# Generic functions & Custom types

Laurent Mathy

Object-Oriented Programming Projects

April 20, 2020

# Outline

# Generic functions

Functions whose argument/result types are unknown until use.

- `find` can find value of any *appropriate* type in any container.
- *any appropriate type*
  - language support: ways in which a function uses a parameter constrains the possible parameter type. Operations used must be supported by type.
  - organisational: set of operations assumed to be supported by type (*e.g.* iterators).

# Template functions

```
template<class T>
T median(std::vector<T> v) {
    auto size = v.size();
    if (size == 0)
        throw std::domain_error("median of an empty vector");
    sort(v.begin(), v.end());
    auto mid = size / 2;
    return size % 2 == 0 ? (v[mid] + v[mid - 1]) / 2 : v[mid];
}
```

- **Type parameter** `T` is a name valid in the function's scope.
- Two equivalent definitions:
  - `template<class T>`
  - `template<typename T>`
- `T` is bound to a real type based on argument type passed to function call, at **compile time**.
- Compiler **instantiates** a specific version of function for each actual type used.

# Template function instantiation

- if `vi` is a variable defined to be of type `vector<int>`, then a call to `median(vi)` binds `T` to **`int`**:
  - whenever `T` is used, then the compiler replaces it by **`int`**.
- if you call `median` with a `vector<double>`, compiler generates an instance of `median` with `T` bound to **`double`**.
- Some compilers do template function instantiation at *compile* time, others at **link** time.
  - Be ready to see *compile* errors at link time!
- Most implementations require that template **definition**, not just declaration, be available during instantiation.
  - Put template function body in the header file.

# Beware of interactions between templates and type conversions

- `find(s.homeworks.begin(), s.homeworks.end(), 0);`
  - `homeworks` is a `vector<double>`, but asking to look for an `int`.
  - This is OK as can compare `int` to `double` without issue.
- `accumulate(v.begin(), v.end(), 0.0);`
  - `accumulate` uses type of third argument as return type.
  - Pass `0` instead of `0.0` and you'll accumulate into an `int`.
- `max(4, 3.14)`

```
template<class T> T max(const T& left, const T& right) {
    return left > right ? left : right;
}
```

  - Can't pass an `int` and a `double`: which one should the compiler choose to bind to `T`?

# **typename**

**typename** must be used to qualify declarations that use types that are defined by the template type parameters. *E.g.*,

- **typename** `T::size_type len;`
  declares `len` to have type `size_type`, which must be defined as a type inside `T`.
- **typedef typename** `vector<T>::size_type vec_sz;`

# Outline

# Use type inference for generic code

Make template code more generic with `auto` and `decltype`.

```
template<class T> void f(T x) {
    auto len = x.size();
    // ...
}
```

We don't care whether `size()` returns a `T::size_type`, a `size_t`, an `unsigned long`, a `short int`, ...

The compiler will replace `decltype(expr)` with the type of expression `expr`. This replacement is done *statically*, *i.e.* at **compile time**, and `expr` is **not** evaluated. *E.g.*,

```
template<class T> void f(T x) {
  for (decltype(x.size()) i = 0; i < x.size(); ++i)
      // ...
}
```

# Use iterators to write generic code

- Write functions independent of container where data is stored.
- `iterators` refer to elements in a container, not to the container itself!
  - Using iterators allows to specify ranges **inside** the container.
  - Algorithm implementation outside of container implementation.
  - Iterators can *extend* container capabilities: *e.g.* reverse iterator.
- Some containers support operations that others don't: iterator design will reflect this.
- Not all algorithms require all iterator operations.

$\Longrightarrow$ **iterator categories**.

# Iterator categories: input iterators

```
template<class In, class X>
In find(In begin, In end, const X& x) {
      while (begin != end && *begin != x)
          ++begin;
      return begin;
}
```

Supports:

- `++` (prefix and postfix)
- `==` and `!=`
- unary `*`
- `->`

# Iterator categories: output iterators

```
template<class In, class Out>
Out copy(In begin, In end, Out dest) {
    while (begin != end)
        *dest++ = *begin++;
    return dest;
}
```

Supports:

- ++ (prefix and postfix)
- = (value assignment)
- **write-once**
    - ++ no more than once between assignments.
    - No more than one assignment without increment.
    - Developer's responsibility!
- `back_inserter` generates an output iterator.

