Part 5 — Generic Programming

Introduction

Code is often independent of actual types. E.g.

- Sorting routines (C's qsort)
- Containers (vectors, lists, sets, maps,...)

Generic programming:

- Implement once and reuse code for arbitrary types

Benefit: code is easier to maintain!

- C way: use `void*` as generic pointer type and pass function pointers
- C++ way: function templates, class templates, and functors

Example: Sorting Array

C:

```c
                start address   #elements    #bytes per element
                      |             |              |
void qsort(void *base, size_t number, size_t size, int(*compar)(void *, void *));
                                                            |
                        pointer to function that compares two keys via generic pointers


Foo a[N];
...
qsort(a, N, sizeof(a[0]), Foo_compare);
```

C++:

```cpp
std::sort(v.begin(), v.end(), Compare());
```

Shorter and faster because compiler can inline code, such as simple integer comparisons for which CPUs have dedicated machine instructions.

We will soon see how this works.

Function Templates

```cpp
int min(int a, int b) { return a < b ? a : b; }
float min(float a, float b) { return a < b ? a : b; }
```

In C++, there is no need for defining long lists of (almost) identical functions! The following generic definition covers (almost) all:

```cpp
template <typename T>
T min(T a, T b) {
    return a < b ? a : b;
}
```

Function `min` is now parametrized by type T. The compiler generates implementations for actual type instances when the function is used.

Note: function templates usually make some assumptions about type parameters.

For instance, `min` assumes that operator < is defined for values of type T. If this is not the case, you'll see a compiler error message.

Function Templates — Example

```cpp
template <typename T>
T max(T a, T b) {
    return a > b ? a : b;
}

template <typename T>
void swap(T &a, T &b) {
    T temp(a); a = b; b = temp;
}

int main() {
    int a = 10, b = 5, c = max(a,b);          // calls max<int,int>
    float e = 2.0, f = 1.0, g = max(e,f);      // max<float,float>
    swap(a, b);                                // swap<int, int>
    swap(e, f);                                // swap<float, float>
}
```

swap assumes that type T can be copy constructed and assigned.

If this is not the case (because maybe the author of class T was lazy and made T's AO private), the compiler will complain.

Function Templates — Example

```
template <typename T>                  // OK, forward declaration; eventually,
T max(T a, T b);                       //code needs to be provided in header file

template <typename T>                  // OK, forward declaration; eventually,
void swap(T &a, T &b);                 // code needs to be provided in header file

template <typename U, typename V>      // OK, class/typename are synonyms
U foo(U a, V b) {                      // U,V appear in function signature
    return a;
}

template <typename U, V>
U bar(V a);
// ERROR: no typename/class in front of V
// PROBLEM: when calling bar(x) compiler cannot
// infer type U =>  need to write bar<int>(a)
//                  compiler infers U=int
```

Function Template Instantiation

- Function templates specify how individual functions can be constructed given a set of actual types (instantiation)
- This happens as side-effect of either invoking or taking the address of a function template
- Compiler and/or the linker has to remove multiple identical instantiations
- Template instantiation may be slow — dumb compilers repeat compilation

Type Parameter Binding

General Problem:

```
template <typename T>
void f(ParamType param);

f(expr);
```

- Given type of `expr`, what is the type of T and ParamType?

There are three cases:

- `ParamType` is a reference or pointer (but not universal reference)
- `ParamType` is a universal reference (outside the scope of the course)
- `ParamType` is neither reference nor pointer

See this great talk from Scott Meyers on type deduction:
https://www.youtube.com/watch?v=wQxj20X-tIU

Non Universal Reference/Pointer Parameters

- If `expr` is a reference, ignore it.
- Pattern match `expr` type against `ParamType` to determine T.

```
template <typename T>
void f(T& param);


int x = 22;
const int cx = x;
const int &rx = x;
f(x);          // T = int, param = int&
f(cx);         // T = const int, param = const int&
f(rx);         // T = const int, param = const int&
```

T is not a reference!

Non Universal Reference/Pointer Parameters

If we change `ParamType` to `const T&`, then T changes, but params type does not!

```
template <typename T>
void f(const T& param);


int x = 22;
const int cx = x;
const int &rx = x;
f(x);           // T = int, param = const int&
f(cx);          // T = ~~const~~ int, param = const int&
f(rx);          // T = ~~const~~ int, param = const int&
```

T is not a reference!

