Templates

A class that describes a stack of integers is easy to write. If you need a class of doubles you must write another, very similar, class. With a *class template* you can write a stack where the elements are of an arbitrary type — you supply the actual type as an argument when you *instantiate* the template.

The library containers are class templates: you can create a `vector<int>`, `vector<double>`, `vector<Point>`, ...

There are also *function templates*. You can write a function that takes arguments of arbitrary types.

Note: Lippman mixes the presentation of class templates and function templates. Here, we first present function templates and then class templates.

Defining a Function Template

This is a function template that compares two values of the same type:

```
template <typename T>
int compare(const T& v1, const T& v2) {
    if (v1 < v2) { return -1; }
    if (v2 < v1) { return 1; }
    return 0;
}
```

- `T` is a template parameter, which must be a type.
- You may write `class` instead of `typename` — use `typename` in new programs.

Using a Function Template

When you call a template function, the compiler deduces what types to use instead of the template parameters and instantiates ("writes") a function with the correct types.

```
void f() {
    cout << compare(1, 0) << endl;
    // T is int, the compiler instantiates
    //     int compare(const int&, const int&)

    string s1 = "hello";
    string s2 = "world";
    cout << compare(s1, s2) << endl;
    // T is string, the compiler instantiates
    //     int compare(const string&, const string&)
}
```

Requirements on Template Parameter Types

A template usually puts some requirements on the argument types. If these requirements are not met the instantiation will fail. Example:

```
void f() {
    Point p1(10, 20);
    Point p2(10, 30);
    cout << compare(p1, p2) << endl;
    // T is Point, the compiler tries to instantiate
    //     int compare(const Point&, const Point&),
    // this doesn't compile since Point objects
    // cannot be compared with <
}
```

- If class `Point` implements `operator<`, everything is ok.
- You should try to keep the number of requirements on argument types as small as possible (see next slide).

Writing Type-Independent Code

The only requirement placed on the type `T` in the `compare` template is that objects of `T` can be compared with <. The following implementation puts more requirements on `T` and isn't good:

```
template <typename T>
int compare(T v1, T v2) {
    if (v1 < v2) { return -1; }
    if (v1 == v2) { return 0; }
    return 1;
}
```

- The arguments are passed by value, so it must be possible to copy objects of `T`.
- Objects of `T` must implement comparison with < *and* ==.

Concepts and Models

A *concept* is a set of requirements that a template argument must meet so that the template can compile and execute properly.

Requirements are specified as generally as possible: instead of saying that "class `T` must define the member function `operator++`", you say "for any object `t` of type `T`, the expression `++t` is defined". (It is left unspecified whether the operator is a member or a global function and whether `T` is a built-in type or a user-defined type.)

An entity that meets the requirements of a concept is a *model* of the concept. For example, the class `string` is a model of the concept `LessThanComparable`, which requires that objects can be compared with <.

Basic Concepts

Some important concepts:

DefaultConstructible Objects of type `X` can be constructed without initializing them to any specific value.

Assignable Objects of type `X` can be copied and assigned.

LessThanComparable Objects of type `X` are totally ordered (`x < y` is defined).

EqualityComparable Objects of type `X` can be compared for equality (`x == y` and `x != y` are defined).

All built-in types are models of these concepts. Note that concepts are just words that have no meaning in a program. A proposal to incorporate concepts in the language was rejected at a late stage in the C++11 standardization process, but work continues on a modified proposal.

Type Parameters Must Match

During template instantiation the argument types must match exactly — no conversions are performed. The following call to `compare` will fail:

```
int i = ...;
double d = ...;
cout << compare(i, d) << endl;
```

With this version of the template, the instantiation will succeed:

```
template <typename T, typename U>
int compare2(const T& v1, const U& v2) {
    if (v1 < v2) { return -1; }
    if (v2 < v1) { return 1; }
    return 0;
}
```

A < operator must exist that can compare a `T` object and a `U` object.

Explicit Instantiation

If a function template instantiation fails because the types of the arguments don't match, you can make an explicit instantiation of the template:

```
int i = ...;
double d = ...;
cout << compare<double>(i, d) << endl;
```

Here, `i` is widened to `double` in accordance with the usual conversion rules.

Return Types

A function to compute the smallest of two values of different types:

```
template <typename T, typename U>
T min(const T& v1, const U& v2) {
    return (v1 < v2) ? v1 : v2;
}
```

This is not correct: the return value always has the first argument's type, so `min(4, 2.3)` becomes 2. The return type should be "the most general of the types `T` and `U`", i.e. the type to which the compiler will convert an expression containing `T` and `U` objects.

This can be expressed in the following way `C++11`:

```
template <typename T, typename U>
auto min(const T& v1, const U& v2) -> decltype(v1 + v2) {
    return (v1 < v2) ? v1 : v2;
}
```

Template Compilation

When an "ordinary" function is called, the compiler needs to see only the function declaration. For an inline function, code will be generated at the call site, so the function *definition* must be available at the call.

This means that inline function definitions should be placed in header files. The same holds for template function definitions.

The most common errors when templates are used are related to types: trying to instantiate a template with arguments of types that don't support all requirements. Such errors are reported during instantiation of the template.