

如我们之前提到的一样，为了有效地访问数据，必须使得数据按照其固有特性进行对齐。对于 arrCh，由于数组中的元素都是 char 类型，所以对齐到 1 就行了，而对于我们定义的 arrCV，必须使其符合 ColorVector 的扩展对齐，即对齐到 8 字节的内存边界上。在这个例子中，起到关键作用的代码是下面这一句：

```
char alignas(T) data[1024] = {0};
```

该句指示 data[1024] 这个 char 类型数组必须按照模板参数 T 的对齐方式进行对齐。

编译运行该例子后，可以在实验机上得到如下结果：

```
alignof(char): 1
alignof(arrCh.data): 1
alignof(ColorVector): 32
alignof(arrCV.data): 32
```

如果我们去掉 alignas(T) 这个修饰符，代码清单 8-6 的运行结果会完全不同，具体如下：

```
alignof(char): 1
alignof(arrCh.data): 1
alignof(ColorVector): 32
alignof(arrCV.data): 1
```

可以看到，由于 char 数组默认对齐值为 1，会导致 data[1024] 数组也对齐到 1。这肯定不是编写 FixedCapacityArray 的程序员愿意见到的。

事实上，在 C++11 标准引入 alignas 修饰符之前，这样的固定容量的泛型数组有时可能遇到因为对齐不佳而导致的性能损失（甚至程序错误），这给库的编写者带来了很大的困扰。而引入 alignas 能够解决这些移植性的困难，这可能也是 C++ 标准委员会决定不再“隐藏”变量的对齐方式的原因之一。

C++11 对于对齐的支持并不限于 alignof 操作符及 alignas 描述符。在 STL 库中，还内建了 std::align 函数来动态地根据指定的对齐方式调整数据块的位置。该函数的原型如下：

```
void* align( std::size_t alignment, std::size_t size, void*& ptr, std::size_t& space );
```

该函数在 ptr 指向的大小为 space 的内存中进行对齐方式的调整，将 ptr 开始的 size 大小的数据调整为按 alignment 对齐。我们可以看看代码清单 8-7 所示的这个例子。

**代码清单 8-7**

```
#include <iostream>
#include <memory>
using namespace std;

struct ColorVector {
    double r;
    double g;
    double b;
    double a;
};

int main() {
    size_t const size = 100;
    ColorVector * const vec = new ColorVector[size];

    void* p = vec;
    size_t sz = size;

    void* aligned = align(alignof(double) * 4, size, p, sz);
    if (aligned != nullptr)
        cout << alignof(p) << endl;
}
```

代码清单 8-7 尝试将 vec 中的内容按 alignof(double)*4 的对齐值进行对齐（不过在编写本书的时候，我们的编译器还没有支持 std::align 这个新特性，因此代码清单 8-7 仅供参考）。

事实上，C++11 还在标准库中提供了 aligned_storage 及 aligned_union 供程序员使用。两者的原型如下：

```
template< std::size_t Len, std::size_t Align = /*default-alignment*/ >
struct aligned_storage;
template< std::size_t Len, class... Types >
struct aligned_union;
```

aligned_storage 的第一个参数规定了 aligned_storage 的大小，第二个参数则是其对齐值。我们可以通过代码清单 8-8 所示的这个例子说明它们的用途。

**代码清单 8-8**

```
#include <iostream>
#include <type_traits>
using namespace std;
```

```
    // 一个对齐值为 4 的对象
    struct IntAligned{
      int a;
      char b;
    };

    // 使用 aligned_storage 使其对齐要求更加严格
    typedef  aligned_storage<sizeof(IntAligned),alignof(long  double)>::type
StrictAligned;

    int main() {
      StrictAligned sa;
      IntAligned* pia = new (&sa) IntAligned;
      cout << alignof(IntAligned) << endl;       // 4
      cout << alignof(StrictAligned) << endl;    // 16
      cout << alignof(*pia) << endl;             // 4
      cout << alignof(sa) << endl;               // 16
      return 0;
    }

    // 编译选项 :g++ -std=c++11 8-1-8.cpp
```

在代码清单 8-8 中，我们使用了一个 placement new 来使得 StrictAligned 存储了本来应该只需要按照 4 字节对齐的 IntAligned 对象。虽然 StrictAligned 对象 sa 的内容与 IntAligned 类型指针 pia 所指向的对象完全相同，但通过这样的声明，却产生了比 *pia 更加严格的类型对齐要求（本例中为 16）。因此虽然最后 IntAligned 对象的对齐方式没有发生改变，但实际上却被更严格地对齐了。

有的时候，一个类型声明的代码较长，可能需要程序员上下翻页来阅读（虽然面向对象的规则并不推荐这样做，但在大型项目中，代码很长的类型声明并不少见），通常为了对齐，程序员不得不自己写一些填充来保证其大小。除了代码较难阅读外，每个系统上结构体、联合体的对齐规则也可能不一样，这在代码维护上是一种挑战。如果后加入类型成员的程序员没有注意到这里的对齐要求或者代码编写不慎，很可能会添加了导致对齐改变的代码（对齐变严格一般不是问题，但反之则可能是问题）。

改进的方法可以是使用 alignas 描述符，假如代码清单 8-8 中的 IntAligned 是一个代码很长的 struct 声明，那么对其使用一个 alignas 描述符就是一个可行的方法。不过很多时候，数据的声明是需要共享的，假如超长的 IntAligned 需要在支持和不支持 alignas 的编译环境下共享（典型的，要在老的 C 环境及 C++11 环境下共享头文件），那么使用 aligned_storage 则是一个可行的方法，因为 aligned_storage 可以在产生对象的实例时对对齐方式做出一定的保证。这无疑对“有历史”的代码的重用、维护很有意义。

aligned_union 的用法也基本与此相同。只不过 aligned_union 使用了变长模板参数，程序

员可以根据需要填入多种类型，最后 aligned_union 对象的对齐要求会是多个类型中要求最为严格的一个。

可以看到，在新的 C++11 标准中，对对齐方式的支持是全方面的，无论是查看（alignof）、设定（alignas），还是 STL 库函数（std::align）或是 STL 库模板类型（aligned_storage, aligned_union），程序员都可以找到对应的方法。这使得一些非标准的设定对齐方式的做法规范统一，真正满足程序员在可移植性上的要求。事实上，程序的可移植性还有很多的相关问题，接下来要讲到的通用属性，就与对齐方式有很多关联。

## 8.2 通用属性

☞ **类别：部分人**

### 8.2.1 语言扩展到通用属性

随着 C++ 语言的演化和编译器的发展，人们常会发现标准提供的语言能力不能完全满足要求。于是编译器厂商或组织为了满足编译器客户的需求，设计出了一系列的语言扩展（language extension）来扩展语法。这些扩展语法并不存在于 C++/C 标准中，却有可能拥有较多的用户。有的时候，新的标准也会将广泛使用的语言扩展纳入其中。