Non Universal Reference/Pointer Parameters

Behaviour with pointers is essentially the same

```
template <typename T>
void f(T* param);


int x = 22;
const int *pcx = &x;
f(&x);          // T = int, param = int*
f(pcx);         // T = const int, param = const int*
```

T is not a pointer!

Non Universal Reference/Pointer Parameters

If we change ParamType to `const T*`, then T changes, but params type does not!

```
template <typename T>
void f(const T* param);


int x = 22;
const int *pcx = &x;
f(&x);          // T = int, param = const int*
f(pcx);         // T = ~~const~~ int, param = const int*
```

T is not a reference!

auto and Non-UnRef/Pointer Variables

auto playes the role of T:

```
int x = 22;
const int cx = x;
const int &rx = x;

auto &v1 = x;           // auto = int, v1 = int&
auto &v2 = cx;          // auto = const int, v2 = const int&
auto &v3 = rx;          // auto = const int, v3 = const int&


const auto &v1 = x; // auto = int, v4 = const int&
const auto &v2 = cx; // auto = ~~const~~ int, v5 = const int&
const auto &v3 = rx; // auto = ~~const~~ int, v6 = const int&
```

The rules for auto (when dealing with non-universal reference/pointer variables) is the same as templates!

Universal References

```
template <typename T>
void f(T&& param);

f(expr);
```

Treated like a normal reference parameter, except:

- If `expr` is an lvalue with deduced type E, T is deduced as E&
- Reference collapsing yields E& for `param`

Universal References

```
template <typename T>
void f(T&& param);

f(expr);


int x = 22;
const int cx = x;
const int &rx = x;


f(x);      // T = int &, param = int &
f(cx);     // T = const int &, param = const int &
f(rx);     // T = const int &, param = const int &
f(22);     // T = int, param = int&&
```

Only place where type deduction will deduce a reference type T!

By-Value Parameters

Deductions rule a bit different from reference/pointer:

- If expr is a reference, ignore that
- If expr is const or volatile, ignore that
- T is the result

```
template <typename T>
void f(T param);
```

```
int x = 22;
const int cx = x;
const int &rx = x;
f(x);         // T = int, param = int
f(cx);        // T = int, param = int
f(rx);        // T = int, param = int
```

Non-Reference Non-Pointer auto

As before, auto plays role of T

```
int x = 22;
const int cx = x;
const int &rx = x;
auto v1 = x;          // auto = int, v1 = int
auto v2 = cx;         // auto = int, v2 = int
auto v3 = rx;         // auto = int, v3 = int
```

auto never deduced to be a reference, it must be manually added!

- If added, use by-reference rules

```
auto v4 = x;          // auto = int, v4 = int
auto& v5 = cx;        // auto = int, v5 = const int&
auto&& v6 = rx;       // v6 = const int& (rx is lvalue)
```

Function Template vs. Macros

- Function templates are type-aware, macros are not
- Function templates can be specialized easily, macros cannot
- Inlined function templates are as fast as macro-generated code
- No ***unwanted*** side-effects with function templates. Macro parameters may be evaluate more than once which can cause trouble:

Example: min

```
#define min(a,b) ((a) < (b) ? (a) : (b))

// vs.

template <typename T>
T min(const T a, const T b) {
    return a < b ? a : b;
}
```

Function Template vs. Macros

Example: array size

```
#define array_size_macro(A) (sizeof(A)/sizeof(A[0]))

// vs.

template <typename T, std::size_t n>
constexpr std::size_t array_size(const T (&A)[n]) {
    // note that we pass a reference to an array
    // otherwise, size information is lost as arrays
    // are then treated as pointers
    return n;
}

...
int A[10];
cout << array_size(A) << endl;  // 10
int B[array_size(A)];           // works

int *i;
cout << array_size_macro(i) << endl;    // shouldn't!
```

Class Templates

Overview:

- Reuse type independent code
- Classes are parameterized by types

```
template <typename T>
class X  {
    ... T can be used here ...
};
```

Class templates are very useful for container types, such as vector, set, list, whose code does not depend on the type of values stored in them.

