templates

an introduction

definition from Wikipedia

“Templates are a feature of the C++ programming language that allows functions and classes to operate with **generic types**.

This allows a function or class to work on many different data types without being rewritten for each one.”

kinds of templates

1. function templates
2. class templates
3. variable templates (C++14)

- variadic: variable number of arguments
- non-variadic: fixed number of arguments

function templates
(from C++ Primer)

function template motivation

comparing two values

```
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
int compare(const string &v1, const string &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int compare(const double &v1, const double &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

is this bad? why?

function template example

comparing two values

```
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

comma-separated template parameter list acts a lot like function parameter list

type(s) not specified until the template is used

function template example

when template function is used, a specific version of the function is **instantiated** for the particular set of template parameters

```
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}

int x=1, y=0;
cout << compare <int> (x, y) << endl; // T is int

vector vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare <vector<int>> (vec1, vec2) << endl; // T is vector<int>
```

function template example

the appropriate instantiation can be determined automatically based on the types of the function arguments, without having to specify them explicitly

```
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}

int x=1, y=0;
cout << compare(x, y) << endl; // T is int

vector vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare(vec1, vec2) << endl; // T is vector<int>
```

template type parameters

a type parameter can generally be used like a built-in or class type specifier

```
// ok: same type used for the return type and parameter
template <typename T>
T foo(T* p)
{
    T tmp = *p; // tmp will have the type to which p points
    // …
    return tmp;
}
```

here `T` is used both to name the function return type and declare a variable inside the function body

template type parameters

each **type** parameter must be preceded by `class` or `typename` keyword

```
template <typename T, U> calc(const T&, const U&); // error

template <typename T, class U> calc(const T&, const U&); // ok
```

there is no distinction between `class` and `typename` in a template parameter list, but you may see both used

nontype template parameters

templates can take nontype parameters, which represent values

the specific typename is given instead of `class` or `typename` keyword

```
template <unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M])
{
    return strcmp(p1, p2);
}
```

here `compare` has been written to handle two string literals (`const char` arrays) of different lengths, with the lengths specified by template parameters `N` and `M`

```
cout << compare(“hi”, “mom”) << endl; // what are N and M in this instantiation?
```

writing type-independent code

```
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

important principles for writing **generic** code, illustrated by `compare`

- function parameters in the template are `const` references
- tests in the body use only `<` comparisons

why are these principles important for type independence?

template compilation and code organization

the compiler generates code for a template only when we instantiate a specific instance of the template

usually, the compiler only needs to see function declaration (or class definition, but not member definitions)

because the compiler must have all the code that defines a function template, header files for templates typically include definitions *and* declarations

class templates

class template example (from cplusplus.com)

```
template <class T>
class mypair {
    T a, b;
public:
    mypair (T first, T second)
        {a=first; b=second;}
    T getmax();
};

template <class T>       // how is T being used here?
T mypair<T>::getmax ()  // how is T being used here?
{
    T retval;
    retval = a>b? a : b;
    return retval;
}
```