扩展语法中比较常见的就是“属性”（attribute）。属性是对语言中的实体对象（比如函数、变量、类型等）附加一些的额外注解信息，其用来实现一些语言及非语言层面的功能，或是实现优化代码等的一种手段。

不同编译器有不同的属性语法。比如对于 g++，属性是通过 GNU 的关键字 __attribute__ 来声明的。程序员只需要简单地声明：

```
__attribute__ ((attribute-list))
```

即可为程序中的函数、变量和类型设定一些额外信息，以便编译器可以进行错误检查和性能优化等。我们可以看看代码清单 8-9 所示的例子。

**代码清单 8-9**

```
extern int area(int n) __attribute__((const));

int main() {
    int i;
    int areas = 0;
    for (i = 0; i < 10; i++) {
        areas += area(3) * i;
    }
}
```

```
// 编译选项 :g++ -c 8-2-1.cpp
```

这里的 const 属性告诉编译器：本函数返回值只依赖于输入，不会改变任何函数外的数据，因此没有任何副作用。在了解该信息的情况下，编译器可以对 area 函数进行优化处理。area(3) 的值只需要计算一次，编译之后可以将 area(3) 视为循环中的常量而只使用其计算结果，从而大大提高了程序的执行性能。

**注意** 事实上，在 GNU 对 C/C++ 的扩展中我们可以看到很多不同的 __attribute__ 属性。常见的如 format、noreturn、const 和 aligned 等，具体含义和用法读者可以参考 GNU 的在线文档 http://gcc.gnu.org/onlinedocs/。

而在 Windows 平台上，我们会找到另外一种关键字 __declspec。__declspec 是微软用于指定存储类型的扩展属性关键字。用户只要简单地在声明变量时加上：

```
__declspec ( extended-decl-modifier )
```

即可设定额外的功能。以对齐方式为例，在 C++11 之前，微软平台的程序员可以使用 __declspec(align(x)) 来控制变量的对齐方式，如代码清单 8-10 所示。

**代码清单 8-10**

```
__declspec(align(32)) struct Struct32 {
    int i;
    double d;
};
```

在代码清单 8-10 中，结构体 Struct32 被对齐到 32 字节的地址边界，其起始地址必须是 32 的倍数。这跟 C++11 中 alignas 的效果是一样的。

**注意** 同样的，微软也定义了很多 __declspec 属性，如 noreturn、oninline、align、dllimport、dllexport 等，具体含义和用法可以参考微软网站上的介绍：http://msdn.microsoft.com/en-US/library/。

事实上，在扩展语言能力的时候，关键字往往会成为一种选择。GNU 和微软只能选择“属性”这样的方式，是为了尽可能避免与用户自定义的名称冲突。同样，在 C++11 标准的设立过程中，也面临着关键字过多的问题。于是 C++11 语言制定者决定增加了通用属性这个特性。不过 C++11 的通用属性设计跟 GNU 和微软都不一样，至少直观地看来，其更加简洁。

### 8.2.2 C++11 的通用属性

C++11 语言中的通用属性使用了左右双中括号的形式：

```
[[ attribute-list ]]
```

这样设计的好处是：既不会消除语言添加或者重载关键字的能力，又不会占用用户空间的关键字的名字空间。

语法上，C++11 的通用属性可以作用于类型、变量、名称、代码块等。对于作用于声明的通用属性，既可以写在声明的起始处，也可以写在声明的标识符之后。而对于作用于整个语句的通用属性，则应该写在语句起始处。而出现在以上两种规则描述的位置之外的通用属性，作用于哪个实体跟编译器具体的实现有关。

我们可以看几个例子。第一个是关于通用属性应用于函数的，具体如下：

```
[[ attr1 ]] void func [[ attr2 ]] ();
```

这里，[[attr1]] 出现在函数定义之前，而 [[attr2]] 则位于函数名称之后，根据定义，[[attr1]] 和 [[attr2]] 均可以作用于函数 [func]。下一个是数组的例子：

```
[[ attr1 ]] int array [[ attr2 ]] [10];
```

这跟第一个例子很类似，根据定义，[[attr1]] 和 [[attr2]] 均可以作用于数组 array。下面这个例子则稍显复杂：

```
[[ attr1 ]] class C [[ attr2 ]] { } [[ attr3 ]] c1 [[ attr4 ]], c2 [[ attr5 ]];
```

这个例子声明了类 C 及其类型的变量 c1 和 c2。本语句中，一共有 5 个不同的属性。按照 C++11 的定义，[[attr1]] 和 [[attr4]] 会作用于 c1，[[attr1]] 和 [[attr5]] 会作用于 c2，[[attr2]] 出现在声明之后，仅作用于类 C，而 [[attr3]] 所作用的对象则跟具体实现有关。

下面是一个 switch-case 加标签的例子：

```
[[ attr1 ]] L1:

switch(value){
    [[ attr2 ]] case 1:  // do something...
    [[ attr3 ]] case 2:  // do something...
    [[ attr4 ]] break;
    [[ attr5 ]] default: // do something...
}

[[ attr6 ]] goto L1;
```

这里，[[attr1]] 作用于标签 L1，[[attr2]] 和 [[attr3]] 作用于 case 1 和 case 2 表达式，[[attr4]] 作用于 break，[[attr5]] 作用于 default 表达式，[[attr6]] 作用于 goto 语句。下面的 for 语句也是类似的：

```
[[ attr1 ]] for( int i = 0; i < top; i++ ) {
   // do something...
}
[[ attr2 ]] return top;
```

这里，[[attr1]] 作用于 for 表达式，[[attr2]] 作用于 return。下面是函数有参数的情况：

```
[[ attr1 ]] int func([[ attr2 ]] int i, [[ attr3 ]] int j)
{
    // do something
    [[ attr4 ]] return i + j;
}
```

[[attr1]] 作用于函数 func，[[attr2]] 和 [[attr3]] 分别作用于整型参数 i 和 j，[[attr4]] 作用于 return 语句。

事实上，在现有 C++11 标准中，只预定义了两个通用属性，分别是 [[ noreturn ]] 和 [[carries_dependency ]]。而在 C++11 标准委员会的最初提案中，还包含了形如 [[ final ]]、[[ override ]]、[[ restrict ]]、[[ hides ]]、[[ base_check ]] 等通用属性。不过最终，标准委员会只通过了以上两个，原因大概有以下几点：