Class Templates

Class template instantiation is explicit.

```
vector<int> a; // a is a vector of integers
```

whereas instantiation of function templates is usually implicit — the compiler infers types — but can also be explicit:

```
min<int>(a, b)
```

The compiler instantiates class templates on demand. In this process, formal type parameters get replaced by actual parameters.

This also means: class template methods must be implemented in header files, because the compiler needs to know the code to generate when instantiating class templates.

Stack Class Template

```
template <typename T>
class Stack {
public:
    Stack();                                  // Stack is a container class =>
    ~Stack();                                 // method code independent of T

    void push_back(T &v) {...}                // append element
    T &back() {...}                           // last element (if !empty)
    void pop_back() {...}                     // remove last
    bool empty() const {...}                  // true iff empty

private:
    T *p;                                     // implemented using dynamic array
    int size;
};

Stack<int> si;                                // int stack
Stack<float> sf;                              // float stack
si.push_back(5);
std::cout << si.back() << std::endl;          // "5"

if (si.empty()) { exit(0); }
sf.push_back(3.5);
```

Stack Class Template

- When the compiler sees `Stack<int> si;` it compiles class `Stack` for `T=int`. From then on, we can create and manipulate `int`-stacks without subsequent compilations of `Stack<int>`.
- Likewise, when the compiler sees `Stack<float> sf;` later, it compiles class `Stack` for `T=float`, and so on.
- So, our life as programmers just got simpler: similar to function templates, we only have to maintain one generic class definition. The compiler adapts it to various element types T for us on demand.
- Because class templates may be instantiated for multiple types on demand in each compilation unit (.cpp file), the compiler has more work to do and the size of object code it generates can be much bigger compared to regular template-less programs.

Stack Class Template

- It is the job of the linker, which combines all object files to create the final executable file, to remove all but one instance of identical template instantiations.
- E.g., if we use `Stack<int>` in file `Foo.cpp` and `Bar.cpp` the linker must not complain about multiple definitions of stack class methods, but silently drop one set of functions it generated code for twice
- *What impact does this have on compilation in terms of overhead*?

Vector Class Template

```cpp
template <typename T>
class Vector {
public:
    Vector(int size=0);
    ~Vector();

    T &operator[](int i) {
        check(i);
        return p[i];
    }

    // const version (chosen when called on const Vector)
    const T &operator[](int i) const {
        check(i);
        return p[i];
    }

private:
    void check(int i) const {
        assert(0 <= i && i < size);
    }
    T *p;                           // implemented using dynamic array
    int size;
};

Vector<int> vi(10);             // 10 ints
Vector<const char*> vs(20);     // 20 C-strings

vi[0] = 10;                     // OK
vs[1] = "text";                 // OK
```

Pair Class Template

```cpp
template <typename T>
struct Pair {
    Pair(T v1, T v2); // implementations can be deferred
    T first, second;
};

// implementation can be provided outside class template
// definition but still in header file!
// Improves readability
template <typename T>
Pair<T>::Pair(T v1, T v2)
    : first(v1), second(v2)
{
    // ...
}

//... in main():
Pair<int> pi(0, 0);          // pair of ints
Pair<float> pf(0.0, 2);      // pair of floats

// pair of int pairs
Pair<Pair<int>> pp(Pair<int>(0, 2), Pair<int>(3, 1));
pi = pp.first;
```

Tricky Bits — Dependent Names

In class templates sometimes we have to clarify whether an identifier is a type or not

```cpp
template <typename T>
class X {
    typename T::SubType *ptr;
};
```

Without ***typename*** SubType would be considered a static member variable of type T! Thus, the compiler would view

```
T::SubType * ptr
```

as a multiplication!

When a name depends on a template type you need to use `typename`.

Tricky Bits — Using `this`

In class templates with base classes, using a name x by itself is not always equivalent to `this->x`, even though a member x is inherited.

```cpp
template <typename T>
class X {
public:
    void bar();
};

template <typename T>
class Y : public X<T> {
public:
    void foo() { bar(); }       // call global bar() if exists, error otherwise
};
```

Issue is that X is a template, so it is supposed to have template arguments. Solution:

```cpp
void foo() { this->bar(); }
```

or

```cpp
void foo() { X<T>::bar(); }
```

Tricky Bits — Using `this`

What about a templated dependent name?

```
template <typename T>
class X {
public:
    template <typename Y>                // Method template parameter Y is separate
    void bar();                          //   from class template parameter
};

template <typename T>
class Y : public X<T> {
public:
    void foo() { this->template bar<int>(); }
};
```

Tricky Bits — Explicit Call of Default Constructor

POD types such as int, char, double have no default constructor.

