Other data types

Type aliases (typedef / using)

A type alias is a different name by which a type can be identified. In C++, any valid type can be aliased so that it can be referred to with a different identifier.

类型别名是用来标识类型的另一个名称。在c++中，任何有效类型都可以使用别名，这样就可以用不同的标识符引用它。

In C++, there are two syntaxes for creating such type aliases: The first, inherited from the C language, uses the `typedef` keyword:

在c++中，有两种创建此类类型别名的语法:第一种语法继承自C语言，使用typedef关键字:

`typedef existing_type new_type_name ;`

where `existing_type` is any type, either fundamental or compound, and `new_type_name` is an identifier with the new name given to the type.

其中existing_type是任何类型，基本类型或复合类型，new_type_name是一个标识符，该标识符为该类型指定了新名称。

For example:

```
typedef char C;
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```

This defines four type aliases: `C`, `WORD`, `pChar`, and `field` as `char`, `unsigned int`, `char*` and `char[50]`, respectively. Once these aliases are defined, they can be used in any declaration just like any other valid type:

这定义了四种类型别名:C、WORD、pChar和field分别为char、unsigned int、char*和char[50]。一旦定义了这些别名，它们就可以像其他有效类型一样在任何声明中使用:

```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
```

More recently, a second syntax to define type aliases was introduced in the C++ language:

最近，c++语言中引入了定义类型别名的第二种语法:

补充：只有c++11 版本支持支持

```
using new_type_name = existing_type ;
```

For example, the same type aliases as above could be defined as:

例如，与上面相同的类型别名可以定义为:

```
using C = char;
using WORD = unsigned int;
using pChar = char *;
using field = char [50];
```

Both aliases defined with `typedef` and aliases defined with `using` are semantically equivalent. The only difference being that `typedef` has certain limitations in the realm of templates that `using` has not. Therefore, `using` is more generic, although `typedef` has a longer history and is probably more common in existing code.

用typedef定义的别名和用using定义的别名在语义上是等价的。唯一的区别是typedef在使用的模板方面有一定的限制。因此，使用更通用，尽管typedef有更长的历史，并且可能在现有代码中更常见。

Note that neither `typedef` nor `using` create new distinct data types. They only create synonyms of existing types. That means that the type of `myword` above, declared with type `WORD`, can as well be considered of type `unsigned int`; it does not really matter, since both are actually referring to the same type.

注意，typedef和using都不会创建新的不同的数据类型。它们只创建现有类型的同义词。这意味着上面用WORD类型声明的myword类型也可以被认为是unsigned int类型;这并不重要，因为两者实际上引用的是相同的类型。

Type aliases can be used to reduce the length of long or confusing type names, but they are most useful as tools to abstract programs from the underlying types they use. For example, by using an alias of `int` to refer to a particular kind of parameter instead of using `int` directly, it allows for the type to be easily replaced by `long` (or some other type) in a later version, without having to change every instance where it is used.

类型别名可以用来减少长类型名或容易混淆的类型名的长度，但它们最有用的用途是将程序从所使用的底层类型抽象出来。例如，通过使用int的别名来引用特定类型的形参，而不是直接使用int，它允许该类型在以后的版本中很容易被long(或其他类型)替换，而不必更改使用它的每个实例。

Unions

Unions allow one portion of memory to be accessed as different data types. Its declaration and use is similar to the one of structures, but its functionality is totally different:

联合允许以不同的数据类型访问内存的一部分。它的声明和使用类似于结构的声明，但它的功能完全不同:

```
union type_name {
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
  .
  .
} object_names;
```

This creates a new union type, identified by `type_name`, in which all its member elements occupy the same physical space in memory. The size of this type is the one of the largest member element. For example:

这将创建一个新的联合类型，由' type_name '标识，其中所有成员元素在内存中占据相同的物理空间。这种类型的大小是最大的成员元素之一。例如:

```
union mytypes_t {
  char c;
  int i;
  float f;
} mytypes;
```

declares an object (`mytypes`) with three members:

声明一个具有三个成员的对象(' mytypes '):