- final、override、restrict 等是 C++ 语言中需要支持的语言特性。通用属性从设计上讲，是可忽略的属性，其设计的目的主要是为了帮助编译器更好地检查代码中的错误或帮助编译器更好地优化代码。因此，语义相关的部分还是需要使用在关键字上。
- 预定义的通用属性应该是可移植的。一旦预定义了过多的通用属性，会导致 C++ 代码的可移植性变弱。
- C 语言是没有通用属性的。

虽然看起来通用属性的使用受到了一些限制，但至少其语法规则为编译器厂商或组织提供了实现不同属性的办法。

### 8.2.3 预定义的通用属性

如上文所述，C++11 预定义的通用属性包括 [[ noreturn ]] 和 [[ carries_dependency ]] 两种。

[[ noreturn ]] 是用于标识不会返回的函数的。这里必须注意，不会返回和没有返回值的（void）函数的区别。没有返回值的 void 函数在调用完成后，调用者会接着执行函数后的代码；而不会返回的函数在被调用完成后，后续代码不会再被执行。

[[noreturn]] 主要用于标识那些不会将控制流返回给原调用函数的函数，典型的例子有：有终止应用程序语句的函数、有无限循环语句的函数、有异常抛出的函数等。通过这个属性，开发人员可以告知编译器某些函数不会将控制流返回给调用函数，这能帮助编译器产生更好的警告信息，同时编译器也可以做更多的诸如死代码消除、免除为函数调用者保存一些特定寄存器等代码优化工作。

我们可以看看代码清单 8-11 所示的这个例子。

**代码清单 8-11**

```
void DoSomething1();
void DoSomething2();

[[ noreturn ]] void ThrowAway() {
    throw "expection"; // 控制流跳转到异常处理
}

void Func(){
    DoSomething1();
    ThrowAway();
    DoSomething2(); // 该函数不可到达
}

// 编译选项 :clang++ -std=c++11 -c 8-2-3.cpp
```

在代码清单 8-11 中，由于 ThrowAway 抛出了异常，DoSomething2 永远不会被执行。这个时候将 ThrowAway 标记为 noreturn 的话，编译器会不再为 ThrowAway 之后生成调用 DoSomething2 的代码。当然，编译器也可以选择为 Func 函数中的 DoSomething2 做出一些警告以提示程序员这里有不可到达的代码。

不返回的函数除了是有异常抛出的函数外，还有可能是有终止应用程序语句的函数，或是有无限循环语句的函数等。事实上，在 C++11 的标准库中，我们都能看到形如：

```
[[noreturn]] void abort(void) noexcept;
```

这样的函数声明。这里声明的是最常见的 abort 函数。abort 总是会导致程序运行的停止，甚至连自动变量的析构函数以及本该在 atexit() 时调用的函数全都不调用就直接退出了。因此声明为 [[noreturn]] 是有利于编译器优化的。

不过程序员还是应该小心使用 [[ noreturn ]]，也尽量不要对可能会有返回值的函数使用 [[noreturn]]。代码清单 8-12 所示的是一个错误使用 [[ noreturn ]] 的例子。

**代码清单 8-12**

```
#include <iostream>
using namespace std;

[[ noreturn ]] void Func(int i){
    // 当参数 i 的值为 0 时，该函数行为不可估计
    if (i < 0)
        throw "negative";
    else if (i > 0)
        throw "positive";
}

int main(){
```

```
    Func(0);
    cout << "Returned" << endl; // 无法执行该句
    return 1;
}

// 编译选项 :clang++ -std=c++11 8-2-4.cpp
```

代码清单 8-12 的例子中，Func 调用后的打印语句永远不会被执行，因为 Func 被声明为 [[ noreturn ]]。不过由于函数作者的疏忽，忘记了 i == 0 时的状况，因此在 i == 0 时，Func 运行结束时还是会返回 main 的。在我们的实验机上，编译运行该例子会在运行时发生“段错误”。当然，具体的错误情况可能会根据编译器和运行时环境的不同而有所不同。不过总的来说，程序员必须审慎使用 [[ noreturn ]]。

另外一个通用属性 [[ carries_dependency ]] 则跟并行情况下的编译器优化有关。事实上，[[carries_dependency]] 主要是为了解决弱内存模型平台上使用 memory_order_consume 内存顺序枚举问题。

如我们在第 6 章里讲到的，memory_order_consume 的主要作用是保证对当前原子类型数据的读取操作先于所有之后关于该原子变量的操作完成，但它不影响其他原子操作的顺序。要保证这样的“先于发生”的关系，编译器往往需要根据 memory_model 枚举值在原子操作间构建一系列的依赖关系，以减少在弱一致性模型的平台上产生内存栅栏。不过这样的关系则往往会由于函数的存在而被破坏。比如下面的代码：

```
atomic<int*> a;
...
int* p = (int *)a.load(memory_order_consume);
func(p);
```

上面的代码中，编译器在编译时可能并不知道 func 函数的具体实现，因此，如果要保证 a.load 先于任何关于 a（或是 p）的操作发生，编译器往往会在 func 函数之前加入一条内存栅栏。然而，如果 func 的实现是：

```
void func(int * p) {
   // ... 假设 p2 是一个 atomic<int*> 的变量
   p2.store(p, memory_order_release)
}
```

那么对于 func 函数来说，由于 p2.store 使用了 memory_order_release 的内存顺序，因此，p2.store 对 p 的使用会被保证在任何关于 p 的使用之后完成。这样一来，编译器在 func 函数之前加入的内存栅栏就变得毫无意义，且影响了性能。同样的情况也会发生在函数返回的时候。

而解决的方法正是使用 [[carries_dependency]]。该通用属性既可以标识函数参数，又可以标识函数的返回值。当标识函数的参数时，它表示数据依赖随着参数传递进入函数，即不需要产生内存栅栏。而当标识函数的返回值时，它表示数据依赖随着返回值传递出函数，同

样也不需要产生内存栅栏。更具体的我们可以看看代码清单 8-13 所示的例子。

**代码清单 8-13**

```
#include <iostream>
#include <atomic>
using namespace std;

atomic<int*> p1;
atomic<int*> p2;
atomic<int*> p3;
atomic<int*> p4;

void func_in1(int* val) {
    cout << *val << endl;
}

void func_in2(int* [[carries_dependency]] val) {
    p2.store(val, memory_order_release);
    cout << *p2 << endl;
}

[[carries_dependency]] int* func_out() {
    return (int *)p3.load(memory_order_consume);
}

void Thread() {
    int* p_ptr1 = (int *)p1.load(memory_order_consume);      // L1
    cout << *p_ptr1 << endl;// L2

    func_in1(p_ptr1);          // L3
    func_in2(p_ptr1);          // L4

    int * p_ptr2 = func_out();  // L5
    p4.store(p_ptr2, memory_order_release); // L6
    cout << *p_ptr2 << endl;
}

// 编译选项 :g++ -std=c++11 8-2-5.cpp -c
```