To ensure that template variables of such types are initialized we must call constructors explicitly:

```cpp
template <typename T>
void foo() {
    T x{};          // POD is initialized with 0
                    // T x; leaves x uninitialized for POD types T...
    // ...
}
```

Or

```cpp
template <typename T>
    class X {
    public:
        X() : a() {}    // this also works for POD T
    private:
        T a;
    };
```

Tricky Bits — Explicit Call of Default Constructor

This initialization works for all POD types, including structs, not just for class template variables, but also for arrays:

```
int *p = new int[100]{};
```

initializes all values with 0.

Template Specialization

Suppose the general implementation of a class template can be improved for specific types.

Example: `std::vector<bool>`

Bools occupy one byte each, a waste of 7 bits! A customized `vector<bool>` class template could store 8 bits in a byte instead.

```
template <typename T> class X { ... };                // (A) general form
template <> class X<bool> { ... };                    // (B) specialized form
template <> class X<int>  { ... };                    // (C) specialized form

X<float> x;            // instantiates (A)
X<bool> x;             // instantiates (B)
X<int> x;              // instantiates (C)
```

Adapts class templates to special needs. There is no relation between above implementations — except for the same class template name X!

Result: Different code for different types!

Flashback — Function Templates

There, too, we sometimes need specialization. Consider this min template:

```cpp
template <typename T>
T min(const T a, const T b) {
    return a < b ? a : b;
}
```

Does it work for T `= char *`?

```cpp
const char *foo = "foo", *bar = "bar";
std::cout << min(foo, bar) << std::endl;
```

The code compiles, but will do something unexpected ...

The result is kind of random, because < applied to pointers returns true iff the first pointer points to a lower address in memory than the second.

Oops. Our intention was to print the lexicographically smaller string

Flashback — Function Templates

Remedy: function template specialization! You can either write:

```cpp
template <>
const char *min<const char*>(const char *a, const char *b) {
    // compare the two C-strings lexicographically
    return strcmp(a, b) < 0 ? a : b;
}
```

or simply (preferred)

```cpp
const char *min(const char *a, const char *b) {
    return strcmp(a, b) < 0 ? a : b;
}
```

Specialized functions take precedence over templated versions in the process of the compiler searching for the function to call.

Is there any other benefit or flexibility for preferring overloads?

Best Practice: Overload when you can, specialize when you need to!

Partial Template Specialization

Allows us to specialize classes (but not functions) where some, but not all, of the template parameters have been explicit defined.

```cpp
// (A) general case
template <typename T>
class List { ... };


// (B) special case where we handle pointers to types: share void*
// implementation and use pointer casts ...
template <typename T>
class List<T*> {
    List<void*> impl;  // infinite recursion?!
    ...
    T* get(int i) const {
        return reinterpret_cast<T*>(impl.get(i));
    }
};


// (C) if we don't provide this, we run into an
// infinite recursion at compile-time:
template <> class List<void*> { ... };

List<int>   x;          // instantiates (A)
List<int*>  y;          // instantiates (B)
List<void*> z;          // instantiates (C)
```

Partial Template Specialization

- Compiler looks for best (= most specialized) type match
- What is the advantage?
- For pointer types we only fully instantiate `List<void*>` once
- All other `List<T*>` classes are just wrappers which delegate method calls to `impl`.
- Partial class template specialization adapts class templates even more.
- Compiler chooses the most specialized match.

More details in "***C++ Templates***" by Vandevoorde and Josuttis.