# Iterator categories: forward iterators

```
template<class For, class X>
void replace(For beg, For end, const X& x, const X& y)
{
    while (beg != end) {
        if (*beg == x)
            *beg = y;
        ++beg;
    }
}
```

Supports:

- `++` (prefix and postfix)
- `==` and `!=`
- Unary `*` (both reading and writing)
- `->`

# Iterator categories: bidirectional iterators

```
template<class Bi>
void reverse(Bi begin, Bi end) {
    while (begin != end) {
        --end;
        if (begin != end)
            std::swap(*begin++, *end);
    }
}
```

Supports:

- Forward iterator operations.
- `--` (both prefix and postfix)

# Iterator categories: random-access iterators

```cpp
template<class Ran, class X>
bool binary_search(Ran begin, Ran end, const X& x) {
    while (begin < end) {
        // Find midpoint of range
        Ran mid = begin + (end - begin) / 2;
        // See which sub-range contains `x`, and adapt range
        if (x < *mid) end = mid;
        else if (*mid < x) begin = mid + 1;
        else return true; // `*mid == x` so we're done
    }
    return false;
}
```

Supports:

- Bidirectional iterator operations.
- Let `p`, `q` be iterators, and `n` an integer:
  - `p + n`, `p - n` and `n + p`.
  - `p - q` distance between iterators as integral type.
  - `p[n]` equivalent to `*(p+n)`.
  - `p < q`, `p > q` , `p <= q` and `p >= q`.

# Input/Output stream iterators

In `<iterator>` header.

Input stream iterator: `istream_iterator`.

```
typedef typename Seq::value_type Elem;
copy(istream_iterator<Elem>(cin),
     istream_iterator<Elem>(),
     back_inserter(xs));
```

`istream_iterator` default value represents *end-of-file*.

Output stream iterator: `ostream_iterator`.

```
copy(xs.begin(), xs.end(),
     ostream_iterator<Elem>(cout, " "));
cout << endl;
```

Second argument to `ostream_operator()` is separator.

# Using iterators for flexibility

```
template<class Out>                               // Changed
void split(const std::string& str, Out os) { // Changed
    typedef std::string::const_iterator Iter;

    Iter i = str.begin();
    while (i != str.end()) {
        // Ignore leading blanks
        i = std::find_if_not(i, str.end(), isspace);
        // find end of next word
        Iter j = std::find_if(i, str.end(), isspace);
        // Copy characters in `[i, j)`
        if (i != str.end())
            *os++ = std::string(i, j);        // Changed
        i = j;
    }
}
```

Can pass a list iterator, vector iterator, output stream iterator, *etc.*

# Outline

# Object-based programming

C++ has two kinds of types: built-in types and class types.

Pervasive idea in C++: let users create types that are as easy to use as built-in types.

Starting our study of *Object-based programming* with C++.

# Why Object-based programming?

- Separation of **interface** and **implementation**.
- Can control initialisation of objects (*i.e.* make sure they are created in a consistent state).
- Can enforce object properties through language features (*e.g.* immutability).

# Member functions

```
struct Student_info {
    std::string name;
    double midterm, final;
    std::vector<double> homeworks;

    std::istream& read(std::istream&);  // New
    double grade() const;               // New
};
```

- `Student_info` has four **data members** (*a.k.a.* fields) and two **member functions** (*a.k.a.* methods).
- `const` after the declaration of grade is a promise that grade will not change any of the data members of the object.
- Use of member function, given `Student_info s`:
  - `s.read(cin);`
  - `s.grade();`

# Member function definition

```
istream& Student_info::read(istream& in) {
    in >> name >> midterm >> final;
    read_hws(in, homeworks);
    return in;
}
```

- Type declarations in header file (e.g. `Student_info.hpp`) and function definitions in common source file (*e.g.* `Student_info.cpp`).
- Function name is `Student_info::read` instead of `read`.
  - scope operator `::` defines the function `read` to be a member of `Student_info` type.
- No need to pass a `Student_info` object as argument: object is implicit in call.
- Function can access data member of object *directly*.

