IN2029: Programming in C++

Session 10 – Revision

Vahid Rafe

Department of Computer Science
City, University of London

Autumn term, 2025

Overview

Assessment in this module:

Coursework (30% of the module)

Exam (70% of the module): two questions

To pass the module, you need an overall module mark of at least 40.

If you don't manage that, you will need to retake the component(s) for which you obtained less than 40. Any retake marks will be capped at 40. The marks will then be combined again, and need to be at least 40.

Big variables

Unlike in Java, in C++ variables of class type (including **`string`**, **`vector`**, etc) contain whole objects. Consequences:

- We try to avoid copying these objects (session 3).
- Such a variable is initialized when it is declared, so

```
my_type v;
```

  implicitly invokes the default constructor of **`my_type`**. We can supply arguments to use a different constructor (session 6):

```
my_type v("foo", 5);
```

- Copying from a derived type involves slicing (session 7).
- We cannot declare variables of abstract class type (session 8).

Parameter passing modes (session 3)

by value The parameter `v` is a new variable, initialized as a copy of the argument to the function.

```
void f(my_type v);
```

by non-const reference The parameter `v` is an alias for the argument to the function (no copying), and any change to `v` happens to the argument.

```
void f(my_type &v);
```

by const reference The parameter `v` is an alias for the argument to the function (no copying), but `v` cannot be modified.

```
void f(const my_type &v);
```

These modes can also be used for function returns.

Other uses of `const`

The `const` specifier can also be used

- to declare global constants (session 3), *e.g.*

```
const int days_per_week = 7;
```

- to declare parameters and local variables with fixed values, *e.g.*

```
    const unsigned middle = v.size()/2;
```

- to declare that a member function does not change the state of the object it is called on (session 6):

```
    int x() const { return _x; }
```

Advice (*Effective C++* Item 3)

Use `const` whenever possible.

Structure of the standard library (session 5)

container an object holding a collection of values.

- **sequential** *e.g.* **`string`**, **`vector`**, **`deque`**, **`list`**
- **associative** *e.g.* **`map`** (session 4)

iterator (session 4) an object representing a position within a container, supporting at least **`*`**, **`++`**, **`==`** and **`!=`**. Each standard container has

- associated types **`iterator`** and **`const_iterator`**,
- member functions **`begin()`**, **`end()`**, **`cbegin()`** and **`cend()`**.

inserter (not covered) an object used to insert values into a container at a particular position.

algorithm (session 5) a function operating on the elements of a container indirectly, through iterators and inserters referring to the container.

Standard containers

Sequential containers:

- **string** a sequence of characters
- **vector** provides indexing, growing and shrinking at the back
- **deque** provides indexing, growing and shrinking at both ends
- **list** allows insertion at any point

Associative containers:

- **map** provides indexing by a key type (session 5)

All containers provide `size()`.

Iterators (session 4)

- Iterators support at least `*`, `++`, `==` and `!=`.
- Each standard container defines two iterator types:
  - *container* `::iterator`
  - *container* `::const_iterator`

  The difference is that for `const_iterator`, `*it` is a `const` reference (cannot be modified).
- Each standard container provides member functions:

begin()	`iterator` pointing at the first element
end()	`iterator` pointing just after the last element
cbegin()	`const_iterator` pointing at the first element
cend()	`const_iterator` pointing just after the last element

Standard iterator loop (session 4)

Not changing the elements:

```
for (auto it = l.cbegin(); it != l.cend(); ++it)
    cout << *it << '\n';
```

(`it` has type `list<double>::const_iterator`.)

Changing the elements:

```
for (auto it = l.begin(); it != l.end(); ++it)
    *it += 3;
```

(`it` has type `list<double>::iterator`.)

Range-based `for` loop (session 4)

Not changing the elements (for primitive types):

```
for (auto x : l)
    cout << x << '\n';
```

or (for non-primitive types)

```
for (const auto &x : l)
    cout << x << '\n';
```

Changing the elements:

```
for (auto &x : l)
    x += 3;
```

Standard algorithms

- You will need to be able to use standard algorithms (session 5).
- For any algorithm you are expected to use, you will be given documentation.
- Algorithms operate on iterators, rather than directly on containers.
- The documentation typically refers to [*b, e*), which means a range in a container starting at the element indicated by the iterator *b*, and up to (but not including), the position of the iterator *e*.
- *b* and *e* must point at positions in the same container, and *e* must be reachable from *e*, but this is not checked.
- Most commonly, but not always, *b* is the **`begin()`** or **`cbegin()`** of a container, and *e* is the **`end()`** or **`cend()`** of the same container.