```
mytypes.c
mytypes.i
mytypes.f
```

Each of these members is of a different data type. But since all of them are referring to the same location in memory, the modification of one of the members will affect the value of all of them. It is not possible to store different values in them in a way that each is independent of the others.

每个成员都具有不同的数据类型。但由于它们都指向内存中的相同位置，因此修改其中一个成员将影响所有成员的值。不可能以一种相互独立的方式在其中存储不同的值。

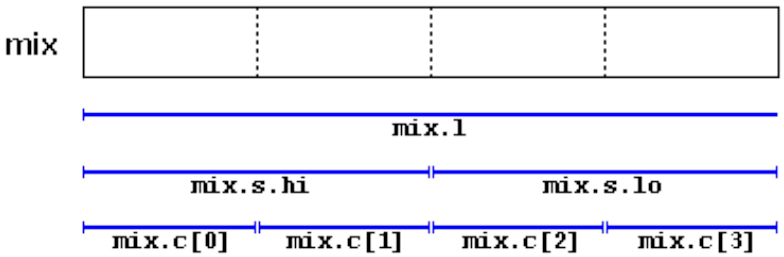
One of the uses of a union is to be able to access a value either in its entirety or as an array or structure of smaller elements. For example:

联合的用途之一是能够访问整个值，或者作为数组或更小元素的结构访问值。例如:

```
union mix_t {
  int l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```

If we assume that the system where this program runs has an `int` type with a size of 4 bytes, and a `short` type of 2 bytes, the union defined above allows the access to the same group of 4 bytes: `mix.l`, `mix.s` and `mix.c`, and which we can use according to how we want to access these bytes: as if they were a single value of type `int`, or as if they were two values of type `short`, or as an array of `char` elements, respectively. The example mixes types, arrays, and structures in the union to demonstrate different ways to access the data. For a little-endian system, this union could be represented as:

如果我们假设这个程序运行的系统有一个大小为4字节的int类型和一个大小为2字节的short类型，那么上面定义的union允许访问相同的4字节组:mix.l,mix.S和mix.c，我们可以根据想要访问这些字节的方式使用它们:就好像它们是int类型的单个值，或者就好像它们是short类型的两个值，或者分别是char元素的数组。该示例在联合中混合了类型、数组和结构，以演示访问数据的不同方法。对于小端系统，这个联合可以表示为:



The exact alignment and order of the members of a union in memory depends on the system, with the possibility of creating portability issues.

联盟成员在内存中的准确对齐和顺序取决于系统，可能会产生可移植性问题。

Anonymous unions

When unions are members of a class (or structure), they can be declared with no name. In this case, they become *anonymous unions*, and its members are directly accessible from objects by their member names. For example, see the differences between these two structure declarations:

当联合是类(或结构)的成员时，可以不使用名称来声明它们。在这种情况下，它们变成了“匿名联合”，它的成员可以通过成员名直接从对象中访问。例如，看看这两个结构声明之间的区别:

常规用法

```
struct book1_t {
  char title[50];
  char author[50];
  union {
    float dollars;
    int yen;
  } price;
} book1;
```

匿名用法

```
struct book2_t {
  char title[50];
  char author[50];
  union {
    float dollars;
    int yen;
  };
} book2;
```

The only difference between the two types is that in the first one, the member union has a name (`price`), while in the second it has not. This affects the way to access members `dollars` and `yen` of an object of this type. For an object of the first type (with a regular union), it would be:

这两种类型之间的唯一区别是，在第一个类型中，成员联合有一个名称(price)，而在第二个类型中没有。这将影响访问该类型对象的成员美元和日元的方式。对于第一种类型的对象(具有常规联合)，它将是:

```
book1.price.dollars
book1.price.yen
```

whereas for an object of the second type (which has an anonymous union), it would be:

而对于第二种类型的对象(具有匿名联合)，则为:

```
book2.dollars
book2.yen
```

Again, remember that because it is a member union (not a member structure), the members `dollars` and `yen` actually share the same memory location, so they cannot be used to store two different values simultaneously. The `price` can be set in `dollars` or in `yen`, but not in both simultaneously.

同样，请记住，因为它是一个成员联盟(而不是成员结构)，成员dollar和yen实际上共享相同的内存位置，因此不能同时使用它们存储两个不同的值。价格可以用美元或日元确定，但不能同时用两种货币。

Enumerated types (enum)

Enumerated types are types that are defined with a set of custom identifiers, known as *enumerators*, as possible values. Objects of these *enumerated types* can take any of these enumerators as value.

