

Programming in C++

Session 1 – Introduction

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)



CITY
ST GEORGE'S
UNIVERSITY OF LONDON

Copyright © 2005 – 2024

What's this module about?

Goal Become a novice C++ programmer.

- That's actually advanced!
- Hard for novice programmers.
- C++ is hard
 - Multiple programming styles (procedural, OO, generic programming)
 - Language & compilers geared towards experienced programmers
 - Function calls are often hidden
 - Compiler messages can seem cryptic
 - Different standards: 1998, 2011 (major changes!), 2020, 2023
- Please ask questions!!! (lecture/Moodle)

This module: more OO programming, in C++

Assuming that you are a reasonably skillful **Java/C#**/etc. programmer, by the end of this course you should be able to

- read and modify substantial well-written C++ programs
- create classes and small programs in C++ that are:
 - Correct
 - Robust
 - Clear
 - Reusable
- use various object-oriented features, including genericity, inheritance and multiple inheritance

A bit of language history

1960 Algol 60: block structure, static typing

1967 Simula: Algol plus object-orientation (for simulation)

1970 C: statically typed procedural language with low-level features

1972 Smalltalk: object-orientation (for graphical interfaces), no static types

1985 C++: **C** + Object-Oriented features and (later) genericity

1995 **Java**: "**C++ greatly simplified**"

Procedural Algol 60, C, ...

"To dress a young child you do X, Y, Z"

Object-Oriented Simula, Smalltalk, C++, Java, ...

"To dress a grown up, you ask them to dress themselves"

A bit of language history — Part II

1972 C Procedural, static typing, low-level access

1985 C++ Your beloved (top) language C **extended!**

- C++ compilers **can** compile C programs
(The Linux kernel is compiled in this way)

C++ “C is good”

1995 Java Your beloved (top) language C++ **simplified!**

- Java compilers **cannot** compile C++ programs

Java “C++ is **too** complex”

The differences between C++ & Java are serious pain points

One needs to understand them to understand the C++ language
(expert knowledge of Java not really required for this)

C++ design criteria

Started as “C with Classes”

- support a variety of programming styles, including object oriented (give the programmer more choices)
- powerful (give the programmer more control)
- enable efficient implementation (shift some implementation concerns to the programmer)
- extension of C (machine-level access)
Often C features coexist with newer, cleaner versions.
And C++98 features coexist with C++11 & C++20 versions. . .

Java design criteria

Keep things as simple as possible

- object orientation
- (moderate) simplicity (fewer variant ways of doing things)
- robustness and security (type-safe, automatic memory allocation)
- architecture-neutral (fairly high level)
- syntax based on C++

This session: non-OO programming in C++

This session introduces the philosophy of C++, and some simple non-OO programs.

We will touch on the following features of C++:

- Operator overloading
- Constants
- Initialization vs. assignment **
- Parameter passing by value and reference **
- Some library classes

All will be explored in greater detail later.

** NOT like Java!

The toolset

To	Java	C++
Compile (notes)	<code>javac -g pkg1/pkg2/.../pkgN/X.java</code> <code>-g</code> debug on	<code>g++ -g -c x.cpp</code> <code>-c</code> compile only
Link/etc	<code>jar cfe prog.jar X X.class</code>	<code>g++ -g -o prog x.o</code>
or (notes)	<code>echo Main-Class: X > manifest.txt</code> <code>jar cfm prog.jar manifest.txt X.class</code> e executable ("main" is in class X)	<code>-o</code> output to
Execute	<code>java -jar prog.jar</code>	<code>./prog</code>
Debug	<code>jdb -classpath prog.jar X</code> <code>stop in X.main</code> <code>run a1 a2 a3</code> <code>print 3+4</code> <code>print args</code> <code>step</code>	<code>gdb prog</code> <code>break main</code> <code>run a1 a2 a3</code> <code>print 3+4</code> <code>print argv[0]</code> <code>step</code>
Curious	<code>javap -c X</code>	<code>nm -C x.o</code> <code>nm x.o c++filt</code>

A C++ program is processed by the preprocessor (**cpp**), the compiler (**g++**), and the linker (**ld**) – all of these can complain.

