Programming in C++

Session 4 – Genericity, Containers

Dr Christos Kloukinas

City, UoL

https://staff.city.ac.uk/c.kloukinas/cpp

(slides originally produced by Dr Ross Paterson)

Polymorphism

Code that works for many types.

- ad-hoc polymorphism (overloading)
- subtype polymorphism (dynamic binding)
- parametric polymorphism (genericity)

See also:

- Savitch, sections 16.1–2 and 19.1.
- Stroustrup, chapter 13 (sections 2 and 3)
- Horstmann, section 13.5

A problem of reuse

- Often an operation looks much the same for values of different types.
- This situation is common with operations on *container* types, such as vectors, lists, stacks, trees, tables, etc.
  For example reversing a vector looks much the same whatever the types of the elements.
- Reuse: separate what varies (the type of the elements) from what doesn't (the code), and reuse the latter.
- Instead of writing many similar versions, we will write one generic implementation (parameterized by type), and reuse it for various types.

Swapping arguments

Swapping a pair of integers:

```
void swap(int & x, int & y) {
        int tmp = x; x = y; y = tmp;
}
```

x & **y** are references, i.e., aliases of real objects - so what does **swap** do? Copies CONTENTS !!!

Swapping a pair of strings is very similar:

```
void swap(string & x, string & y) {
        string tmp = x; x = y; y = tmp;
}
```

And so on for every other type.

Idea: make the type a parameter, and instantiate it to **int**, **string** or any other type.

A generic swapping procedure

Instead of the preceding versions, we can write:

```
template <typename T>
void swap(T & x, T & y) {
        T tmp = x; x = y; y = tmp;
}
```

Here **T** is a *type parameter*. When we use this function, **T** is instantiated to the required type:

```
int i, j;
swap(i, j);     // T is int
string s, t;
swap(s, t);     // T is string
```

but in each use **T** must stand for a single type.

A generic swapping procedure
Instead of the preceding versions, we can write:
template <typename T>
void swap(T & x, T & y) {
T tmp = x; x = y; y = tmp;
}
Here T is a type parameter. When we use this function, T is instantiated to the required type:
int i, j;
swap(i, j); // T is int
string s, t;
swap(s, t); // T is string
but in each use T must stand for a single type.

```
template <typename T>
void swap(T & x, T & y) {
        T tmp = x; x = y; y = tmp;
}
```

What is the ***interface*** of class **T** we use here?

- In `T tmp = x;` we introduce a new variable of type **T** and ***initialise*** it with **x**.

  This calls the ***copy constructor*** of class **T** – can you see why it's that constructor?

  `T( const T & o );`
- In `x = y;` we are ***assigning*** **y** into **x**.

  This calls the ***assignment operator*** of class **T**.

  ```
  T & operator=( const T & o );
  // form 1 - member function (*almost always*)
  ```
- In `y = tmp;` we are ***assigning*** **tmp** into **y**.

  This calls the ***assignment operator*** of class **T** again.

  `T & operator=( const T & o );`

You should be able to understand why these functions are called. If not, please post on Moodle.

Writing generic code

- Prefix the function (or class) with

  `template <typename T>`

  and then **T** stands for a type, which will be supplied when the function or class is used.
- You can equivalently use **class** instead of **typename** (and some old compilers do not recognize **typename**).
- Multiple parameters are also permitted:

  `template <typename Key, typename Value>`

Reversing a vector of integers

```
void reverse(vector<int> & v) {
        int l = 0;
        int r = v.size()-1;
        while (l < r) {
                swap(v[l], v[r]);
                ++l; // *prefer* over l++
                --r; // *prefer* over r--
        }
}
```

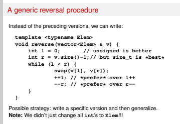
Reversing a vector of strings is the same, except for **string** instead of **int** as the element type.

A generic reversal procedure

Instead of the preceding versions, we can write:

```
template <typename Elem>
void reverse(vector<Elem> & v) {
    int l = 0;        // unsigned is better
    int r = v.size()-1;// but size_t is *best*
    while (l < r) {
            swap(v[l], v[r]);
            ++l; // *prefer* over l++
            --r; // *prefer* over r--
    }
}
```

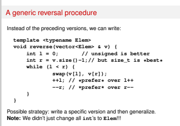
Possible strategy: write a specific version and then generalize.
Note: We didn't just change all **int**'s to **Elem**!!!



