`C++11` öncesinde bir türe eş isim `(type alias)` oluşturmanın tek yolu `C`'den gelen `typedef` bildirimleriydi:

```
#include <vector>
#include <string>

typedef int Word;
typedef int *Iptr;
typedef int SMatrix10[10][10];
typedef int(*Fptr)(int, int);
typedef const std::vector<std::string> Csvec;
```

`C++11` standartları ile türlere eş isim oluşturmak için ikinci bir araç daha geldi. Artık eş isim bildirimlerini `using` anahtar sözcüğü ile yapabiliyoruz. Sentaks çok basit: `using` anahtar sözcüğünü seçilen eş isim izliyor ve `=` atomundan sonra ise eş ismin hangi türe karşılık geldiği yazılıyor. Yukarıdaki `typedef` bildirimleri yerine using bildirimleri yazalım:

```
using Word = int;
using Iptr = int *;
using SMatrix10 = int [10][10];
using Fptr = int(*)(int, int);
using Csvec = const std::vector<std::string>;
```

Ancak eş isim bildirimleri konusunda yeni bir aracın daha dile eklenmesinin ana nedeni, daha önce `typedef` bildirimleriyle mümkün olmayan eş isim şablonlarının `(alias templates)` oluşturabilmesini mümkün kılmak:

```
template <class T>
struct Alloc {
 //...
};

#include <vector>

template<class T>
using Vec = std::vector<T, Alloc<T>>;

Vec<int> v;
```

Yukarıdaki kodda `Allocator` olarak kullanılacak `Alloc` isimli bir sınıf tanımlanıyor. Daha sonra `Vec` isimli bir eş isim şablonu oluşturuluyor. Böylece kod içinde `vector` sınıf şablonunda ikinci şablon tür parametresi olarak `Alloc` sınıf şablonunun kullanılması durumunda, şablon tür argümanı olarak `Alloc` sınıfını belirtmeye gerek kalmayacak. Örneğin

```
std::vector<int, Alloc<int>>
```

yazmak yerine, yalnızca

```
Vec<int>
```

yazılabilecek. Birkaç örnek daha verelim:

```
#include <map>
#include <string>

template<typename T>
using Smap = std::map<std::string, T>;

Smap<int> simap;
```

Yukarıdaki kodda oluşturulan `Smap` eş isim şablonunun tür parametresi, standart `map` sınıfının ikinci şablon tür parametresini belirleyecek. Birinci şablon tür parametresi standart `string` sınıfı olacak. Bu durumda

```
Smap<int>
```

yazmak ile

```
std::map<std::string, int>
```

yazmak aynı anlama gelecek. Şimdi de aşağıdaki koda bakalım:

```
template<typename T>
using Ptr = T *;

double dval = 2.3;
Ptr<double> p = &dval;
```

Yukarıdaki kodda `p` `double` türden bir nesneyi gösteren bir `pointer` değişken.

Eş isim şablonları da varsayılan tür argümanı alabilir:

```
#include <set>
#include <functional>

template<typename T, typename C = std::greater<T>, typename A = std::allocator<T>>
using Gset = std::set<T, C, A>;

int main()
{
 Gset<int> myset;
 //...
}
```

Yukarıdaki kodda tanımlanan `myset` değişkeni

```
std::set<double, std::greater<int>, std::allocator<int>>
```

türünden.

Şablon sabit parametreleri de `(non type parameters)` eş isim şablonlarında kullanılabilir:

```
template<typename T, size_t low, size_t high>
class Rand {
 //...
};

template<size_t high>
using IRand = Rand<int, 0, high>;

int main()
{
 IRand<100> x;
        //...
}
```

Yukarıdaki kodda, `Irand` şablon ismi `100` argüman değeri ile kullanıldığında bu şablon açılımı

```
Rand<int, 0, 100>
```

açılımı anlamına geliyor.

Sınıf şablonlarında ya da işlev şablonlarında yapılabilen açık özelleştirme `(explicit specialization)` ya da yalnızca sınıf şablonlarında mümkün olan kısmi özelleştirme `(partial specialization)` araçları eş isim şablonlarında kullanılamıyor. Eş isim şablonlarında tür çıkarımı da söz konusu değil. Eş isim şablonları isim alanı kapsamında `(namespace scope)` ya da sınıf kapsamında `(class scope)` bildirilebiliyor. Ancak sınıf şablonlarında ve işlev şablonlarında olduğu gibi isim alanı şablonlarının da yerel bir blok içinde bildirilmesi geçerli değil.