Programming in C++

Session 2 – Classes in C++

Dr Christos Kloukinas

City St George's, UoL

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

(based on slides originally produced by Dr Ross Paterson)

Copyright © 2005 – 2024

C++ source files

A C++ source file may contain:

include directives	`#include <iostream>`
comments	`// what this does`
constant definitions	`const double pi = 3.14159;`
global variables	`int count;`
function definitions	`int foo(int x) { ... }`
class definitions	`class foo_bar { ... };`

Unlike Java, C++ requires that things are declared before use.

Naming – NoMoreCamels!!!

In C++ names of classes, functions, variables, constants, files, *etc.* are all ***lower*** case and multiple words are separated by underscores ("_").
So, never write **`class MyString`** – it should be **`class my_string`** instead.
The exception is things that have been defined in the pre-processor, *e.g.*, **`NULL`** (the old way of naming the null pointer – now it's called **`nullpointer`**).
Pre-processor? What's that?!?! → *(next note page)*

Sidenote – The toolbox

Your source code is treated internally by a sequence of programs:
pre-processor (cpp) → C++ compiler → assembler → linker (ld)

1. The pre-processor (**`cpp`** for C-Pre-Processor). Treats all **`#`**'s.
   It includes files (inserts their contents verbatim at the point where the **`#include`** directive appears, and allows you to define constants and macros that cause changes to your code:
   ```
   #define LOCALHOST "banana.city.ac.uk"
   #define MAX(a, b) (((a)<(b)) ? (b) : (a)) /* many parens but still unsafe - try calling MAX(++i, ++j) */
   ```
   Use flag -E with g++ to ask just for the preprocessor to run.
2. The compiler itself (**`cc1`**) – this one reads text without any **`#include`**'s and compiles to assembly code.
   Use flag -S with g++ to run just up to this point (pre-process & compile only).
3. The assembler (**`as`**). Translates the assembly code into object (*i.e.*, machine) code, producing a file with a suffix **`.o`** (equivalent to a **`.class`** file in Java).
   Use flag -c to run just up to this point.
4. The linker (**`ld`** – Link eDitor). Links all the object files together to produce a standalone executable (somewhat equivalent to when creating a standalone, executable jar file in Java).

Classes in C++

- Like Java, C++ supports
  - classes, with public, protected and private members and methods
  - inheritance and dynamic binding
  - abstract methods and classes

  but the syntax and terminology is different.
- Major semantic difference: ***copying of objects***

  (because now you have direct access to objects)

The elements of a C++ class

```
class date {
```

As in Java, C++ classes contain:

- fields, called *members*

  ```
  int day, month, year;
  ```
- constructors

  ```
  date() ...
  date(int d, int m, int y) ...
  ```
- methods, called *member functions*

  ```
  int get_day() { return day; }
  ...
  ```

```
};
```


Visibility of members and methods

Visibility is indicated by dividing the class into sections introduced by *access specifiers*:

```
class date {
private:
    int day, month, year;

public:
    date() ...
    date(int d, int m, int y) ...
    int get_day() { return day; }
    ...
};
```

In this case, the fields are private, and the constructors and methods are public.

Access specifiers

C++ has the same keywords as in Java, but as there are no packages, the situation is simpler:

private	visible only in this class.
protected	visible in this class and its descendents.
public	visible in all classes.

- Access specifiers may occur in any order, and may be repeated.
- An initial "`private:`" may be omitted.

Constant member functions

Recall that the **`const`** keyword is used for values that cannot be changed once initialized:

```
const int days_per_week = 7;
int last(const vector<int> &v) { ... }
```

We can indicate that the member function **`get_day()`** doesn't change the state of the object by changing its declaration to

```
int get_day() const { return day; }
```

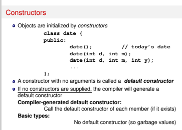
This will be checked by the compiler.
Advice: add **`const`** where appropriate.

Constructors

- Objects are initialized by *constructors*

```
class date {
public:
        date();            // today's date
        date(int d, int m);
        date(int d, int m, int y);
        ...
};
```

- A constructor with no arguments is called a ***default constructor***
- If no constructors are supplied, the compiler will generate a default constructor
  Compiler-generated default constructor:
  Call the default constructor of each member (if it exists)
  Basic types:
  No default constructor (so garbage values)



What do we need a default constructor for?