A small C++ program (vs in Java)

<pre> /* C++: */ #include <iostream> using namespace std; int main(int argc , char *argv[]) { cout << "Hello world!\n"; return 0; } </pre>	<pre> /* Java: */ class MyProg { public static void main(String[] args) { System.out .print("Hello world!\n"); } } </pre>
---	--

- The first two lines make available names from the standard library, like `cout`. *C++ pretends to not know the standard types...*
- In C++ (like C), a function (**main**) can exist outside of any class.
 - Java: oh, that's a (public) **static** method! *Q: Called on what?*
- Style: C++ – **lower_case**, Java – **CamelCase**
 - Q: Where's the **print** function call in C++?*
 - Q: Java: Where do **args** come from?*

Accessing names from standard libraries

- In Java, classes are collected in packages, and accessed with **import** declarations.
- In C++, there are two (mostly) independent ways of controlling access to names:
 - header files** like **iostream** contain collections of related definitions (in this case for I/O streams). A typical program will begin with several **#include** lines.
 - namespaces** like **std** are collections of names, which must usually be qualified (**std::cout**), unless there is a **using** command. Each source file will include the above **using** line, but we will not make any other use of namespaces.

Text output

```
cout << "Hello world!\n";
```

- The **iostream** header defines three standard streams:
 - cin** standard input (cf. Java's **System.in**)
 - cout** standard output (cf. Java's **System.out**)
 - cerr** error output (cf. Java's **System.err**)
- Applied to integers, `<<` performs a left shift (as in Java)
- Applied to an output stream and a string, writes the string to the stream
- The `<<` operator is *overloaded*

Programming in C++

Differences

Text output

Text output

```
cout << "Hello world!\n";
```

- The `iostream` header defines three standard streams:
 - `cout` standard output (or `ios::out`)
 - `cerr` standard error output (or `ios::cerr`)
 - `clog` standard error output (or `ios::clog`)
- Applied to integers, `<<` performs a left shift (as in Java)
- Applied to an output stream and a string, writes the string to the stream
- The `<<` operator is overloaded

- Why do we need both `cout` and `cerr`?
 - We need both so that we can separate the output from the errors into different files (or sockets), e.g., when using the bash command shell:


```
program > output.txt 2> errors.txt
```
- What's the difference between `cout` and `cerr`? Why would one want to use both if not splitting the output as above?
 - We need both because they behave differently.
 - When printing to `cout`, our output is *buffered*, i.e., it is placed into a temporary area and stays there until the output buffer has been filled. When the buffer is full, the output is sent out to wherever it is supposed to be sent (terminal, file, network).
 - Unlike `cout`, when printing to `cerr` the output is not buffered – it is printed immediately.
 - This is why when printing to `cout` we sometimes have to use `flush` to tell the buffer to output whatever it has stored, even if it is not full:


```
cout << "Hi"; cout.flush();
```

 Or alternatively:


```
cout << "Hi" << flush;
```

Programming in C++

Differences

Text output

Text output

```
cout << "Hello world!\n";
```

- The `iostream` header defines three standard streams:
 - `cout` standard output (or `ios::out`)
 - `cerr` standard error output (or `ios::cerr`)
 - `clog` standard error output (or `ios::clog`)
- Applied to integers, `<<` performs a left shift (as in Java)
- Applied to an output stream and a string, writes the string to the stream
- The `<<` operator is overloaded

Flushing streams – `endl`

- Another way to flush the output stream is to use `endl`. We've seen so far how to use the special character '`\n`' to insert a newline character into the output. With `endl` we can insert a newline and at the same time flush the output stream:

```
cout << "Hello, how are you?\n" // no printing yet
    << "How could I be of assistance?"
    << endl; // Add a new line & flush everything
```

Differences

Input and output

```
int i;
cout << "Type a number: " << flush;
cin >> i;
cout << i << " times 3 is " << (i*3) << '\n';
```

- The `>>` operator reads from an input stream.
- The `<<` operator associates to the left, and returns the stream; the above is equivalent to


```
((cout << i) << " times 3 is ") << (i*3) << '\n';
```
- It is also overloaded for `int` (`i`, `i*3`) and `char` (`'\n'`).
- The `>>` operator is similar.