# Member function definition (2)

```
double Student_info::grade() const {
    return ::grade(midterm, final, homeworks);
}
```

- grade is a `const` member function:
  - can call `const` function on any objects;
  - cannot call non-`const` functions on `const` objects.
- `::` in front of a name insists on using version that is not a member of anything.
  - Without it, compiler would look for `Student_info::grade` and complain about argument mismatch.

# Protection

User of our `Student_info` type no longer *have to* manipulate object internals, but they still *can*.

C++ supports access control through `public` and `private` **access specifiers**.

```
class Student_info {
public:
    // Interface
    double grade() const;
    std::istream& read(std::istream&);

private:
    // Implementation
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};
```

# Access specifiers

- Access specifier defines accessibility for all members that follow it (until next access specifier).
- Access specifiers can occur in any order and multiple times.
- Compiler enforces protection.
- Difference between `struct` and `class`:
  - *Only* difference is default protection of members until first protection label
  - `struct`: default protection is `public`.
  - `class`: default protection is `private`.

# Student_info class

```
class Student_info {
public:
    std::string name() const { return _name; }
    bool valid() const { return !homeworks.empty(); }
    std::istream& read(std::istream&);
    double grade() const;
private:
    std::string _name; // Changed to avoid confusion with name()
    double midterm, final;
    std::vector<double> homeworks;
};
```

Note that `name` and `valid` functions are defined in the header file: this is a hint for compiler to avoid function calls by making it **`inline`**.

# Constructors

What is the state of a new object?

- **Constructors** are special member functions that define how objects are initialised.
- No way to call constructors explicitly: they are called as side-effect of object creation.
- If we do not define any constructor, the compiler synthesizes one for us.
- All data members are initialized:
  - Objects with local scope are **default-initialized**.
  - Objects used to initialize container elements are **value-initialized**.

# Default vs value initialisation

- If class has one or more constructors, the appropriate one is called (full initialisation control);
- If object is built-in type: value-initialisation sets it to zero, default-initialisation sets it to **any** value (garbage);
- If class has no constructor: synthetic constructor that recursively value/default-initialise the data members.

`Student_info` has no constructor:

- Members `_name` and `homeworks` automatically initialised as empty `string` and `vector` (because that's what the corresponding default constructors for the corresponding classes do).
- `midterm` and `final` get garbage in case of default-initialisation; 0.0 in case of value-initialisation.

# Constructor definitions

The **default constructor** takes no argument.
Its job is to make sure data members are *always* initialised properly.

```
class Student_info {
public:
    Student_info(); // Default constructor
    Student_info(std::istream&); // Constructs object by reading stream
// As before ...
};

Student_info::Student_info(): midterm(0), final(0) {}
```

This default constructor uses a constructor **initialisation list**.
When an object is constructed:

- Memory is allocated for object.
- *All* members are initialized in the order of declaration in class (even members not in initializer list – but if in constructor initialisation list, then get corresponding value).
- Constructor body is run (so body can *change* initial values).
- `_name` and `homeworks` initialised by their default constructor.

# Constructor definitions (2)

```
Student_info::Student_info(istream&is) {
    read(is);
}
```

- No explicit initializers, so `_name` and `homeworks` get initialized to *empty* values by their default constructor.
- `midterm` and **`final`** only get initialised to meaningful values if object being value-initialised.
- `read` then explicitly changes the values.

# Using the Student_info class

```
// Read all the records, and find the length of the longest name
Student_info record;
vector<Student_info> students;
string::size_type maxlen = 0;
while (record.read(cin)) {                          // Changed
    maxlen = max(maxlen, record.name().size());     // Changed
    students.push_back(record);
}

// Alphabetize the records
sort(students.begin(), students.end(), compare);

auto prec = cout.precision(3);
for (vector<Student_info>::size_type i = 0;
        i != students.size(); ++i) {
    // Write the name, padded on the right
    cout << students[i].name() // This and next line changed
         << string(maxlen + 1 - students[i].name().size(), ' ');
    // Compute and write the grade
    try {
        double final_grade = students[i].grade(); // Changed
        cout << final_grade << endl;
    } catch (domain_error e) {
        cerr << e.what() << endl;
    }
}
cout.precision(prec); // Restore precision
```