

C++0X: The New Face of Standard C++

informIT

From the book [_INFORMIT C++ Reference Guide](#) by [_Danny Kalev](#)

The current C++ standard, officially known as ISO/IEC 14882, is already seven years old. The C++ standards committee is currently working on extending and enhancing C++. Most of the work focuses on additions to the Standard Library, with very minimal core language changes. In this series, I will explore some of the novel features added to the Standard Library.

Rationale

Extensions and enhancements are the best indicator that a programming language is being used heavily in the real world rather than being confined to esoteric academic niches. Therefore, it's no surprise that the current standard is undergoing an overhaul. There's no need to panic, though: it will still remain the same C++ as you've known for years, except that it will include many new facilities that are long overdue. As always, the standardization process is long-winded but in the long run, it will pay off. When you consider the frequent wholesale upheavals that new programming languages undergo with every new release, I'm sure you'll agree that it's better to make a good standard right from the start (even if it takes longer) than to rush the standardization process only to discover later that "My goodness, we forgot generics!"

As I showed before, adding a half-baked feature at the last moment is never a good idea. The committee is well aware of this. Therefore, proposals for new features are always based on existing implementations. For instance, the regular expressions library of C++ (to which I will dedicate a separate section) is based on the `boost::regex` library (see http://sourceforge.net/project/showfiles.php?group_id=7586). This way, the feedback of users can help the designers fine-tune their implementation, and add new features to it. Only then is the refined implementation proposed to the committee. The official paper that specifies the newly added features is called *The Standard Library Extensions Technical Report 1* (known as the "Library TR1"). The committee in turn reviews the TR, suggesting improvements, proper Standardese and implementation guidelines. There is no obligation to accept the facilities described in the TR as is; in fact, it is more than likely that the final version will look different. Namespace issues, naming conventions, exception-safety, interaction with existing Standard Library components and other factors will affect the final version of the proposed features. Therefore, all the code listings, class and function names, and other design and implementation details provided in this section are subject to change.



Buy This Book From informIT

The <tuple> Library

The Library TR includes the new header file <tuple>, which contains a tuple class template and its helper functions. A tuple is a fixed-size heterogeneous collection of objects. Several programming languages, e.g., Python and ML, have tuple types. Tuple types have many useful applications, such as packing multiple return values for a single function, simultaneous assignment and comparison of multiple objects, and grouping related objects (such as a function's arguments). A tuple's size is the number of elements stored in it. The current specification supports tuples with 0-10 elements. Each element can be of a different type. The following example creates a tuple type that has two elements:

```
double and void *:
#include <tuple>
    tuple <double, void *> t(2.5, &x);
```

If you omit the initializers, default initialization will take place instead:

```
tuple <double, std::string> t; //default initialized to (0.0, string())
```

Helper Functions

The `make_tuple()` function simplifies the construction of tuple types. It detects the types of its arguments and instantiates a tuple type accordingly:

```
void func(int i);
make_tuple(func); // returns: tuple<void (*)(int)>
    make_tuple("test", 9); // tuple< const char (&)[5], int>
```

To examine a tuple's size, use the `tuple_size()` function:

```
int n=tuple_size < tuple <int, std::string> >::value;//2
```

To obtain the type of an individual element, use the `tuple_element()` function.

`tuple_element()` returns the type of an individual element. This function takes an index and the tuple type. The following example uses this function to retrieve the type of the first element in a tuple:

```
typedef tuple_element <0, tuple<float, int, char> >::type
    T1;// T1 is a synonym for float
```

To access the element itself, either for reading its value or to assign a new value to it, use the `get<>()` function template. The template argument (the argument passed in the angle brackets) is the index of sought-after element, and the argument in parentheses contains the tuple type:

```
tuple <int, double> tpl;
int n=get<0>(tpl); //read 1st element
get<1>(t)=9.5; //assign to the 2nd element
```



Buy This Book From informIT

Applications

Functions with a dual interface can use tuples to pack two or more return types. Some of the better candidates for dual interfacing include Standard Library functions that return only `char *` but not `std::string`. For example, the `getenv()` function retrieves an environment variable and returns its value in the form of a C-string. In this case, overloading can do the trick:

```
char * getenv(const char * name); //existing version
//hypothetic overloaded version
    const std::string& getenv(const std::string name);
```

However, if the function can't use different arguments to play this trick, tuples can save the day. For example, a function that translates a file name to its equivalent `FILE *` and a *file descriptor* can't rely on overloading because its parameter doesn't change, only the return type. POSIX often solves this problem by defining multiple functions with slightly different names:

```
int traslatefile(const char * path);
    FILE * ftraslatefile(const char * path);
```

Using a tuple can solve this problem neatly:

```
typedef tuple <int, FILE *> file_type;
file_type translatefile (const char *);
```

Summary

Tuple types are only one of many new facilities that are being added to the Standard Library. The fact that very minimal core changes, if any, are needed to support the new proposals shows the foresight of C++ designers. Consequently, the new facilities can be neatly integrated in existing code without causing conflicts or design changes.