Programming in C++

Differences

Input and output

Input and output

```
int i;
cout << "Type a number: " << flush;
cin >> i;
cout << i << " times 3 is " << (i*3) << '\n';
```

- The `>>` operator reads from an input stream.
- The `<<` operator associates to the left, and returns the stream; the above is equivalent to


```
((cout << i) << " times 3 is ") << (i*3) << '\n';
```
- It's also overloaded for `int` (`i`, `i*3`) and `char` (`'\n'`).
- The `>>` operator is similar.

```
cout << i << " times 3 is " << (i*3) << '\n'; // same
((cout << i) << " times 3 is ") << (i*3) << '\n';
```

In order for this to work, the `operator<<` has to return an output stream. That's why when `(cout << i)` is computed we can use its result (the modified `cout` (`cout'`)) to apply the next `operator<<` with the next argument (`" times 3 is "`).

So:

```
((cout << i) << " times 3 is ") << (i*3) << '\n';
cout' << " times 3 is "
    cout'' << (i*3)
    cout''' << '\n';
```

```
#include <string>
```

The standard library provides a `string` type:

```
string s = "fred";
cout << s;
cin >> s;    // reads a word
```

The `+` operator is overloaded on strings:

```
s = s + " and bill";
s = s + ', ';
```

So are `+=`, `==`, `<`, etc.

Unlike in Java, strings are modifiable:

```
s.erase();    // now s == ""
```

```
#include <string>
#include <iostream>
using namespace std;

int main() {
    string s;
    while (cin >> s)
        cout << s << '\n';
    return 0;
}
```

- The `>>` operator on strings reads words.
- The stream returned by the `>>` operator can be used in a conditional, to test if the read was successful.***

(what do these words mean?)

2024-11-13

Programming in C++

Differences

Breaking the input into words

```
while (cin >> s)
    "The stream returned by the >> operator can be used in a conditional, to test if the read was successful."
    The expression cin >> s returns the modified input stream cin, which is what we ask while to evaluate so as to decide whether the loop body should be executed or not.
    The C++ library has functions that allow one to translate an input stream into a boolean – the boolean is true if the last attempt to read from the stream succeeded, and it's false otherwise (e.g., the input had finished, the input is corrupted, etc.). These functions work like when we write s1 = s2 + " Hi " + 3; in Java – there they translate automatically the array of characters " Hi " and the integer 3 into string objects, that they concatenate with the string object referenced by s2 to obtain the value of the string object that will be referenced by s1 (s1 and s2 are not objects in Java, they are pointing to objects.).
    The meaning of while (cin >> s) is:
    "Try to read a word from cin into string object s and if that has succeeded, then continue executing the body of the while loop."
```

Breaking the input into words

```
#include <string>
#include <iostream>
using namespace std;

int main() {
    string s;
    while (cin >> s)
        cout << s << '\n';
    return 0;
}
```

• The `>>` operator on strings reads words.
 • The stream returned by the `>>` operator can be used in a conditional, to test if the read was successful.
(what do these words mean?)

```
#include <vector>
```

C++ has arrays, but we'll use vectors instead (*cf.* Java's `ArrayList`):

```
vector<int> vi(5);    // vector of 5 ints
vector<string> si;    // empty vector of strings
```

Vectors can be accessed just like arrays:

```
vi[1] = x;            // vi.set(1, x);    <3 Java! :-P
vi[2] = vi[1] + 3;    // vi.set(2, vi.get(1) + 3); <3 <3
```

Vectors can also be extended:

```
si.push_back(s);
```

The current length of `si` is `si.size()`

2024-11-13

Programming in C++
└─Differences

└─Vectors

Vectors

```
#include <vector>
// has things but not 3 same vectors instead of java's ArrayList
vector<int> v1(5); // creates v1 of 5 ints
vector<int> v2; // empty vector of ints
// can be accessed just like array
v1[1] = 4; // v1[0] is 0, v1[1] is 4, v1[2] is 0, v1[3] is 0, v1[4] is 0
v2[1] = v1[1] + 5; // v2[0] is 0, v2[1] is 9, v2[2] is 0, v2[3] is 0, v2[4] is 0
// can also be accessed
v2.push_back(4);
// current length of v2 is 5, v2.size()
// current length of v1 is 5, v1.size()
```

Syntax seems simple but the meaning is not...