Type Traits

- We sometimes want to know something about types at compile-time, to create faster code, for instance.
- For this, we need to infer type properties at compile-time, a process known as compile-time type reflection.
- For example, POD types can be copied with memcpy, but others can't.
- A smart copy method could take advantage of this fact.
- **Refresher**: The code that follows makes use of enumeration types, which define integer constants in the program.

```cpp
// named integer constants, start with 0 and increment
enum { APPLE, ORANGE, PLUM };

struct X {
    // can evaluate constant expressions at compile-time
    enum { value = 5, foo = value + 1 };
};

cout << APPLE << " " << ORANGE << " " << PLUM << endl;
// 0 1 2
cout << X::value << " " << X::foo << endl;
// 5 6
```

Type Traits

With this and partial class template specialization we can find out whether a type is a pointer:

```cpp
template <typename T>
struct TypeTraits {
    // general case
    template <typename U>
    struct PointerTraits {
        enum { value = 0 };             // no pointer
    };

    // special case
    template <typename U>
    struct PointerTraits<U*> {
        enum { value = 1 };             // is pointer
    };

    enum { is_pointer = PointerTraits<T>::value };
    // other interesting type properties ...
    enum { is_reference = ... };
    enum { is_const = ... };
    enum { has_trivial_assign = ... };  // bit-wise copy OK?
};
```

Here, enums are initialized with the result of a recursive COMPILE-TIME COMPUTATION (such as `PointerTraits<T>::value`)

Type Traits

Why did we use enums on the previous example,

```cpp
template<typename>
struct is_pointer {
    enum { value = false };
};;

template<typename T>
struct is_pointer<T*> {
    enum { value = true };
};

bool b = is_pointer<void*>::value;
```

and not the following?

```cpp
template<typename>
struct is_pointer {
    enum { value = false };
};

template<typename T>
struct is_pointer<T*> {
    static const bool value = true;
};

bool b = is_pointer<void*>::value;      // Compiler error when linking
```

Type Traits

```cpp
template<typename>
struct is_pointer {
    static const bool value = false;        // Declaration, not a definition
};

template<typename T>
struct is_pointer<T*> {
    static const bool value = true;         // Declaration, not a definition
};

// We need the following, like we previously saw when we learned about static.
template<typename T>
const bool is_pointer<T>::value;

template<typename T>
const bool is_pointer<T*>::value;

bool b = is_pointer<void*>::value;
```

Type Traits

In C++17, we can do the following:

```cpp
template<typename>
struct is_pointer {
    static constexpr bool value = false;
};

template<typename T>
struct is_pointer<T*> {
    static constexpr bool value = true;
};

bool b = is_pointer<void*>::value;
```

Fast Array Copy Function

The fast memcpy function can be used to copy arrays holding primitive (POD) objects. Otherwise, we need to iterate assignments for all array elements.

Approach: class template that defines a flag telling the ` `smart'' copy routine when to use memcpy.

Goal:

```cpp
const int N = 1000;
int a[N];
int b[N];
copy(b, a, N);                             // uses memcpy: FAST

double a[N];
double b[N];
copy(b, a, N);                             // also uses memcpy

struct Bar { virtual ~Bar() {...} };       // not POD
Bar a[N];
Bar b[N];
copy(b, a, N);                             // uses Bar's AO: SLOW
```

Fast Array Copy Function

copy is a function template:

Goal:

```
template <typename T>
void copy(T *dst, T *src, int n) {
    CopyImpl<has_triv_assign<T>::value>::copy(dst, src, n);
    //                           *****  0: slow,  1: fast
}
```

The class in which we call the copy function is dependent on a compile-time constant that tells us whether T has a trivial assignment operator or not.

Note: here we use an int as template parameter.

Fast Array Copy Function

```cpp
// default (slow)
template <int U>
struct CopyImpl {
    // static method is a function template
    template <typename T>
    static void copy(T *dst, T *src, int n) {
        for (int i=0; i < n; ++i) {
            dst[i] = src[i];
        }
    }
};

// specialization (fast)
template <>
struct CopyImpl<1> {
    // static method is a function template
    template <typename T>
    static void copy(T *dst, T *src, int n) {
        memcpy(dst, src, n*sizeof(T));
    }
};
```

Fast Array Copy Function

How to figure out whether type T has a trivial assignment?

For example:

```cpp
// safe default: no trivial assignment
template <typename T>
struct has_triv_assign { enum { value=0 }; };

// basic types can be assigned trivially
template <>
struct has_triv_assign<char> { enum { value=1 }; };
...

template <>
struct has_triv_assign<double> { enum { value=1 }; };

// all pointers can be assigned trivially
// Shallow copy, not deep copy!
template <typename T>
struct has_triv_assign<T*> { enum { value=1 }; };
```

Fast Array Copy Function

For all other types let the user decide:

```cpp
struct Foo {    // can be assigned trivially
    int a, b;
};

// Followed by this line to make copy() aware of it
// User must opt-in to this
template <>
struct has_triv_assign<Foo> { enum { value=1 }; };

struct Bar {    // can't be assigned trivially
    virtual ~Bar() {...}
};
```

For Bar we don't have to say anything, because having no trivial assignment is the default.