- There are cases where there are valid default values for an object – then we should offer a default constructor that initialises the object with the default values.
- There are equally cases where there are no good default values – then we should ***not*** offer a default constructor.
- ***It is a design issue – you need to think before programming one.***
- One additional thing you need to think of is whether you'd like to be able to declare ***arrays*** of objects of that class:
  `some_class array[3];`
  When declaring arrays there is no way to pass arguments to the constructor of the array elements – the only constructor that is available to the constructor for initialising the array elements is the default constructor.
  This means that if there is no default constructor then we cannot declare arrays of objects of that class like we've done above.
  Note: Since C++14 we can use array initialisers to bypass this shortcoming:
  `some_class array[3] = { o1, o2, o3 };`
  This way we're initialising the array elements using the ***copy constructor*** [*], copying **`o1`** into **`array[0]`**, **`o2`** into **`array[1]`**, and **`o3`** into **`array[2]`**.
  [*] *Or the move constructor if it exists and it's safe to apply it. . .*

Initialization and assignment of objects

Unlike basic types, objects are always initialized.

```
date today;      // uses default constructor
                 // NOTE: NO PARENTHESES!!!
date christmas(25, 12);
```

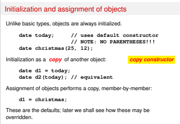
Initialization as a ***copy*** of another object: ***copy constructor***

```
date d1 = today;
date d2(today); // equivalent
```

Assignment of objects performs a copy, member-by-member:

```
d1 = christmas;
```

These are the defaults; later we shall see how these may be overridden.



If we had written **`date today();`** then the compiler would have thought that we want to *declare* (but not define) a FUNCTION called **`today`**, which takes no parameters and returns a **`date`** object. . .

This is the meaning in C and C++ wants to be compatible with C.

Using objects

Declaring object variables:

```
date today;
date christmas(25, 12); // Reminder: book tickets...
```

In C++ (unlike Java) these variables contain objects (not pointers to objects) and they are already initialized.

Methods are invoked with a similar syntax to Java:

```
cout << today.get_day();
christmas.set_year(christmas.get_year() + 1);
```

Except that in C++ **`today`** is an. . . OBJECT.

Qualification in C++ and Java

Java uses dot for all qualification, while C++ has three different syntaxes:

C++	Java
object . *field*	***(no equivalent)***
pointer−>*field*	*Java "reference"* . *field*
Class : : *field*	*Class* . *field*
(no equivalent)	*package* . *Class*

Can't access objects in Java!

Java "ref" = C++ pointer!

Temporary objects

We can also use the constructors to make objects inside expressions:

```
cout << date().get_day();
```

1. A temporary, anonymous **`date`** object is created and initialized using the default constructor;
2. The method **`get_day()`** is called on the temporary object;
3. The result of the method is printed; and
4. The temporary object is discarded (destructor called).

(Can do similarly in Java with **`new`**, but relies on GC.)
Another example:

```
date d;
...
d = date(25, 12);
```

A temporary **`date`** object is created and initialized using the **`date(int,int)`** constructor, copied into **`d`** using the assignment operator, and then discarded (destructor called).

Important

You must be able to describe the ***order of calls*** and be ***precise***:

```
cout << date().get_day();
```

1. A ***temporary `date` object*** is created and initialized ***using the default constructor***;
2. The method `get_day()` is called ***on the temporary object***;
3. The result of the method is printed; and
4. The ***temporary object*** is discarded (destructor called).

```
d = date(25, 12);
```

A temporary `date` object is created and initialized using the `date(int,int)` constructor, copied into `d` using the assignment operator, and then discarded (destructor called).
(Advanced) *Since C++11, the temporary object will be* ***moved*** *into* `d` *using the* ***move assignment operator****, i.e., its contents will be "stolen" by* `d` *(the compiler will consider it as no longer being used), before being discarded (destructor called).*

Initializing members

Members are initialized in initialisation lists, ***NOT*** in constructor bodies!
(it's legal to give default values since C++11)

```
class date {
        int day, month, year;
public:
        date() : day(current_day()),
                month(current_month()),
                year(current_year()) {}

        date(int d, int m, int y) :
                day(d), month(m), year(y) {}
        ...
};
```