Expression “`vi[1]`” in Java would have to be written as “`vi.get(1)`”, where `vi` would have been declared instead as a Java pointer to an `ArrayList` container.

- Thanks to operator overloading C++ allows us to type less (2 characters for “`[]`” instead of 6 characters for “`.get()`”).
- It also allows us to keep the syntax of arrays that we’re familiar with and treat vectors as if they’re advanced arrays (that we can extend/shorten).
- But this comes at a price – the code is not as clear now as it was in Java. In Java it’s obvious we’re calling a function while in C++ it is not so obvious – one has to remember that **every** use of an operator is actually a function call in C++!
- So `vi[1]` is actually `vi.operator[](1)`.

Differences

Language notes

- `string` is a class
- `vector` is a template (generic) class
- C++ has pointers (like in Java), but we won't use them till later:

```
string s1 = "bill", s2;
```

declares (and initializes) string objects, not pointers
- assignments like

```
s1 = s2;
```

copy the objects (not the Java pointers!)

Note: syntax looks like Java, but meaning is VERY different

Capitalisation: In C++ everything is lower case – words are separated by underscores: `class string, void push_back`

Dr Christos Kloukinas (City St George's, UoL)

Programming in C++

17/31

Differences

Initialization vs. assignment

Initialization of variables:

```
string s1;
string s2 = "bill";
```

Objects are always initialized; variables of primitive type aren't. Assignment replaces an existing value:

```
s1 = s2;
```

Initialization defines a new variable:

```
string s3 = s2;
```

Slide has 4 different method calls!
(C++ function calls are often hidden!)

2024-11-13

Programming in C++
└─Differences

└─Initialization vs. assignment

Initialization vs. assignment

```
Initialization of variables:
string s1;
string s2 = "bill";
// Objects are always initialized; variables of primitive type aren't.
// Assignment replaces an existing value.
s1 = s2;
// Initialization defines a new variable.
string s3 = s2;
```

Note: look at difference between code!
(Even function calls with other languages)

SUPER IMPORTANT!!! – I

This slides looks simple and boring – initialise some variables, assign some variables, blah blah blah, whatever... Your success in the module depends on understanding it fully – and it ain't easy. It actually shows **four different methods**. Remember that `s1`, `s2`, and `s3` are real objects in C++ – unlike Java where they are *pointers*.

```
string s1;
/* INITIALISATION: To initialise s1, the string
constructor must be called.
Which constructor? The one taking no arguments.
So here, we call:
string()
```

SPECIAL NAME: ``Default Constructor`` */

2024-11-13

Programming in C++
└ Differences
└ Initialization vs. assignment

Initialization vs. assignment
Initialization of variable:
string s1;
string s2 = "hello";
Objects are always initialized; variables of primitive type aren't.
Assignment replaces an existing value:
s1 = s2;
Initialization defines a new variable:
string s3 = s2;
(Both kinds of statements are valid C++ code.)

SUPER IMPORTANT!!! – II

```
string s2 = "bill";  
/* INITIALISATION: Which constructor do we call to  
initialise s2?  
The one taking an array of characters:  
string( const char a[] ) */  
  
s1 = s2;  
/* ASSIGNMENT: s1 and s2 are OBJECTS, not just  
pointers to objects (as in Java).  
  
So here we're calling a FUNCTION:  
string & operator=(string &o, const string &o);  
Though usually we're calling a METHOD:  
string & operator=(const string &o);  
SPECIAL NAME: ``Assignment Operator`` */  
  
string s3 = s2;  
/* INITIALISATION: Which constructor do we call  
to initialise s3?  
The one taking another object of class string:  
string( const string &o )  
SPECIAL NAME: ``Copy Constructor`` */
```

2024-11-13

Programming in C++
└ Differences
└ Initialization vs. assignment

Initialization vs. assignment
Initialization of variable:
string s1;
string s2 = "hello";
Objects are always initialized; variables of primitive type aren't.
Assignment replaces an existing value:
s1 = s2;
Initialization defines a new variable:
string s3 = s2;
(Both kinds of statements are valid C++ code.)

Is it initialisation or assignment?