Reference Wrapper Class

Continuing our journey into the new C++ standard, this time I will present another proposal that has been incorporated into the Library Extensions Technical Report: the reference wrapper class.

Rationale

In certain contexts, built-in reference types cannot be used as first-class citizens in C++. For example, you can't define a container of references:

```
std::vector <int &> vri; //won't compile
```

In this respect, reference types are different from bare pointers, which can be used as container elements. Additionally, it is sometimes necessary to pass references as arguments to algorithms and functions that would usually create a copy of their arguments.



Buy This Book From informIT

A reference wrapper class template enables you to wrap a reference in the guise of an object. The object can then be used in contexts in which built-in references won't work. Consider the following example:

```
void func(int & r)
{
    r++;
}

template<class F, class T> void g(F f, T t)
{
    f(t);
}

int main()
{
    int i = 0;
    g(func, i);
}
```

The second parameter of `g()` is passed by value. If you want to force `g()` to take a reference instead, a reference wrapper can do the trick.

The reference wrapper Class Template

`reference_wrapper<T>` is a *copy-constructible* and *assignable* wrapper around an object of type `T&`. The copy-constructible and assignable properties ensure, among other things, that `reference_wrapper` objects can be used as an element of STL containers.

The `reference_wrapper` class and its helper functions are declared in the `<utility>` header. This header, as you probably know, already contains several other facilities such as `auto_ptr`. (For more on `auto_ptr`, see Chapter 4, "Memory Management.")

There are two helper functions: `ref()` and `cref()`, which create a `reference_wrapper` object that wraps their argument. `cref()` returns a `const T&` wrapper object, whereas `ref()` returns a `T&` wrapper object.

When dealing with plain references and references to `const` objects, people often use the term "const reference" when they actually mean a reference to a `const` object. Notice that there's no point in having a `const` reference in C++ in the first place because a reference, once initialized, can't be bound to a different object.

Notice how the use of a `reference_wrapper` helps us ensure that the function template `g()` behaves as if it took a reference rather than a copy of its argument:

```
int main()
{
    int i = 0;
    g(f, i); //pass i by value
    cout << i << endl; //as expected, 0
    g(f, ref(i)); //bind a reference to i and pass it as arg
    cout << i << endl; // output: 1
}
```



Buy This Book From informIT

Let's look at what this program does. First, the function `g()` is called with two arguments that are passed by value: a function pointer and a copy of `i`. `g()` in turn invokes the function bound to `f`, which increments the copy of `i`. As expected, when `g()` returns, the change to the local copy of `i` isn't reflected in the `i` that was declared in `main()`. Therefore, the first `cout` expression displays 0. In the second call to `g()`, the `ref()` helper function creates a temporary `reference_wrapper()` that is bound to `i`. The side effects of `func()` are therefore reflected in `i` after the call and the second `cout` expression displays 1.

`reference_wrapper` can be used where ordinary references cannot, such as in containers:

```
std::list<int> num;
std::vector<reference_wrapper<int> >
    num_refs; // a list of references to int

for(int i = 0; i < 10; ++i)
{
    numbers.push_back(2*i*i^4 - 8*i + 7); //ordinary copy semantics
    num_refs.push_back(
//create a reference to the last element in nums
    ref(numbers.back()));
}
```

A `reference_wrapper` enables you to pass `T&` as an argument to algorithms that expect the underlying type, i.e., `T`:

```
std::sort(num_refs.begin(), num_refs.end());
```

Using `reference_wrapper` with Tuples

`reference_wrapper` also enables you to create tuples of references and references to `const` in cases where the tuple class would use the underlying, non-cv-qualified type instead:

```
void f(const A& ca, B& b)
{
    make_tuple(ca, b); // returns tuple<A, B>
}
```

To override the default behavior of `make_tuple`, use `ref()` and `cref()` like this:

```
A a; B b; const A ca=a;
make_tuple( cref(a), b); // tuple <const A&, B> (a,b)
make_tuple( ref(a), b); // tuple <A&, B> (a,b)
make_tuple( ref(a), cref(b) ); // tuple <A&, const B&> (a,b)
```

Summary

The `reference_wrapper` utility class and its helper functions exemplify how a small library can be neatly incorporated into C++ to fill a few syntactic lacunae. This type of solution is preferable because it guarantees that existing compilers can cope with new C++ code without requiring upgrades and—more importantly—without causing existing code to break.



Buy This Book From informIT