在代码清单 8-13 中，L1 句中，p1.load 采用了 memory_order_consume 的内存顺序，因此任何关于 p1 或者 p_ptr1 的原子操作，必须发生在 L1 句之后。这样一来，L2 将由编译器保证其执行必须在 L1 之后（通过编译器正确的指令排序和内存栅栏）。而当编译器在处理 L3 时，由于 func_in1 对于编译器而言并没有声明 [[ carries_dependency ]] 属性，编译器则可能采用保守的方法，在 func_in1 调用表达式之前插入内存栅栏。而编译器在处理 L4 句时，由于函数 func_in2 使用了 [[ carries_dependency ]]，编译器则会假设函数体内部会正确地处理内存顺序，因此不再产生内存栅栏指令。事实上 func_in2 中也由于 p2.store 使用了内存顺序

memory_order_release，因而不会产生任何的问题。而当编译器处理 L5 句时，由于 func_out 的返回值使用了 [[ carries_dependency ]]，编译器也不会在返回前为 p3.load(memory_order_consume) 插入内存栅栏指令去保证正确的内存顺序。而在 L6 行中，我们看到 p4.store 使用了 memory_order_release，因此 func_out 不产生内存栅栏也是毫无问题的。

事实上，本书编写时 [[carries_dependency]] 还没有被编译器支持，而对一些强内存模型的平台来说，编译器也常常会忽略该通用属性，因此其可用性比较有限。不过与 [[noreturn]] 相同的是，[[carries_dependency]] 只是帮助编译器进行优化，这符合通用属性设计的原则。当读者使用的平台是弱内存模型的时候，并且很关心并行程序的执行性能时，可以考虑使用 [[carries_dependency]]。

## 8.3 Unicode 支持

☞ 类别：所有人

### 8.3.1 字符集、编码和 Unicode

在了解 Unicode 之前，我们先回顾一下计算机表示信息的方式。无论是存储器中的晶体管通断，还是磁盘中磁畴的极性，或者是光盘中的坑槽，计算机总是使用两种不同的状态来作为基本信息，即二进制信息。而要标识现实生活中更为复杂的实体，则需要通过多个这样的基本信息的组合来完成。在计算机中，首当其冲需要被标识的就是字符。为了使二进制组合标识字符的方法在不同设计的计算机间通用，就迫切需要统一的字符编码方法。于是在 20 世纪 60 年代的时候，现在使用最为广泛的 ASCII 字符编码就出现了。

在 ANSI 颁布的标准中，基本 ASCII 的字符使用了 7 个二进制位进行标识，这意味着总共可以标识 128 种不同的字符。这对英文字符（以及一些控制字符、标点符号等）来说绰绰有余，不过随着计算机在全世界的普及，非字符构成的语言（如中文）也需要得到支持，128 个字符对于全世界众多语言而言就显得力不从心了。

到了 20 世纪 90 年代，ISO 与 Unicode 两个组织共同发布了能够唯一地表示各种语言中的字符的标准。通常情况下，我们将一个标准中能够表示的所有字符的集合称为字符集。通常，我们称 ISO/Unicode 所定义的字符集为 Unicode。在 Unicode 中，每个字符占据一个码位（Code point）。Unicode 字符集总共定义了 1 114 112 个这样的码位，使用从 0 到 10FFFF 的十六进制数唯一地表示所有的字符。不过不得不提的是，虽然字符集中的码位唯一，但由于计算机存储数据通常是以字节为单位的，而且出于兼容之前的 ASCII、大数小段数段、节省存储空间等诸多原因，通常情况下，我们需要一种具体的编码方式来对字符码位进行存储。比较常见的基于 Unicode 字符集的编码方式有 UTF-8、UTF-16 及 UTF-32（一般人常常把 UTF-16 和 Unicode 混为一谈，在阅读各种资料的时候读者要注意区别）。

以 UTF-8 为例，其采用了 1~6 字节的变长编码方式编码 Unicode，英文通常使用 1 字节表示，且与 ASCII 是兼容的，而中文常用 3 字节进行表示。UTF-8 编码由于较为节约存储空间，因此使用得比较广泛。表 8-1 所示就是 UTF-8 的编码方式。

**表 8-1　UTF-8 的编码方式**

| Unicode 符号范围（十六进制） | UTF-8 编码方式（二进制） |
| --- | --- |
| 0000 0000—0000 007F | 0xxxxxxx |
| 0000 0080—0000 07FF | 110xxxxx 10xxxxxx |
| 0000 0800—0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx |
| 0001 0000—0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx |

**注意**　事实上，现行桌面系统中，Windows 内部采用了 UTF-16 的编码方式，而 Mac OS、Linux 等则采用了 UTF-8 编码方式。

除了基于 Unicode 字符集的 UTF-8、UTF-16 等编码外，在中文语言地区，我们还有一些常见的字符集及其编码方式，GB2312、Big5 就是其中影响最大、使用最广泛的两种。

GB2312 的出现先于 Unicode。早在 20 世纪 80 年代，GB2312 作为简体中文的国家标准被颁布使用。GB2312 字符集收入 6763 个汉字和 682 个非汉字图形字符，而在编码上，是采用了基于区位码的一种编码方式，采用 2 字节表示一个中文字符。GB2312 在中国大陆地区及新加坡都有广泛的使用。

而 BIG5 则常见于繁体中文，俗称“大五码”。BIG5 是长期以来的繁体中文的业界标准，共收录了 13 060 个中文字，也采用了 2 字节的方式来表示繁体中文。BIG5 在中国台湾、香港、澳门等地区有着广泛的使用。

**扩展　关于内码和交换码**

内码实际就是字符在计算机存储单元中的二进制表示，在早期中文字符编码混乱的时候，内码和交换码等概念就产生了。每一种二进制表示的中文的编码都被认为是一种内码，而在有多种内码的情况下，交换码被设计为协调不同的内码间数据交换的手段。依照这种认知方式，UTF-8 等编码即是内码，也是交换码。随着时代的发展和各种标准的先后制定，内码和交换码的概念也在被逐渐淡化，因为通常情况下，两者总是一致的。

不同的编码方式对于相同的二进制字符串的解释是不同的。常见的，如果一个 UTF-8 编码的网页中的字符串按照 GB2312 编码进行显示，就会出现乱码。而 BIG5 和 GB2312 之间的乱码则在中文地区软件中有着“悠久”的历史。不过随着 Unicode 的使用和发展，以及软件系统对多种编码的支持，程序发生乱码的现象也越来越少。总的说来，Unicode 还在其发展期，Unicode、GB2312 以及 BIG5 等多种编码共存的状况可能在以后较长的时间内都会持