- Actually, the type of the indices shouldn't have been **int**
- They're supposed to hold non-negative values, so they should be **unsigned**
- And since they need to represent the length of an array, they should actually have been **std::size_t**, *according to the C++ standard*.
- **std::size_t** is an unsigned integer type, that is long enough to hold the length of an array (**unsigned int** might not be long enough).

```
template <typename Elem>
void reverse(vector<Elem> & v) {
    std::size_t l = 0;
    std::size_t r = v.size()-1;
    while (l < r) {
            swap(v[l], v[r]);
            ++l; // *prefer* over l++
            --r; // *prefer* over r--
    }
}
```

Well-known (*very* well-known!) C++ experts claim that **std::size_t** was defined wrongly in the standard and should have been a signed type, since that would have avoided a number of bugs when writing loops (comparison of signed and unsigned values and the fact that unsigned variables loop when over/under-flowing, while signed variables don't loop).
As such, they advise to use **int** instead of **size_t**. But doing so is going to produce compilation warnings. Compilation warnings are an indication that your code is incorrect (indeed it will be if the array/vector has more elements than an **int** can index).
To resolve this, avoid writing loops that use an "integer" index – prefer to use *range-based for loops* instead where applicable:
en.cppreference.com/w/cpp/language/range-for
Here we need two index (offset really) values, so a range-based for loop is not applicable – we need to use the **begin** and **end** iterators instead (more on these when we consider pointers) – see next note.



Looping using iterators instead of offsets:

```
template <typename Elem> // now impl works for lists too!
void reverse(vector<Elem> & v) {
    auto l = begin(v);
    auto r = end(v);
    // r points one element *after* the right target.
    while (l != r) {
            if (l == --r) return;
            swap(*l, *r); // *iterator = element
            ++l; // *prefer* over l++
    }
}
```

See p. 173 of Stepanov's "Elements of Programming"
elementsofprogramming.com/
Even better – use one of the standard C++ algorithms if applicable!
en.cppreference.com/w/cpp/algorithm

Hey, can you print the array elements in reverse order here? (see code commented out at the bottom)
coliru.stacked-crooked.com/a/2c2dc58a2c81fc8c

Using the generic procedure

We can call **reverse** with vectors of any type, and get a special version for that type:

```
vector<int> vi;
vector<string> vs;
...
reverse(vi);          // Elem = int
reverse(vs);          // Elem = string
```

This works for any type:

```
vector<vector<int> > vvi;
...
reverse(vvi);         // Elem = vector<int>
```

(reversing a vector of vectors may seem expensive but a vector's swap has been optimised)

Implementation methods

Code sharing: a single instance of the generic code is generated, and shared between all uses. This requires a common representation for types, and is often used in functional languages.
In Java too: **`Object`**.

Instantiation (or specialisation): an instance of the code is generated for each specific type given as an argument, possibly avoiding unused instances (C++).
Caution: these methods are only instantiated (and fully checked) when used.

Another example

Testing whether a value occurs in a vector (algo std::find):

```
template <typename Elem>
bool member(const Elem & x, const vector<Elem> & v){
     // v & x are const - cannot modify them!!!
      for ( std::size_t  i = 0; i < v.size(); ++i)
        if (v[i] == x)
           return true;
     return false;
}
```

The generic definition of **`member`** only makes sense

1. If the operator **`==`** is defined for **`Elem`**.
2. And if **`operator==`** promises not to modify **`v[i]`** or **`x`**.
3. And if **`operator[]`** promises not to modify **`v`**
4. And if **`size`** promises not to modify **`v`**. . .

⇒How can you optimise **`member`** ? *(apart from using std::find instead)*

- What will happen if we write **`if (v[i] = x)`** instead of **`if (v[i] == x)`**?

  Parameter **`v`** has been declared as a **`const`** reference, so the compiler will catch the error – ***use `const` as much as possible!***

- How can you optimise the loop? It keeps computing **`v.size()`** on each iteration.