Fast Array Copy Function

Result: copying 1,000,000 integers with the smart copy function is 2.3x faster (measured with time a.out) on my computer.

Starting with C++11, a wide variety of type traits are supported and defined in the `type_traits` header:

```cpp
#include <type_traits>
struct X { virtual ~X() { } };
cout << std::is_polymorphic<X>::value;            // 1
cout << std::is_polymorphic<X*>::value;           // 0
cout << std::is_trivially_copyable<X*>::value;    // 1
```

See https://en.cppreference.com/w/cpp/header/type_traits and the boost section below.

Compile Time Recursion

Factorial function:

$$0! = 1$$
$$n! = n * (n-1)!$$

for $n \geq 0$.

```cpp
template <int n>
struct Fac {                            // general case n! = n*(n-1)!
    // compile-time constant
    enum { value = n * Fac<n-1>::value };
};

template <>
struct Fac<0> {                         // base case 0! = 1
    enum { value = 1 };                 // compile-time constant
};

// Fac<N>::value is now a compile-time constant!
int main() {
    cout << Fac<5> ::value << endl;            // =  5!  = 120
    cout << Fac<10>::value << endl;            // = 10!  = 3628800
    cout << Fac<0> ::value << endl;            // =  0!  = 1
}
```

Compile Time Recursion

- Computation done at compile-time — beware, this can take a long time
- It has been shown that C++'s template mechanism is Turing complete
- This means that in principal you can run any kind of computation while compiling your program
- ***Upside***: compile-time computation speeds up runtime computation
- ***Downside***: you can't even be sure anymore whether compilation eventually stops
- Other useful applications: unrolling loops, peeling off recursion levels, etc.

Compile Time Recursion

How bad can it be? Try this:

```
// general case: type recursion
template<int Depth, int A, typename B>
struct X {
  static const int x =
    X <Depth-1, 0, X<Depth,A,B>>::x     // spawns two recursive calls
  + X <Depth-1, 1, X<Depth,A,B>>::x;    // can't reuse previous result (1 vs. 0)
};

// base case
template <int A, typename B>
struct X<0,A,B> { static const int x = 1; };

static const int z = X<17,0,int>::x;

int main() {
    return 0;
}
```

On my computer this program takes 17 seconds to compile. Replace the 17 by 18, 19, 20... and enjoy ... with each increment, the compilation time roughly doubles

Boost: A Collection of C++ Libraries

Playground for C++ libraries available at www.boost.org. Community proposes/reviews/improves libraries.

Large variety of useful libraries:

- regular expressions, formatted output, type traits,
- portable threading, random number generators, ...

Some of them like `shared_ptr` and `type_traits` already made it into the C++ standard.

To use boost libraries with g++ include the need boost header files, e.g.

```
#include <boost/filesystem.hpp>
```

and link with libraries (if necessary, most boost libraries are just headers)

```
g++ -o hello hello.cpp -lboost_filesystem
```

Boost: Format

Formatting output with output stream operator << can be a pain (look it up, it's ugly!)

If you like C's `printf`, you'll love this:

```
cout << boost::format("%+4d %.2f %40s") % 399 % 1.34567 % "foo";
                            |                     |      |       |
                printf format string   arguments separated by %,
                                        those match the % in the
                                        format string
```

If you get the number of parameters or their type wrong, the runtime system will let you know.

The `fmt` library https://github.com/fmtlib/fmt is also a good alternative, which C++20s `std::format` is based off of.

Arrays

C++ arrays are safe (and equally fast) replacements for standard C-arrays. They made it into the C++11 standard after being tested in the Boost library.

```
#include <array>

std::array<int, 100> a;         // array of 100 ints

a[99] = 1;                      // C-array syntax
a[100] = 0;                     // runtime error in debug mode (see below)
```

g++ generates `std::array` index bound checks when using

`-D_GLIBCXX_DEBUG`

Why is this opt-in?

I suggest to say good-bye to unsafe C-arrays and use `std::array` as safe replacement in your code.