- To distinguish between initialisation and assignment you need to look at the form of the statement.
- If it's initialisation we are just introducing a new variable, so we have to tell the compiler what is its type.
string s1;
string s2 = "Bill";
string s3 = s2;
All initialisations of objects call a constructor of the object's class.
- When assigning a variable the variable exists already, so we do not declare its type:
s1 = s2;
Assignments call the assignment operator: operator=

Differences

The BIG Difference

Java	C++
String s; // s == null // s is a Java *POINTER*!!! // nothing called	string s; // s != null // s is an OBJECT // constructor called!

- You can never access an object directly in Java (for *safety*).
- C++ gives you direct access to objects (for *performance/control*).

Many of their core differences are a consequence of this!

- Garbage collection **vs** Manual memory deallocation
- Sharing objects by copying Java pointers **vs** Copying objects
- Immutable strings **vs** Modifiable strings
- Call by value **vs** Call by reference

2024-11-13

Programming in C++
└ Differences
└ The BIG Difference

The BIG Difference
Java
String s;
// s == null
// s is a Java *POINTER*!!!
// nothing called
C++
string s;
// s != null
// s is an OBJECT
// constructor called!
• C++ gives you direct access to objects (for performance/control).
Many of their core differences are a consequence of this!
• Garbage collection vs Manual memory deallocation
• Sharing objects by copying Java pointers vs Copying objects
• Immutable strings vs Modifiable strings
• Call by value vs Call by reference

DANGER!!!

If you don't understand what the big difference is here, you're in dangerous waters.

- Draw a picture of the memory for Java and another for C++.
- Draw the objects in each – there is one for Java and one for C++.
- The C++ object is called **s** – that's all there is in the memory of C++.
- The Java object has NO NAME. In Java, the name **s** is the name of an object POINTER [*], and this (Java) POINTER is in another location in memory and is pointing to the actual Java object.

Confused? Go over this again (and again, and again, ...) till you have understood it – it's super-basic and you'll suffer if you don't get it.

[*] Java's "references" are **pointers** – that's why when you try to use a NULL Java "reference" you get a "NullPointerException". You do not get a "NullReferenceException", do you?

Passing parameters by value

Formal parameters are new variables, initialized from the actual parameters (a.k.a. arguments)

```
void f(int i) {
    i = i + 5;
}

void g() {
    int j = 3;
    f(j);    // no effect on j
    f(j*2);  // acceptable
}
```

Passing parameters by value

```
void f(int i) {
    i = i + 5;
}

void g() {
    int j = 3;
    f(j);    // no effect on j
    f(j*2);  // acceptable
}
```

Pass by value

- `void f(int i)` – here `i` is a **local** variable of function `f`, which gets initialised with whatever we pass as argument to the function.
- That's why we can call the function with an expression as an argument:
`f(3 * 2);`
 Parameter `i` will be initialised with the value of that expression
`int i = 3*2; /* 6 */`

Parameter passing in Java

- In Java/C, all parameters are passed by value
 Even Java/C pointers!

By Value the method is given **a copy** of the parameter

- Any changes have no effect on the original
- If param is a pointer, the copy *points* to the same object
 - So it is possible to modify the object *pointed* to

But Cannot modify original pointer to point to another object

Limitations of value parameters

- Might wish to change an actual parameter inside a function
- Parameter might be large (e.g. an object), so expensive to copy
- A solution (Fortran, Pascal, C++, etc) is reference parameters
- Can get similar effects with value parameters & pointers
 (but more error-prone)
 (so we prefer references)

Passing parameters by reference

A *reference* parameter is another name (an alias) for the actual parameter

```
void f(int &i) {
    i = i + 5;
}

void g() {
    int j = 3;
    f(j);    // j is updated
    // f(j*2);    // NOT ALLOWED!
}
```

Note: There is no relationship to Java's pointers ("references").

Less error prone: Reference params can never be NULL!

Passing large values by reference

Reference parameters are also used to avoid copying large values:

```
int last(vector<int> &v) { // v exists!
    return v[v.size() - 1];
}

void g() {
    vector<int> x(100);
    ...
    int n = last(x);    // don't copy x
}
```

Constant parameters: **const** <3 <3 <3

We can indicate that the function doesn't change the parameter with the keyword **const**:

```
int last(const vector<int> &v) {
    return v[v.size() - 1];
}

void g() {
    vector<int> x(100);
    ...
    int n = last(x);    // don't copy x
}
```

This makes programs **safer**, and **helps** the compiler.