  - Optimisation 1:

```
template <typename Elem>
bool member(const Elem & x, const vector<Elem> & v) {
 size_t i = v.size();
 if (0 == i) return false; // no elements
 for (i -= 1; 0 < i ; --i) // backwards search
    if (v[i] == x) return true;
 return (v[0] == x); // v[0] exists: v.size() != 0
}
```

  - Optimisation 2: Best because simplest.

```
template <typename Elem>
bool member(const Elem & x, const vector<Elem> & v) {
 for (size_t i = 0, limit = v.size(); i < limit; ++i)
    if (v[i] == x) return true;
 return false;
}
```

Since **`v`** is **`const`** the compiler might be able to optimise the original code – ***use `const` as much as possible!***
Note: **`Elem x`** does not promise the compiler that we'll treat **`v`** as a constant inside **`member`**.
`const Elem & x` does promise that (and avoids copying potentially large objects).

Bounded genericity

- Sometimes a generic definition makes use of functions or member functions that are not defined for all types (e.g. **`member`** uses **`==`**).
- In C++, this is checked when the definition is specialized for some type. (Unused functions are not specialized.)
- In some other languages, **`T`** might be constrained to be a subtype of a class that provides the required operations, *e.g.*, in Java: **`List< ? extends Serializable > myList;`**

Since C++20, one can use concepts to provide bounds for the generic types: en.cppreference.com/w/cpp/concepts

A generic class

The following class is defined in `<utility>`:

```
template <typename A, typename B>
class pair {
public:
        A first;     // Members are
        B second;    // public!
        pair(const A& a, const B& b) :
            first(a), second(b) {}
};
```

Some `pair` objects:

```
pair<int, int> p(3, 4);
pair<int, string> n(12, "twelve");
```

Note we must specify the type arguments (unlike generic functions).

- Why not use a `vector<int> p = {3, 4};` instead of `pair<int, int> p(3,4);`?
  - Apples 'n' oranges. . .
    - When using a vector you are stating that all its elements are of the ***same*** type.
    - When using a pair you are stating that the two elements are of different types, even if they happen to be represented by the same basic type.
      Number of apples and number of oranges – this cannot be stored in a vector.
    - Plus – a vector allows enlarging/reducing its size, while a pair always has exactly two elements.
  - A pair is more efficient than a vector (less space, faster).
- Why not use a `int p[2] = {3, 4};` instead of `pair<int, int> p(3, 4);`?
  - Apples 'n' oranges. . . (a vector is a generalisation of an array)

Have you noticed the ***initializer list constructors***?
`vector<int> p1 = {3, 4}; int p2[2] = {3, 4};`
https://www.cplusplus.com/reference/initializer_list/initializer_list/

Container classes in the STL

The *Standard Template Library* is part of the C++ standard library, and provides several template classes, including

- Containers
  - Sequences
    - `vector`
    - `deque`
    - `list`
  - Associative Containers
    - `set`
    - `map`
- Iterators

See en.cppreference.com/w/cpp/container

Just taught you about ***`deque`*** *and* ***`set`****!* ***`:-)`***

The vector class

```
template <typename T>
class vector {
public:
  vector();
  vector(size_t initial_size);
  size_t size() const;
  void clear();
  const T & operator[](size_t offset) const;//The Good
        T & operator[](size_t offset)      ;//& the Bad
  const T & front() const { return operator[](0); }
        T & front()       { return operator[](0); }
  const T & back() const{return operator[](size()-1);}
        T & back()      {return operator[](size()-1);}
  void push_back(const T & x);
  void pop_back();
};
```


- Why do we return a `T &`?
  So that we can ***assign*** into the returned value.
  That's why we can write `v[i] = 3;` – what `operator[]` returns is a reference, so it's assignable.
- Note that for the compiler, `v[i]` is actually `v.operator[](i)`

Another container: lists

- A list is a sequence of items of the same type, that may be efficiently modified at the ends.
- We may access the first or last elements, add elements at ***either*** end and remove elements from ***either*** end.
- All these operations are fast, independently of the size of the list.
- Lists are implemented as linked structures, using pointers.
- Other uses of lists require *iterators* (covered next session).

The list class

```
template <typename T> class list {
public:
    list();
    size_t size() const;
    void clear();
    const T & front() const ; // The Good
          T & front()       ; // & the Bad
    void push_front(const T & x);
    void pop_front();
    const T & back() const ; // The Good
          T & back()       ; // & the Bad
    void push_back(const T & x);
    void pop_back();
};
```

Missing: `operator[]` – too slow with lists!
(just like `push/pop_front` *is too slow with vectors)*

Using a list

Reversing the order of the input lines:

```
list<string> stack;
string s;
while (getline(cin, s))
        stack.push_back(s);
while (stack.size() > 0) {
        cout << stack.back() << '\n';
        stack.pop_back();
}
```

- Can we implement this with vectors?
  Yes – vectors support `back`, `push_back`, and `pop_back`.
- What if we had used `push_front` and `pop_front` instead?
  No.