续下去。

### 8.3.2 C++11 中的 Unicode 支持

在 C++98 标准中，为了支持 Unicode，定义了“宽字符”的内置类型 wchar_t。不过不久程序员便发现 C++ 标准对 wchar_t 的“宽度”显然太过容忍，在 Windows 上，多数 wchar_t 被实现为 16 位宽，而在 Linux 上，则被实现为 32 位。事实上，C++98 标准定义中，wchar_t 的宽度是由编译器实现决定的。理论上，wchar_t 的长度可以是 8 位、16 位或者 32 位。这样带来的最大的问题是，程序员写出的包含 wchar_t 的代码通常不可移植。

这一状况在 C++11 中得到了一定的改善，至少 C++11 解决了 Unicode 类型数据的存储问题。C++11 引入以下两种新的内置数据类型来存储不同编码长度的 Unicode 数据。

- char16_t：用于存储 UTF-16 编码的 Unicode 数据。
- char32_t：用于存储 UTF-32 编码的 Unicode 数据。

至于 UTF-8 编码的 Unicode 数据，C++11 还是使用 8 字节宽度的 char 类型的数组来保存。而 char16_t 和 char32_t 的长度则犹如其名称所显示的那样，长度分别为 16 字节和 32 字节，对任何编译器或者系统都是一样的。

此外，C++11 还定义了一些常量字符串的前缀。在声明常量字符串的时候，这些前缀声明可以让编译器使字符串按照前缀类型产生数据。事实上，C++11 一共定义了 3 种这样的前缀：

- u8 表示为 UTF-8 编码。
- u 表示为 UTF-16 编码。
- U 表示为 UTF-32 编码。

3 种前缀对应于 3 种不同的 Unicode 编码。一旦声明了这些前缀，编译器会在产生代码的时候按照相应的编码方式存储。以上 3 种前缀加上基于宽字符 wchar_t 的前缀“L”，及不加前缀的普通字符串字面量，算来在 C++11 中，一共有了 5 种方式来声明字符串字面量，其中 4 种是前缀表达的。

通常情况下，按照 C/C++ 的规则，连续在代码中声明多个字符串字面量，则编译器会自动将其连接起来。比如 "a" "b" 这样声明的方式与 "ab" 的声明方式毫无区别。而一旦连续声明的多个字符串字面量中的某一个是前缀的，则不带前缀的字符串字面量会被认为与带前缀的字符串字面量是同类型的。比如声明 u"a" "b" 和 "a" u"b"，其效果跟 u"ab" 是完全等同的，都是生成了连续的字面量等于 UTF-16 编码 "ab" 的字符串。不过最好不要将各种前缀字符串字面量连续声明，因为标准定义除了 UTF-8 和宽字符字符串字面量同时声明会冲突外，其他字符串字面量的组合最终会产生什么结果，以及会按照什么类型解释，是由编译器实现自行决定的。因此应该尽量避免这种不可移植的字符串字面量声明方式。

对于 Unicode 编码字符的书写，C++11 中还规定了一些简明的方式，即在字符串中用 '\u' 加 4 个十六进制数编码的 Unicode 码位（UTF-16）来标识一个 Unicode 字符。比如 '\u4F60' 表示的就是 Unicode 中的中文字符“你”，而 '\u597D' 则是 Unicode 中的“好”。此外，也可以通过 '\U' 后跟 8 个十六进制数编码的 Unicode 码位（UTF-32）的方式来书写 Unicode 字面常量。程序员获得 Unicode 码位的编码的方法很多，比如在 Windows 系统下，可以使用系统自带的字符映射表，而在网络上，也可以轻松地找到很多免费提供的中文到 Unicode 的在线转换服务的网站。

我们可以来看一下代码清单 8-14 所示的这个例子。

**代码清单 8-14**

```
#include <iostream>
using namespace std;

int main(){
    char utf8[] = u8"\u4F60\u597D\u554A";
    char16_t utf16[] = u"hello";
    char32_t utf32[] = U"hello equals \u4F60\u597D\u554A";

    cout << utf8 << endl;
    cout << utf16 << endl;
    cout << utf32 << endl;

    char32_t u2[] = u"hello";    // Error
    char u3[] = U"hello";        // Error
    char16_t u4 = u8"hello";     // Error
}

// 编译选项 :clang++ 8-3-1.cpp -std=c++11
```

在本例中，我们声明了 3 种不同类型的 Unicode 字符串 utf8、utf16 和 utf32。由于无论对哪种 Unicode 编码，英文的 Unicode 码位都相同，因此只有非英文使用了 "\u" 的码位方式来标志。我们可以看到，一旦使用了 Unicode 字符串前缀，这个字符串的类型就确定了，仅能放在相应类型的数组中。u2、u3、u4 就是因为类型不匹配而不能通过编译。

如果我们注释掉不能通过的定义，编译并运行代码清单 8-14，在我们的实验机上可以得到以下输出：

```
你好啊
0x7fffaf087390
0x7fffaf087340
```

对应于 char utf8[] = u8"\u4F60\u597D\u554A" 这句，该 UTF-8 字符串对应的中文是“你好啊”。而对于 utf16 和 utf32 变量，我们本来期望它们分别输出“hello”及“hello equals 你

好啊”。不过实验机上我们都只得到了一串数字输出。这是什么原因呢?

事实上，C++11 虽然在语言层面对 Unicode 进行了支持，但语言层面并不是唯一的决定因素。用户要在自己的系统上看到正确的 Unicode 文字，还需要输出环境、编译器，甚至是代码编辑器等的支持。我们可以按照编写代码、编译、运行的顺序来看看它们对整个 Unicode 字符串输出的影响。

首先会影响 Unicode 正确性的过程是源文件的保存。以字符 "\u4F60" 为例，其保证的是输入数据等同于 Unicode 中码位为 4F60 的字符，而被保存的源代码文件中，数据采用的编码则跟编辑器有关。如编辑器采用了 GB2312 编码保存数据，则源代码文件中 utf8 变量的前 2 字节保存的是 GB2312 编码的中文字“你”。而如果编辑器采用了 UTF-8 编码，则源代码文件中的 utf8 变量的前 3 字节保存的是 UTF-8 的中文字“你”。

第二个会影响 Unicode 正确性的过程是编译。C++11 中的 u8 前缀保证编译器把 utf8 变量中的数据以 UTF-8 的形式产生在目标代码的数据段中。不过通常编译器也会有自己的设定，如果编译器被设置了正确的编码形式，(比如文件保存为 GB2312 编码，编译器也设置了文件格式为 GB2312，或者两者均为 UTF-8)，则 u8 前缀能够正常工作。