Constants

- The C++ keyword **const** introduces a *constant*:

```
const int days_per_week = 7;
```
- Constants may (must!) be initialized, but cannot be assigned to.
- const** parameters are a special case.
- C programmers: use **const** instead of **#define**, or use **enum** definitions:

```
enum class traffic_light { red, yellow, green };
traffic_light r = traffic_light::red;

enum class colour_rgb { red, green, blue };
colour_rgb r = colour_rgb::red;
```
- A different use of **const** will be mentioned later.

Use **const** wherever you can!

Programming in C++

└ Differences

└ Constants

Constants

- The C++ keyword `const` introduces a constant.
- `const int days_per_week = 7;`
- Constants may (must) be initialized, but cannot be assigned to.
- A constant parameter is a special case.
- C programmers use `const` instead of `final` (C++), or use `enum` definitions.
- `enum class traffic_light { red, yellow, green };`
- `traffic_light x = traffic_light::red;`
- `enum class outdoor_light { on, green, blue };`
- `outdoor_light y = outdoor_light::red;`
- A different set of constants will be mentioned later.

The constant reference rule

We should always try to use `const` wherever we can and only remove it if the compiler complains that we cannot update something because it is `const` (and we cannot figure another way to do what we want without updating).

Consts improve our code — make it more robust and help the compiler optimise further.

Other ways to restrict the code and help the compiler is to use the more restrictive versions of things, e.g., (lecture 7) prefer `unique_ptr<T>` over `shared_ptr<T>`, if possible.

John Carmack (founder and technical director of Id Software) had written a blog post (back in 2013) about this — read it here:

<https://web.archive.org/web/20130819160454/http://www.altdevblogaday.com/2012/04/26/functional-programming-in-c/>

In his Quakecon 2013 keynote he also talked about it (among other things) — this is the relevant part:

https://www.youtube.com/watch?v=1PhArSujR_A

Differences

References

- The C++ symbol `&` after a type defines a **reference**, which is another name (or alias) for a piece of **storage (a.k.a. lhs)**
- Initialization defines the reference as an alias:

```
int x;
int &y = x; // there's only one int here
```

```
person dr_jekyll;
person &mr_hyde = dr_jekyll; // only one person
```

- Assignment assigns to the original storage:

```
y = 3;
```

is the same as assigning to `x`.

References can never be NULL!

Programming in C++

└ Differences

└ References

References

- The C++ symbol `&` after a type defines a **reference**, which is another name (or alias) for a piece of **storage (a.k.a. lhs)**
- Initialization defines the reference as an alias:
- `int x;`
- `int &y = x; // there's only one int here`
- `person dr_jekyll;`
- `person &mr_hyde = dr_jekyll; // only one person`
- Assignment assigns to the original storage:
- `y = 3;`
- is the same as assigning to `x`.

References can never be NULL!

- C++ references are *almost* like (const) pointers:
 - A reference can never be `NULL` - it must always refer to a legitimate object;
 - Once established, a reference can never be changed so that it refers to a different object - a `const` pointer;
 - A reference does not require any explicit mechanism to de-reference the memory address & access data values (it's just an alias).

- C++ references are NOT pointers.

- Never state in public or write down that they are pointers.
- Never say that they "point" to an object or say that they "have its address".

All of these demonstrate a gross misunderstanding of what a C++ reference is.

A C++ reference IS the thing it refers to. They are one and the same.