⇒ Use APIs that are supported by most containers, to make it easy to change the container.

Commonality between STL containers (pre C++20!)

- `push_back`, `size`, `back` and `pop_back` common to `list` and `vector`
- Use vectors instead? Only a small change is required!
- Those common methods could have been inherited from a common parent class, but the STL designers decided not to. The various STL classes use common names, but this commonality is not enforced by the compiler (it is since C++20! – ***concepts!***).
- It is not possible to use subtype polymorphism with STL containers (but is possible with other container libraries).
  - How come?
    Because the use of subtype polymorphism (*a.k.a.* inheritance) has an extra cost.
    (Non-overridable member functions are faster than overridable ones – more when we look at inheritance)

Requirements on containers in the STL

- A ***Container*** has methods

  ```
  size_t size() const;
  void clear();
  ```

  with appropriate properties.
- A ***Sequence*** has these plus

  ```
  T & front() const;
  T & back() const;
  void push_back(const T & x);
  void pop_back();
  ```

But Container, Sequence, *etc.* are not C++ (in C++20 they are!): they do not appear in programs, and so cannot be checked by compilers.

Some STL terminology

The STL documentation uses the following terms:

- A ***concept*** is a set of requirements on a type (e.g., an interface). Examples are Container, Sequence and Associative Container.
- A type that satisfies these properties is called a ***model*** of the concept.
  For example, `vector` is a model of Container and Sequence.
- A concept is said to be a ***refinement*** of another if all its models are models of the other concept.
  For example, Sequence is a refinement of Container.

Remember that all this is outside the C++ language.
Note: The C++ standard committee has made concepts part of the language and thus testable by the compilers. (since C++20)
See standard ones:
https://en.cppreference.com/w/cpp/named_req

New template classes from old

Often template classes are built using existing template classes. The following is defined in `<stack>`:

```
template <typename Item>
class stack {
    vector<Item> v;
public:
    bool empty() const { return v.size() == 0; }
    void push(const Item & x) { v.push_back(x); }
    const Item & top() const { return v.back(); }
          Item & top()       { return v.back(); }
    void pop() { v.pop_back(); }
};
```


Defining methods outside the class

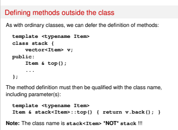
As with ordinary classes, we can defer the definition of methods:

```
template <typename Item>
class stack {
    vector<Item> v;
public:
    Item & top();
    ...
};
```

The method definition must then be qualified with the class name, including parameter(s):

```
template <typename Item>
Item & stack<Item>::top() { return v.back(); }
```

Note: The class name is `stack<Item>` ***NOT*** `stack` !!!



- Note that the full name of the class is `stack<Item>` as `stack` is a generic class.
  So it's
  `Item & stack<Item>::top() {...`
  and not
  `Item & stack::top() {...`
- Also note that the definition needs to be preceded again by `template <typename Item>`, just like the original class, because the class name contains a type parameter.
  So it's

  ```
  template <typename Item>
  Item & stack<Item>::top() { return v.back(); }
  ```

  and not just

  ```
  Item & stack<Item>::top() { return v.back(); }
  ```

Maps

A map is used like an vector, but may be indexed by any type:

```
map<string, int> days;
days["January"] = 31;
days["February"] = 28;
days["March"] = 31;
...
string m;
cout << m << " has " << days[m] << " days\n";
cout << "There are " << days.size() << " months\n";
```

This is a mapping from strings to integers.

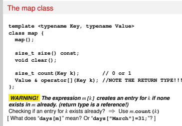
The map class

```
template <typename Key, typename Value>
class map {
  map();

  size_t size() const;
  void clear();

  size_t count(Key k);       // 0 or 1
  Value & operator[](Key k); //NOTE THE RETURN TYPE!!!
};
```

***WARNING!* The expression m `[`k`]` creates an entry for k if none exists in m already. (return type is a reference!)**

Checking if an entry for k exists already? $\Rightarrow$ Use m`.count(`k`)`

[What does "`days[m]`" mean? Or "`days["March"]=31;`"?]