第三个会影响 Unicode 正确性的过程是输出。C++ 的操作符“<<”保证把数据以字节(char)、双字(char16_t)、四字(char32_t)的方式输出到输出设备，但输出设备(比如在 Linux 下的 shell，或是 Windows 下的 console)是否能够支持该编码类型的输出，则取决于设备驱动等软件层。

我们的实验机是一台 Linux 机器。对于 Linux 而言，大多数软件如 shell、编辑器 vi，以及编译器 g++ 等都会根据 Linux 系统 locale 设定而采用 UTF-8 编码。在代码清单 8-14 所示的例子中，utf8 变量会输出正确，而 utf16、utf32 数据输出均失败，原因就是因为系统并不支持 UTF-16 和 UTF-32 输出。

在现有的编程环境支持下，如果要保证在程序中直接输入中文得到正确的输出，我们建议程序员要使用与系统环境中相同的编码方式。比如在 Linux 下(现在很多 Linux 系统的发布版均使直接用 UTF-8 作为系统中的编码)，u8 前缀的 UTF-8 编码 Unicode 会得到广泛的支持。而 Windows 由于内部采用了 UTF-16 的方式保存文字编码，因此 u 前缀的 UTF-16 编码的 Unicode 可能会被支持得更好。而如果程序员想在不同系统下编译相同的文件(这也并不少见，比如在一些基于 QT IDE 的跨平台开发上，程序员会在各平台间共享源代码)，程序员则应该注意查看编辑器与编译器是否使用了不同的编码方式，并按需调整。

如果在用户确认了使用环境没有问题，在程序员排除了上述环境上的困难之后，又有了 char16_t、char32_t 以及各种前缀表示、\u 字面值等，是否意味着 Unicode 真的就可以良好运作了呢?让我们来看看代码清单 8-15 所示的这个的例子。

**代码清单 8-15**

```
#include <iostream>
using namespace std;

int main() {
    char utf8[] = u8"\u4F60\u597D\u554A";
    char16_t utf16[] = u"\u4F60\u597D\u554A";


    cout << sizeof(utf8) << endl;        // 10 字节
    cout << sizeof(utf16) << endl;       // 8 字节

    cout << utf8[1] << endl;     // 输出不可见字符
    cout << utf16[1] << endl;    // 输出 22909(0x597D)
}

// 编译选项 :g++ -std=c++118-3-2.cpp
```

这个例子里，我们首先看不同编码情况下 Unicode 字符串的大小。可以看到，UTF-8 由于采用了变长编码，在这里把每个中文字符编码为 3 字节，再加上 '\0' 的字符串终止符，所以 utf8 变量的大小是 10 字节。而 UTF-16 则是定长编码，所以 utf16 占用了 8 字节空间。倘若我们按照使用 ASCII 字符的思路来使用 Unicode 字符，比如使用数组来访问的时候，我们发现 utf8 的输出是不正确的（这里的 utf16 是正确的，只是实验机无法正常输出）。事实上，我们将 UTF-8 编码的数据放在了一个 char 类型中，所以 utf8[1] 只是指向了第一个 UTF-8 字符 3 字节中的第二位，因此输出不正常。

相比于定长编码的 UTF-16，变长编码的 UTF-8 的优势在于支持更多的 Unicode 码位，而且也没有大数端小端段问题（而有字节序问题的 UTF-16 有 LE 和 BE 两种不同版本）。不过不能直接数组式访问是 UTF-8 的最大的缺点。此外，C++11 为 char16_t 和 char32_t 分别配备了 u16string 及 u32string 等字符串类型，却没有 u8string（因为从实现上讲，变长的 UTF-8 编码的数据也不是很容易与 string 配合使用）。这样一来，UTF-8 的字符串不能够被方便地进行增删、查找，至于利用各种高级的 STL 算法，就更加困难了。

倘若用户要完成上面的各种复杂的操作，需要的是一个复杂的类型，比如说用 utf8_t 的类型来保存变长的 UTF-8 字符，而不是像现在这样用 char 数组来“存放”UTF-8 字符。这个想法固然也有一些道理，但 utf8_t 类型给 C++ 带来的冲击可能也是很大的，因为它看起来像是个基本类型，却是变长的，与已有算法结合并不一定有性能上的优势（比如计算第 $N$ 个元素的时间复杂度不再是 O(1)）。

UTF-8 变长的设定更多时候是为了在序列化时节省存储空间，定长的 UTF-16 编码或者 UTF-32 则更适合在内存环境中操作。因此，在现有 C++ 编程中，总是倾向于在 I/O 读写的时候才采用 UTF-8 编码，即在进行 I/O 操作时才将定长 Unicode 编码转化为 UTF-8 使用。内

存中一直操作的是定长的 Unicode 编码，故不过在这种使用方式下，编码转换就成了更加常用且不可或缺的功能。

### 8.3.3 关于 Unicode 的库支持

C++11 在标准库中增加了一些 Unicode 编码转换的支持。由于 char16_t 及 char32_t 也是 C11 标准中新增的类型，所以 C 库及 C++ 库均有一些不同的实现。

首先我们可以看一些比较直观的编码转换函数。在 C11 中，程序员可以使用库中的一些新增的编码转换函数来完成各种 Unicode 编码间的转换。函数的原型如下：

```
size_t mbrtoc16(char16_t * pc16, const char * s, size_t n, mbstate_t * ps);
size_t c16rtomb(char * s, char16_t c16, mbstate _t * ps);
size_t mbrtoc32(char32_t * pc32, const char * s, size_t n, mbstate_t * ps);
size_t c32rtomb(char * s, char32_t c32, mbstate_t * ps);
```

上述代码中，字母 mb 是 multi-byte（这里指多字节字符串，后面会解释）的缩写，c16 和 c32 则是 char16 和 char32 的缩写，rt 是 convert（转换）的缩写。代码中的几个函数原型大同小异，目的就是完成多字节字符串、UTF-16 及 UTF-32 之间的一些转换。除了 mbstate_t 是用于返回转换中的状态信息外，其余部分意义比较明显，读者应该能直观理解它们的含义。代码清单 8-16 所示是一个可能通过编译的例子。

**代码清单 8-16**

```
#include <iostream>
#include <cuchar>
using namespace std;

int main() {
    char16_t utf16[] = u"\u4F60\u597D\u554A";
    char mbr[sizeof(utf16)*2] = {0};     // 这里我们假设 buffer 这么大就够了
    mbstate_t s;

    c16rtomb(mbr, utf16, &s);
    cout << mbr << endl;
}
// 编译选项 :g++ -std=c++11 -c 8-3-3.cpp
```

使用 C11 中编码转换函数需要 include 头文件 <cuchar>。不过在本书写作的时候，我们使用的编译器都还没能提供这个头文件及其实现。所以代码清单 8-16 所示的例子仅供参考。