- Why use references inside a block of code? To simplify things:

```
int &size = tree.left.value.size;
++size;
cout << size;
equivalent to:
++(tree.left.value.size);
cout << tree.left.value.size;
```

Examples

An example function (from `iostream`)

```
istream &getline(istream &in, string &s) {
    s.erase();
    char c;
    while (in.get(c) && c != '\n')
        s += c;
    return in;
}
```

// Use:

```
//string s;while ( getline(cin, s) ){cout<<s<<endl;}
```

Note that

- `get` also uses pass-by-reference
- There's no copying here: arg `in` returned by reference (Cannot return a local by reference)

(never use `getline` unless explicitly told to)

2024-11-13

Programming in C++
└─ Examples

└─ An example function (from `istream`)

An example function (from `istream`)

```
istream &getline(istream &in, string &s) {
    s.clear();
    while (in.get() != '\n')
        s += in.get();
    return in;
}
// Note
// string s, while ( getline(in, s) ){ cout<<endl; }
// Note that
// get also uses pass-by-reference
// There is no copy of data, arg is returned by reference
// (Cannot return a local by reference)
```

- How many things does `getline` return? Three – the result, the modified parameter `in` and the modified parameter `s`.
By using reference parameters you can return multiple things.
- Parameter `in` is passed by reference, because we need to modify the input stream (we modify it when we call `in.get(c)` since we remove one character from it).
- Parameter `s` is passed by reference because we need again to modify the string so as to be able to return to our caller the contents of the line we've read from the input.
- We cannot simply return a `string` from the function, because we need to return a stream – and we need that because we want to use `getline` as in the next slide, where we test the returned stream to see if `getline` succeeded in reading a line or not.
- Note that the returned result (`istream &`) is also returned by reference to avoid returning a copy of `in`!
- In order to return a variable by reference, the variable must not be local – it must have been received as a reference parameter.
 - This is because all local variables are destroyed when a function returns so they no longer exist to be returned themselves – only a copy of them can be returned.

2024-11-13

Programming in C++
└─ Examples

└─ An example function (from `istream`)

An example function (from `istream`)

```
istream &getline(istream &in, string &s) {
    s.clear();
    while (in.get() != '\n')
        s += in.get();
    return in;
}
// Note
// string s, while ( getline(in, s) ){ cout<<endl; }
// Note that
// get also uses pass-by-reference
// There is no copy of data, arg is returned by reference
// (Cannot return a local by reference)
```

- (Advanced)
- Since C++11, one can return an object without copying it. These versions of the C++ language standard support *moving* objects.
- If your class contains sub-objects of classes that are well-behaved (`string`, `vector<T>`, etc.) then objects of your class can be moved without you having to do anything special.
 - Just pass flag `-std=c++23` to the compiler (this flag works for the `g++` and `clang++` compilers).

Examples

Prefixing lines with their lengths

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string s;
    while (getline(cin, s))
        cout << s.size() << '\t' << s << '\n';
    return 0;
}
```

Don't Panic!



- C++ Classes: very similar to Java, but with important differences.
- Reading:
 - *Absolute C++* by Walter Savitch, Addison-Wesley Longman, Reading, Mass, 2002. Chapter 1, sections 6.2 and 7.1.
 - *The C++ Programming Language* (3rd edition) by Bjarne Stroustrup, Addison-Wesley Longman.
 - For this session: sections 2.1–3 (except 2.3.3), 3.2–6 (except 3.5.1), 3.7.1.
 - For next session: sections 2.5.3–4, 2.6, 10.2.1–6.

Final Notes

- Make sure you understand the difference between initialisation (**TYPE VARNAME = EXPRESSION;)** and assignment (**VARNAME = EXPRESSION;)**.
In C++ these call different methods – you need to know which case it is to figure out which method will be called (and to understand how to write these methods – more later).
- **BIG DIFFERENCE** between Java and C++ – in C++ you have direct access to objects, in Java you can only access **pointers** to objects.
- Because of the direct access to objects, C++ supports **call-by-reference** as well as **call-by-value** – make sure you understand the differences! (and call-by-constant-reference. . .)
(and return-by-reference vs return-by-value. . .)

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

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

Programming in C++

C++ source files

C++ source files may contain:

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

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 `nullptr`).

Pre-processor? What's that?!?

→ (next note page)

Programming in C++

C++ source files

C++ source files may contain:

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

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

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 classesbut 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)

```
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
 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.

2024-11-13

Programming in C++

Initialization and assignment of objects

Initialization and assignment of objects

Little basic types, objects are always initialized

```
date today; // some default constructor
date christmas(25, 12); // some: NO parameters!!!
```

Initialization as a copy of another object

```
date d1 = today;
date d2(d1); // 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                     |
|--------------------------|--------------------------|
| <i>object . field</i>    | (no equivalent)          |
| <i>pointer-&gt;field</i> | Java "reference" . field |
| <i>Class :: field</i>    | 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).



## └ Temporary objects

## 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).*

## Temporary objects

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

```
void v1 = 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 anything in Java with new, but rules on C++)

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).