Why do we need to initialise members with the constructor initialisation list?
Because all objects need to have been properly constructed before they're used and the members are used by the body of the class's constructor.
If we don't initialise them explicitly at the constructor initialisation list, then the compiler will insert there calls to their default constructors (if these exist. . .)
Try to compile this:

```
class A { public: A(int i){} }; // no default constructor

class B { public: B(int i){} }; // no default constructor

class AB {
    A a;
    B b;
  public:
  AB() { // Implicitly calls A's and B's default constructors
    return ;
  }
};

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

Initializing subobjects

Initializers supply constructor arguments:

```
class event {
        date when;
        string what;
public:
        event(string name) : what(name) {}

        event(string name, int d, int m) :
                what(name), when(d, m) {}
        ...
};
```

If no initializer is supplied, the default constructor is used.

- What happens to `when` at the first constructor?
- When is its constructor called and which constructor is that?

Two ways to define methods

- Methods can be ***defined*** in class definitions

```
int get_day() const { return day; }
```

C++ compilers treat these as *inline* functions

(expand the body where function's called)

- It is also possible to merely ***declare*** a method in a class

```
int get_day() const;
```

Then give the ***full definition outside the class***:

```
int date:: get_day() const { return day; }
```

  - Because this is ***outside the class***, we must qualify the function name with the class name (`date::`)
  - Underlined parts must match the original declaration exactly

The `date` class minus the method definitions

```
class date {
private:
    int day, month, year;

public:
    date();
    date(int d, int m);
    date(int d, int m, int y);

    int get_day() const;
    int get_month() const;
    int get_year() const;
};
```

Note that this falls short of an ideal interface, as all members (even `private` ones) must be included.

The deferred method definitions

At a later point, ***outside of any class***, we can define the methods.
To state which class they belong to, they are qualified with "`date::`".

```
date::date() : day(current_day()),
        month(current_month()),
        year(current_year()) {}

date::date(int d, int m, int y) :
        day(d), month(m), year(y) {}

int date::get_day() const { return day; }
```

Advice: place only the simplest method bodies in the class.

Differences with Java

- Various minor syntactic differences.
- ***In C++ we have variables of object type***:
  - Initialization and assignment involves copying (or *moving* – advanced **[*]**).
  - Pass-by-Value vs Pass-by-Reference

    Use (`const`) references to avoid copying
- Inheritance (session 6):
  - Copying from derived classes involves ***slicing***
  - Method overriding:
    - In Java method overriding is the default;
    - In C++ you have to ask for it (more when discussing ***static vs dynamic binding***).
- (session 5) C++ also has pointers (similar to Java "references")

[*] You can ***copy*** someone's notes or you can ***move*** (*i.e.*, steal) them. . .

Properties (revision)

pre-condition: a condition that the client must meet to call a method.

post-condition: a condition that the method promises to meet, ***if*** the pre-condition was satisfied. (pre → post [*])

invariant: a condition of the state of the class, which each method can depend upon when starting and must preserve before exiting.

- Properties should always be documented.
- Where possible, they should be checked by the program.

[*] $a \to b = \neg a \lor b$ so it's true when a is false, independently of what b is.

Properties are SUPER-important!

- The job of each constructor is to ***establish the class invariant***.
- Each method depends on the ***invariant being true when it's called***;
- And must ***preserve the invariant right before it returns***.
- A method can also have a pre-condition, for example: vector **v** must have at least $k+1$ elements before calling **v[k]**.
- A method can also have a post-condition, for example: vector's **size()** always returns a non-negative integer.

These are your guide to designing correct code.

- If you don't know what your class invariant and method pre/post-conditions are, then your code is ***wrong***.
- It takes practice to come up with good ones (and correct ones). Aim for simplicity!

C-style assertions

- Properties to be checked at runtime can be written using **assert**:

```
#include <cassert>
.
.
.
assert(position < size);
```

- If condition is **false**, program halts, with filename & line number of failed assertion
- Can turn off assertion checking (Stroustrup 24.3.7.2), but don't!
  - Be like NASA: *test what you fly & fly what you test*

users.cs.duke.edu/~carla/mars.html

www.cse.chalmers.se/~risat/Report_MarsPathFinder.pdf

Assertions or Exceptions?

- What should one use – assertions?