C++ 对字符转换的支持则稍微复杂一点，不过 C++ 对编码转换支持的新方法都需要源自于 C++ 的 locale 机制的支持[⊖]。事实上，locale 的概念在 POSIX 中就用，在 C++ 中，通常情况下，locale 描述的是一些必须知道的区域特征，如程序运行的国家 / 地区的数字符号、日期

⊖ 可以参考该文理解 C++ 的 locale 机制：http://www.cantrip.org/locale.html。

表示、钱币符号等。比如在美国地区且采用了英文和 UTF-8 编码，这样的 locale 可以表示为 en_US.UTF-8，而在中国使用简体中文并采用 GB2312 文字编码的 locale 则可以被表示为 zh_CN.GB2312，等等。

通常知道了一个地区的 locale，要使用不同的地区特征，则需访问该 locale 的一个 facet。facet 可以简单地理解为是 locale 的一些接口。比如对于所有的 locale 都会有 num_put/num_get 的操作，那么这些操作就是针对该 locale 数值存取的接口，即该 locale 情况下数值存取的 facet。在 C++ 中常见的 facet 除去 num_get/num_put、money_get/money_put 等外，还有一种就是 codecvt。

codecvt 从类型上来讲是一个模板类，从功能上讲，是一种能够完成从当前 locale 下多字符编码字符串到多种 Unicode 字符编码转换（也包括 Unicode 字符编码间的转换）的 facet。这里的多字节字符串不仅可以是 UTF-8，也可以是 GB2312 或者其他，其实际依赖于 locale 所采用的编码方式。在 C++ 标准中，规定一共需要实现 4 种这样的 codecvt facet ⊖。

```
std::codecvt<char, char, std::mbstate_t>          // 完成多字节与 char 之间的转换
std::codecvt<char16_t, char, std::mbstate_t>      // 完成 UTF-16 与 UTF-8 间的转换
std::codecvt<char32_t, char, std::mbstate_t>      // 完成 UTF-32 与 UTF-8 间的转换
std::codecvt<wchar_t, char, std::mbstate_t>       // 完成多字节与 wchar_t 之间的转换
```

每种 facet 负责不同类型编码数据的转换。值得注意的是，现行编译器支持情况下，一种 locale 并不一定支持所有的 codecvt 的 facet。程序员可以通过 has_facet 来查询该 locale 在本机上的支持情况，如代码清单 8-17 所示。

**代码清单 8-17**

```
#include <iostream>
#include <locale>
using namespace std;

int main(){
    // 定义一个 locale 并查询该 locale 是否支持一些 facet
    locale lc("en_US.UTF-8");
    bool can_cvt = has_facet<codecvt<wchar_t, char, mbstate_t>>(lc);
    if (!can_cvt)
        cout << "Do not support char-wchar_t facet!" << endl;

    can_cvt = has_facet<codecvt<char16_t, char, mbstate_t>>(lc);
    if (!can_cvt)
        cout << "Do not support char-char16 facet!" << endl;

    can_cvt = has_facet<codecvt<char32_t, char, mbstate_t>>(lc);
    if (!can_cvt)
        cout << "Do not support char-char32 facet!" << endl;
```

⊖ 参见 http://en.cppreference.com/w/cpp/locale/codecvt。

```
    can_cvt = has_facet<codecvt<char, char, mbstate_t>>(lc);
    if (!can_cvt)
        cout << "Do not support char-char facet!" << endl;

    return 0;
}

// 编译选项 :g++ -std=c++11 8-3-4.cpp
```

编译运行代码清单 8-17，在我们的实验机环境及编译器支持情况下，可以得到以下结果：

```
Do not support char-char16 facet!
Do not support char-char32 facet!
```

由上述结果可知，从 char 到 char16 或 char32 转换的两种 facet 还没有被支持（实验机使用的编译器尚未支持）。

而在使用 facet 上，用户并不需要显式地在代码中生成 codecvt 对象。比如在对 C++11 中 stream 进行 I/O 时，我们只需要一些简单的设定，就可以让 stream 自动进行一些编码的转换。我们看一下代码清单 8-18 所示的例子[⊖]。

**代码清单 8-18**

```
#include <iostream>
#include <fstream>
#include <string>
#include <locale>
#include <iomanip>
using namespace std;

int main()
{
    // UTF-8 字符串 , "\x7a\xc3\x9f\xe6\xb0\xb4\xf0\x9d\x84\x8b";
    ofstream("text.txt") << u8"z\u00df\u6c34\U0001d10b";

    wifstream fin("text.txt");
    // 该 locale 的 facet - codecvt<wchar_t, char, mbstate_t>
    // 可以将 UTF-8 转化为 UTF-32
    fin.imbue(locale("en_US.UTF-8"));

    cout << "The UTF-8 file contains the following wide characters: \n";
    for(wchar_t c; fin >> c; )
        cout << "U+" << hex << setw(4) << setfill('0') << c << '\n';
}

// 编译选项 :g++ -std=c++11 8-3-5.cpp
```

⊖ 本例来源于 http://en.cppreference.com/w/cpp/locale/codecvt，仅做了注释上的修改。

在代码清单 8-18 中，我们使用了 wifstream 来打开一个 UTF-8 编码的文件。随后调用了这个 wifstream 的 imbue 函数，为其设定了一个为 en_US.UTF-8 的 locale。这样一来当进行 I/O 操作的时候，会使用完成 UTF-8 到 UTF-32 编码转换的 facet（codecvt<wchar_t, char, mbstate_t>）来完成编码转换。编译运行代码清单 8-18，我们就可以看到定义的 Unicode 字符串的十六进制表示。

```
The UTF-8 file contains the following wide characters:
U+007a
U+00df
U+6c34
U+1d10b
```

codecvt 还派生一些形如 codecvt_utf8、codecvt_utf16、codecvt_utf8_utf16 等可以用于字符串转换的模板类。这些模板类配合 C++11 定义的 wstirng_convert 模板，可以进行一些不同字符串的转换。代码清单 8-19 也是一个 C++11 标准中的示例，不过由于我们编译器尚未支持，所以也仅供参考。

**代码清单 8-19**

```
#include <cvt/wstring>
#include <codecvt>
#include <iostream>
using namespace std;
int main() {
    wstring_convert<codecvt_utf8<wchar_t>> myconv;
    string mbstring = myconv.to_bytes(L"Hello\n");
    cout << mbstring;
}
```

除了 to_bytes 外，wstring_convert 还支持使用 from_bytes 来完成逆向的编码转换。更多关于 wstring_convert、locale、codecvt 的内容，读者可以参看一些在线文档，这里不再展开描述。