Examples

count(b, e, v) returns the number of elements in the range [*b, e*) that are equal to *v*.

count_if(b, e, p) returns the number of elements in the range [*b, e*) for which *p* is **`true`**.

find(b, e, v) searches the range [*b, e*) and returns an iterator pointing at the first occurrence of *v*, if there is one, or *e* if there is not.

find_if(b, e, p) searches the range [*b, e*) and returns an iterator pointing at the first value for which *p* is **`true`**, if there is one, or *e* if there is not.

sort(b, e) Sort the elements in the range [*b, e*).

sort(b, e, f) Sort the elements in the range [*b, e*) using the function *f* as a less-than function on values.

Functions

- Some of the standard algorithms take a function parameter: **`count_if`**, **`find_if`**, **`remove_if`**, **`sort`**
- This may be an external function:

```
bool small(double x) { return x < 10; }
```

- Alternatively, it may be an anonymous function (lambda expression):

```
    [] (double x) { return x < 10; }
```

- Lambda expressions may also *capture* local variables:

```
    [z] (double x) { return x < z; }
```

Class structure (session 6)

```
class point {
    double _x, _y;
public:
    point(double x, double y) : _x(x), _y(y) {}

    double x() const { return _x; }
    double y() const { return _y; }
};
```

data members (usually private)

constructors specify initialization, often using initializers.

member functions which have access to the data members.

Constructors and member functions may be defined inline or not.
C++ has three access specifiers: **`private`**, **`protected`** and **`public`**.

Inheritance (sessions 7 and 8)

```
class base {
    string field;
public:
    base(args) : field(expr) { ... }
    void non_overridable() { ... }
    virtual void overridable() { ... }
    virtual void abstract() = 0;
};
```

- A member function marked `virtual` maybe be overridden in derived classes.
- The notation `virtual ...=0;` marks a pure virtual member function, and makes the class abstract.

Derived classes

```
class derived : public base {
    int field2;
public:
    derived(args) : base(other_args), field2(expr) { ... }
    virtual void overridable() override { ... }
    virtual void abstract() override { ... }
};
```

- Derived classes cannot initialize members of the base part directly: they must select a base constructor.
- `virtual` is implied and optional when overriding.
- `override` is optional, but invokes a compiler check.

Dynamic binding

Dynamic binding may occur through references (session 8):

```
void foo(base &b) {
    b.abstract();
}
```

or through pointers (session 9):

```
    base *bp = new derived(args);
    bp->abstract();
```

or through smart pointers (session 9):

```
    shared_ptr<base> bp = make_shared<derived>(args);
    bp->abstract();
```

Dynamic memory (session 9)

- Dynamic memory is allocated with **`new`**.
- Storage allocated with **`new`** is not automatically reclaimed: the programmer must free it with **`delete`** to avoid memory leaks.
- But freeing too early (or twice) causes memory corruption and crashes.
- This is another good reason to use locally allocated object where possible.
- If dynamic memory is required, the C++11 standard library provides smart pointers like **`shared_ptr`**, which are often a better solution than **`new`** and **`delete`**.

Collections of pointers (session 9)

To make a collection of various derived classes of a base class, we must use pointers

```
vector<base *> vb;
```

or smart pointers

```
vector<shared_ptr<base>> vb;
```

In either case, we can then loop through the container, calling member functions of **base** on each element, with the implementation selected via dynamic binding:

```
for (const auto &bp : vb)
    bp->abstract();
```

Overview

Assessment in this module:

Coursework (30% of the module)

Exam (70% of the module)

- 120 minutes
- do both of the two questions

To pass the module, you need an overall module mark of at least 40.

If you don't manage that, you will need to retake the component(s) for which you obtained less than 40. Any retake marks will be capped at 40. The marks will then be combined again, and need to be at least 40.