- What does "`days[m]`" mean?
  `days[m]` $\equiv$ `days.operator[]( m )`
  `days["March"]= 31` $\equiv$ `days.operator[]("March")= 31;`
- Why does m `[`k`]` create an entry for k if none exists in m already?
  Because `operator[]` needs to be able to return a reference to an existing element (it returns `Value &`!).

Summary

- Generic code is parameterized by a type `T`, and does the same thing for each type.
- To use a generic class, we supply a specific type, which replaces each use of `T` in the definition.
- One way to write a generic class is to write it for a specific type, and then generalize.
- The Standard Template Library includes many useful template classes.
- The STL has a hierarchical organization, but does not use class inheritance (because inheritance introduces extra costs).
- STL uses concepts instead (compiler checked since C++20)

Next session

- Arrays and pointers in C++ (Savitch 10.1; Stroustrup 5.1–3, Horstmann 9.7): a low-level concept we usually avoid.
- *Iterators*: classes that provide sequential access to the elements of containers.
- Iterators in the STL (Savitch 17.3,19.2; Stroustrup 19.1–2) are analogous to pointers to arrays.

Next session

- Arrays and pointers in C++ (Savitch 10.1; Stroustrup 5.1–3, Horstmann 9.7): a low-level concept we usually avoid.
- *Iterators*: classes that provide sequential access to the elements of containers.
- Iterators in the STL (Savitch 17.3,19.2; Stroustrup 19.1–2) are analogous to pointers to arrays.

Final Notes – I

- Humans shouldn't have to write the same code over and over for parameters of type **`int`**, **`char`**, **`float`**, **`big_huge_object`**, etc. We have the right to say it once and have it work for any type (any type that makes sense): **GENERIC PROGRAMMING**

  ```
  // this is a code *template* - T is some name type
  template <typename T>
  void swap( T & x, T & y ) {// x & y of the same type T
    T tmp = x; // calls T's copy-constructor:
    // T(const T &other)

    x = y; // calls T's assignment operator:
   //T & operator=( const T & b ) // "method"


    y = tmp; // assignment operator again:
   //T & operator=( const T & b)
  }
  ```

  See also: "Template Classes in C++ tutorial" (https://www.cprogramming.com/tutorial/templates.html)
- Strategy: write normal code, then generalize it (easier to debug this way!)

2023-11-20
Programming in C++
└Next session

Next session

- Arrays and pointers in C++ (Savitch 10.1; Stroustrup 5.1–3, Horstmann 9.7): a low-level concept we usually avoid.
- *Iterators*: classes that provide sequential access to the elements of containers.
- Iterators in the STL (Savitch 17.3,19.2; Stroustrup 19.1–2) are analogous to pointers to arrays.

Final Notes – II

- Java vs C++ implementation strategies (slide 10):
  - Java produces one version, where **`T`** has been replaced by **`Object`** (a pointer to any kind of object) or a class that's sufficiently generic.

    Good:
    - Java checks your generic code (*).
    - Java doesn't suffer code-bloat – only one version of the code in the program.

    Bad:
    - Java doesn't take advantage of the type parameter to specialize the code for that specific type.
  - In C++ generic code is instantiated, specialized, and checked when it's used – otherwise it's ignored (and so are the bugs in it).

    Good:
    - Type-specific optimized code!
    - Checks at compile time that the type parameter works with this code! (The Java compiler does check but also adds a number of run-time casts (*) – so you can get a run-time exception in it due to type incompatibility, he, he, he...)

    Bad:
    - No checks when the code isn't used.
    - Code-bloat – one version for each type parameter.

(*) "Type erasure" (https://docs.oracle.com/javase/tutorial/java/generics/erasure.html), which leads to a number of "Java restrictions on generic code" (https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html). (advanced – not to be assessed – for curious cats only)

2023-11-20
Programming in C++
└Next session

Next session

- Arrays and pointers in C++ (Savitch 10.1; Stroustrup 5.1–3, Horstmann 9.7): a low-level concept we usually avoid.
- *Iterators*: classes that provide sequential access to the elements of containers.
- Iterators in the STL (Savitch 17.3,19.2; Stroustrup 19.1–2) are analogous to pointers to arrays.

Final Notes – III

- **`vector`**, **`list`**, commonality between STL containers (slides 19–21 – STL container "inheritance" done manually, for increased speed)
- new template classes from old (slide 22),
- syntax for defining generic member functions outside their generic class (slide 23), and maps (slides 24–25)