templates in C++

an overview
(mostly following *C++ Primer*)

polymorphism (“many forms”)

an interface displays polymorphism if it can operate with values of **different types**

```
Type1 x;
Type2 y;

f(x);
f(y);
```

https://stackoverflow.com/a/5854862

there are different ways to achieve this

polymorphism (“many forms”)

polymorphic mechanisms are divided into two main categories based on when the program selects the type-specific code to run:

- compile-time polymorphism
- run-time polymorphism

compile-time polymorphisms

ad-hoc polymorphism (overloading)

parametric polymorphism (templates, macros)

coercion polymorphism (casting)

https://catonmat.net/cpp-polymorphism

overloading in C++

comparing two values

```
// returns 0 if the values are equal,
// -1 if v1 is smaller, 1 if v2 is smaller
// (like spaceship operator <=> in C++20)

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;
}
```

overloading in C++

the compiler determines which function to use based on the argument types

three possible outcomes for a call to an overloaded function:

- best match found
- no match found (error)
- ambiguous call (error)

ranking during “overload resolution” is based on conversions of arguments

https://en.cppreference.com/w/cpp/language/overload_resolution

overloading in C++

some basic rules for overloaded functions:

- must differ in number or type(s) of parameters
- parameter names are irrelevant (they are just documentation)
- cannot differ only in return type

beyond overloading: templates

often we want to perform the same operation on **generic types**

e.g. comparisons, printing, data structures

templates act as “blueprints” for type-specific implementations, saving us from having to write specific code for each type

kinds of templates

function templates

class templates

variable templates (C++14)

non-variadic: fixed number of template arguments

variadic: variable number of template arguments (C++11)

function template example

comparing two values (overloading)

```
// returns 0 if the values are equal,
// -1 if v1 is smaller, 1 if v2 is smaller
// (like spaceship operator <=> in C++20)

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;
}
```

function template example

comparing two values (template)

```
// 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

can also include non-type parameters e.g. `template <typename T, int N>`

function template example

when a function template is used, a specific version of the function is **instantiated** by the compiler 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(x, y) << endl; // instantiation where T is int

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

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 (when we use it)

usually, the compiler only needs to see function declaration, not definition (or class definition, but not member definitions), so we can separate headers and source

for template instantiation, the compiler needs to have all the code that defines the function, so headers for templates typically include declarations *and* definitions

template specializations

it’s not always possible to write a single template that is best suited for every possible template argument with which the template might be instantiated:

- general definition might not compile, or do the wrong thing for a type
- might be more efficient to take advantage of some specific knowledge of type

in such cases we can define a **specialized** version of the template, basically taking over from the compiler by supplying a definition of a specific instantiation of the template (distinct from overloading)

class template specialization scenarios

a template specialization does not have to supply an argument for every parameter, allowing for **partial specializations** which are themselves templates

```
template <typename T, typename U> class Foo {};       // original class template with parameters T & U
template <> class Foo<int, std::string> {};           // full specialization
template <typename T> class Foo<T, std::string> {};   // partial specialization where U is a string
template <typename T> class Foo<T, T*> {};            // partial specialization where U is pointer to T
```

can specialize member functions without specializing the whole class template

```
template <typename T> struct Foo {
    Foo();
    void bar() { /* ... */ }  // some member function
    // ... other members of Foo };

template<>              // we're specializing a template
void Foo<int>::bar()    // we're specializing the Bar member of Foo<int>
{ // do whatever specialized processing that applies to ints }
```

template specialization code organization

to specialize a template, a declaration for the original template must be in scope

a declaration for a specialization must be in scope before any code that uses that instantiation of the template

unlike with ordinary classes and functions, missing declarations of specializations are hard to find, because the compiler will usually just generate code using the original template and not complain