```
assert( 1 <= month && month <= 12 );
```

- Or exceptions?

```
if ( !( 1 <= month && month <= 12 ) )
  throw runtime_error("month out of range\n");
```

Exceptions!

- Assertions are enabled during development but are usually meant to be disabled in the release code – exceptions remain in the release code
- Exceptions allow the program to release resources, while assertions don't – so need exceptions to release resources
  - Not only locally – also in the functions that may have called the current one

Next session: Operator overloading

- A kind of polymorphism: overloading resolved with static types
- Any of the C++ operators may be overloaded, and often are
- An overloaded operator may be either an independent function or a member function (where the object is the first argument)
- Example: object I/O, by overloading the **>>** & **<<** operators

Reading for this session:

- Savitch 1, 6.2, 7.1
- (or Stroustrup 2.5.3-4, 2.6, 10.1-6)
- (or Horstmann 8)

- (Plus, [Stroustrup 24.3.7.2] for how to turn assertions off)

Final Notes – I

- What looks like writing to memory (initialisation: `string s = "Hi";` and assignment: `s = s + " there";`) is in fact a **function call** (initialisation: constructor, assignment: assignment operator, *i.e.*, `operator=`).
  - This is because in C++ you access objects directly.
  - So you need to be able to distinguish between initialisation and assignment, as things are not what they look like!
- **Default constructor:** `date()` – no parameters; it initialises the object with default values.
  - The default constructor `date()` will be created by the compiler if you define **NO** constructors at all. This will try to call the default constructors of your class' fields (if they exist – this may cause a compilation error).
    It'll still leave fields of basic types uninitialised. . . `:-(`
    (cause there's no default constructor for basic types. . .)
- **Copy constructor:** `date(const date &o)` – single parameter, which is (a const reference to) another object of the same class.
  It initialises your object as a copy of the other object `o`.
  - The copy constructor will be created by the compiler if you don't define it yourself (even if you've defined other constructors).
    This will try to call the copy constructors of your class' fields.

Final Notes – II

- **Invariant:** What doesn't change.
  Constructors have one goal; to establish the invariant (*i.e.*, make that property true). The methods should then keep it true when they terminate.
- Constant member functions: `int get_day() const { return day; }`
- pre-/post-conditions and invariants:
  - A pre-condition is a property that needs to hold for a method to work correctly, *e.g.*, the deposit amount should be non-negative.
  - We can check it at the start of the method if we want to make sure that we're not being called with wrong values or when the object is not able to offer the services of that method (you don't call a takeaway when they're closed).
    We can throw an exception if it's violated.
    This is called defensive programming (*e.g.*, Java checks that array indices are not out-of-bounds).
  - In some cases, we simply document it and don't check for it – it's the caller's obligation to ensure it's true (and they may get garbage or crash the program if it isn't – C++ doesn't check array indices, it's your problem!).
- A post-condition is a property that a method promises to the caller after it has completed executing, as long as the pre-condition was true when it started executing.
  Otherwise the method promises nothing – all bets are off.
  - We can check for it right before returning, *e.g.*, the deposit method can check that the new balance is equal to the old balance plus the deposit amount.
    We can throw an exception if it's violated or try to repair the error.
  - Sometimes we document it because it's too expensive to check, *e.g.*, checking if we've indeed sorted an array can take a lot of time, so may want to only do it during testing, not in normal operation.
- An invariant is a property that never changes ("in-variant").
  It should hold immediately after the constructors (that's their main goal!!!), and hold immediately after any non-const member function, *e.g.*, the balance should always be non-negative.
  - It's difficult to identify invariants (and to get them right) but it's them that actually help us to design correct and robust code.
    We usually start by observing what the different constructors try to achieve – that gives us a glimpse into how the invariant might look like.
  - We can then look at the code of each method to see if they preserve the invariant, *i.e.*, if the invariant was true before the method, will it be true after it as well?
- When thinking of pre-/post-conditions and invariants, and when doing code testing we need to think of all possible values – not just the ones we like.
  If we receive numbers as input, always check for -1, 0, 1.
  Just because you call a parameter amount, it doesn't mean that it's a positive number – it could be anything.