此外，还有一点值得注意，在 C++98 标准定义 wchar_t 类型的时候，为其添加了新的 fstream 类型，如 wifstream 及 wofstream 等。不过 C++11 标准并没有为 char16_t 及 char32_t 再次产生 fstream 对象。关于这点，跟前面提到的 UTF-8 操作问题有类似。标准委员会意识到在 Unicode 在序列化存储时很少是 UTF-16 或者是 UTF-32 的（空间太过浪费）。所以从实际情况出发，程序员可以利用不同的 codecvt 的 facet 来将 UTF-8 编码存储的字符与不同的 Unicode 进行转换，而不必直接将 UTF-16 和 UTF-32 编码的字符存储到文件，基于此，也就没在 C++11 标准中提供支持该功能的 u16ifstream、u32ofstream 等。

事实上，尽管 C++11 对 Unicode 做了更多的支持，Unicode 字符串的使用仍然比 ASCII 字符复杂。如我们所见的，程序在进行各种 I/O 操作时，往往需要 UTF-8 编码的字符。程序

员如果想直接在内存中操作 UTF-8 编码字符，那么对 UTF-8 字符串的 string 进行遍历、插入、删除、查找等操作会比较困难。如果遇到这样的情况，程序员可以自行寻求一些第三方库的支持。

## 8.4 原生字符串字面量

**☞类别：所有人**

原生字符串字面量（raw string literal）并不是一个新鲜的概念，在许多编程语言中，我们都可以看到对原生字符串字面量的支持。原生字符串使用户书写的字符串“所见即所得”，不再需要如 '\t'、'\n' 等控制字符来调整字符串中的格式，这对编程语言的学习和使用都是具有积极意义的。

顺应这个潮流，在 C++11 中，终于引入了原生字符串字面量的支持。C++11 中原生字符串的声明相当简单，程序员只需要在字符串前加入前缀，即字母 R，并在引号中用使用括号左右标识，就可以声明该字符串字面量为原生字符串了。请看下面的例子，如代码清单 8-20 所示。

**代码清单 8-20**

```
#include <iostream>
using namespace std;

int main(){
    cout << R"(hello,\n
         world)" << endl;
    return 0;
}


// 编译选项 : g++ 8-1-2.cpp -std=c++11
```

代码清单 8-20 的输出如下，可以看到 '\n' 并没有被解释为换行。

```
hello,\n
         world
```

而对于 Unicode 的字符串，也可以通过相同的方式声明。声明 UTF-8、UTF-16、UTF-32 的原生字符串字面量，将其前缀分别设为 u8R、uR、UR 就可以了。不过有一点需要注意，使用了原生字符串的话，转义字符就不能使用了，这会给想使用 \u 或者 \U 的方式写 Unicode 字符的程序员带来一定影响。下面来看代码清单 8-21 所示的例子。

**代码清单 8-21**

```
#include <iostream>
```

```
using namespace std;

int main(){
    cout << u8R"(\u4F60,\n
         \u597D)" << endl;
    cout << u8R"(你好)" << endl;
    cout << sizeof(u8R"(hello)") << "\t" << u8R"(hello)" << endl;
    cout << sizeof(uR"(hello)") << "\t" << uR"(hello)" << endl;
    cout << sizeof(UR"(hello)") << "\t" << UR"(hello)" << endl;
    return 0;
}

// 编译选项:g++ -std=c++11 8-4-2.cpp
```

编译运行代码清单 8-21，可以得到以下结果：

```
\u4F60,\n
         \u597D
你好
6       hello
12      0x400be6
24      0x400bf4
```

可以看到，当程序员试图使用 \u 将数字转义为 Unicode 的时候，原生字符串会保持程序员所写的字面值，所以这样的企图并不能如愿以偿。而借助文本编辑器直接输入中文字符，反而可以在实验机的环境下在文件中有效地保存 UTF-8 的字符（因为编辑器按照 UTF-8 编码保存了文件）。程序员应该注意到编辑器使用的编码对 Unicode 的影响。而在之后面的 sizeof 运算符中，我们看到了不同编码下原生字符串字面量的大小，跟其声明的类型是完全一致的。

此外，原生字符串字面量也像 C 的字符串字面量一样遵从连接规则。我们可以看看代码清单 8-22 所示的例子。

**代码清单 8-22**

```
#include <iostream>
using namespace std;

int main() {
    char u8string[] = u8R"(你好)" " = hello";
    cout << u8string << endl;    // 输出"你好 = hello"
    cout << sizeof(u8string) << endl;    // 15
    return 0;
}

// 编译选项:g++ -std=c++11 8-4-3.cpp
```

可以看到，代码清单 8-22 中的原生字符串字面量和普通的字符串字面量会被编译器自动连接起来。整个字符串有 2 个 3 字节的中文字符，以及 8 个 ASCII 字符，加上自动生成的 \0，字符串的总长度为 15 字节。与非原生字符串字面量一样，连接不同前缀的（编码）的字符串有可能导致不可知的结果，所以程序员总是应该避免这样使用字符串。

## 8.5 本章小结

本章中我们了解了 C++11 支持的 4 种新特性：对齐方式、通用属性、Unicode，以及原生字符串字面量。

对齐方式本是语言设计者想掩藏的细节，不过在 C++11 编程方式越发复杂的情况下，提供给用户更底层的手段往往是必不可少的。在一些情况下，用户虽然不能保证总是写出平台无关，或者说各平台性能最优的代码，但只需改造 alignas 之后的对齐值参数就可以保证程序的移植性及性能良好，也不失为一种好的选择。而 C++11 对对齐方式的支持从语法规则到库，基本上考虑到了各种情况，可以说是相当完备的。

而通用属性则像是关键字的包装器。一度有人认为，C++ 应该是用通用属性而不是关键字来实现一些特征，不过最后的结论却是：语言本身的所有特性都应该是关键字，通用属性仅仅用在不改变语义的场合，比如产生编译警告、优化提示等。从现在的情况看来，通用属性的语法规则意义大于现在已有的两个预定义通用属性。编译器厂商或组织或者标准委员会在对语言进行扩展的时候，可能还会利用这样的通用属性的语法规则。

C++11 还增强了对 Unicode 的支持。针对以前长度并不明确的 wchar_t，增加了 char16_t 及 char32_t 两种内置类型。考虑到变长编码 UTF-8 使用上的不方便，以及定长的 UTF-16 和 UTF-32 在存储或者一些其他方面的弱势，C++11 在逐步加强对 Unicode 类型转换方面的支持。不过基于 Unicode 的编程是否容易了很多，可移植性是否加强了很多，可能还需要各位读者慢慢体会。此外，C++11 还支持了原生字符串字面量。这是一个在其他较晚发明的语言中常见的特性，C++11 将其引入其中，也算方便了程序员对 C++ 字符串的学习和使用。