枚举类型是用一组自定义标识符定义的类型，称为*枚举器*，作为可能的值。这些*枚举类型*的对象可以将这些枚举数中的任何一个作为值。

Their syntax is:

```
enum type_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

This creates the type `type_name`, which can take any of `value1`, `value2`, `value3`, ... as value. Objects (variables) of this type can directly be instantiated as `object_names`.

这将创建类型type_name，它可以接受value1, value2, value3，...作为价值。这种类型的对象(变量)可以直接实例化为object_names。

For example, a new type of variable called `colors_t` could be defined to store colors with the following declaration:

例如，可以定义一个名为colors_t的新类型变量来存储颜色，声明如下:

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

Notice that this declaration includes no other type, neither fundamental nor compound, in its definition. To say it another way, somehow, this creates a whole new data type from scratch without basing it on any other existing type. The possible values that variables of this new type `color_t` may take are the enumerators listed within braces. For example, once the `colors_t` enumerated type is declared, the following expressions will be valid:

注意，该声明在其定义中不包含其他类型，无论是基类还是复合类。换句话说，这从某种程度上创建了一个全新的数据类型，而不基于任何其他现有类型。这种新的类型color_t的变量可能接受的值是大括号中列出的枚举数。例如，一旦声明了colors_t枚举类型，下面的表达式将是有效的:

```
colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

Values of *enumerated types* declared with `enum` are implicitly convertible to an integer type. In fact, the elements of such an `enum` are always assigned an integer numerical equivalent internally, to which they can be implicitly converted to. If it is not specified otherwise, the integer value equivalent to the first possible value is `0`, the equivalent to the second is `1`, to the third is `2`, and so on... Therefore, in the data type `colors_t` defined above, `black` would be equivalent to `0`, `blue` would be equivalent to `1`, `green` to `2`, and so on...

用enum声明的枚举类型的值可以隐式转换为整数类型。事实上，这样的枚举中的元素总是在内部分配一个等价的整数数值，它们可以隐式地转换为这个数值。如果没有指定，则与第一个可能值相等的整数值为0，与第二个可能值相等的整数值为1，第三个可能值为2，以此类推.....因此，在上面定义的数据类型colors_t中，黑色等于0，蓝色等于1，绿色等于2，以此类推.....

A specific integer value can be specified for any of the possible values in the enumerated type. And if the constant value that follows it is itself not given its own value, it is automatically assumed to be the same value plus one. For example:

可以为枚举类型中的任何可能值指定特定整数值。如果后面的常量本身没有给出它自己的值，它会自动被假定为相同的值加1。例如:

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december} y2k;
```

In this case, the variable `y2k` of the enumerated type `months_t` can contain any of the 12 possible values that go from `january` to `december` and that are equivalent to the values between `1` and `12` (not between `0` and `11`, since `january` has been made equal to `1`).

在这种情况下，枚举类型months_t的变量y2k可以包含从1月到12月的12个可能值中的任何一个，这些值等于1到12之间的值(不是0到11之间的值，因为1月已经等于1)。

Enumerated types with enum class

补充：实际测试要在c++11环境中使用

But, in C++, it is possible to create real `enum` types that are neither implicitly convertible to `int` and that neither have enumerator values of type `int`, but of the `enum` type itself, thus preserving type safety. They are declared with `enum class` (or `enum struct`) instead of just `enum`:

但是，在c++中，可以创建既不能隐式转换为int，也没有int类型枚举器值，而是枚举类型本身的实际枚举类型，从而保持类型安全。它们是用enum类(或enum struct)声明的，而不是只用enum:

```
enum class Colors {black, blue, green, cyan, red, purple, yellow, white};
```

Each of the enumerator values of an `enum class` type needs to be scoped into its type (this is actually also possible with `enum` types, but it is only optional). For example:

枚举类类型的每个枚举器值都需要作用域为其类型(对于枚举类型，这实际上也是可能的，但只是可选的)。例如:

```
Colors mycolor;

mycolor = Colors::blue;
if (mycolor == Colors::green) mycolor = Colors::red;
```

Enumerated types declared with `enum class` also have more control over their underlying type; it may be any integral data type, such as `char`, `short` or `unsigned int`, which essentially serves to determine the size of the type. This is specified by a colon and the underlying type following the enumerated type. For example:

用枚举类声明的枚举类型还可以更多地控制其底层类型;它可以是任何整型数据类型，如char、short或unsigned int，这基本上用于确定类型的大小。这是由冒号和枚举类型后面的基础类型指定的。例如:

```
enum class EyeColor : char {blue, green, brown};
```

Here, `Eyecolor` is a distinct type with the same size of a `char` (1 byte).

在这里，Eyecolor是一个具有相同char(1字节)大小的独特类型。