## 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) {}

 ...
};
```

## └ Initializing members

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 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) {}

 ...
};
```

## 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. ( $\text{pre} \rightarrow \text{post} \text{ [*]}$ )

**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.

$[*] a \rightarrow b = \neg a \vee b$  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  $\mathbf{v}$  must have at least  $k + 1$  elements before calling  $\mathbf{v}[\mathbf{k}]$ .
- 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](https://users.cs.duke.edu/~carla/mars.html)

[www.cse.chalmers.se/~risat/Report\\_MarsPathFinder.pdf](https://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)

## Programming in C++

2024-11-13

Next session: Operator overloading

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 – 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.

## Programming in C++

2024-11-13

Next session: Operator overloading

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.

## Programming in C++

### Session 3 – Overloading

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

## Polymorphism

Code that works for many types.

**ad-hoc polymorphism** (overloading) – this session

The version executed is determined statically from the types of the arguments (Savitch 8.1; Stroustrup 7.4,11; Horstmann 13.4)

**parametric polymorphism** (genericity) – next session

A single version, parameterized by types, is used (Savitch 16.1–2; Stroustrup 13.2–3; Horstmann 13.5)

**subtype polymorphism** (dynamic binding) - session 7

The version executed is determined dynamically. (Savitch 14,15; Stroustrup 12; Horstmann 14)

## Overloading

Term symbol is overloaded. . .

A single symbol has multiple meanings.

The meaning of a particular use is statically determined by the types of its arguments.

The following may be overloaded in C++:

- constructors (as in Java) – often useful.
- member functions (or methods, as in Java) – a dubious (and dangerous) feature.
- independent functions – ditto.
- operators – heavily used in the standard libraries.  
Operator overloading makes for concise programs, but overuse may impair readability.

## Implicit conversions and overloading

- Recall that numeric types may be implicitly converted, e.g. given a definition

```
void f(double x) { ... }
```

it is legal to write `f(1)`, because `1`, of type `int` can be implicitly converted to `double`. (Later: similar situation with inheritance of class types.)

- Now suppose there was another definition

```
void f(int n) { ... }
```

If we call `f(1)`, the best (most specific) definition is selected, i.e. the one closest to the call type.

So, to be explicit - say which you really want: `f(1.0)`  
or even better `f( double(1) )`

## Ambiguity

Given the definitions

```
void f(int i, double y) { ... }
void f(double x, int j) { ... }
```

the following is rejected by the the compiler:

```
f(1, 2); // ambiguous!
```

We could get around this by also defining

```
void f(int i, int j) { ... }
```

Then every application would have a best match.

## Programming in C++

2024-11-13

### Ambiguity

#### Ambiguity

Given the definitions  
void f(int i, double y) { ... }  
void f(double x, int j) { ... }  
the following is rejected by the compiler:  
f(1, 2); // ambiguous!  
We could get around this by also defining  
void f(int i, int j) { ... }  
Then every application would have a best match.

You're writing programs for PEOPLE first!

So, DOCUMENT THEM!

```
f(int(1), double(2));
```

## Overloading – Write fewer if's with OOP!

Overriding – compare:

```
void move(person p) {
if (p isA driver) {
} else if (p isA cyclist) {
} else if (p isA pilot) {
} else { /*DEFAULT* }
```

```
class person { /*DEFAULT*
void move() { ... } }
class driver : person {
void move() { ... } }
class cyclist : person {
void move() { ... } }
class pilot : person {
void move() { ... } }
```

Overloading – compare:

```
void f(x) {
if (x isA double) {
} else if (x isA float) {
} else if (x isA int) {
} else { assert(0); } /*ERROR*
void f(double x) { ... }
void f(float x) { ... }
void f(int x) { ... }
/*NO* (runtime) *ERROR*!!!
```

They allow us to write if/then/else's better – the compiler does it!

## Overloaded equality

In C++, we can compare values of built-in types:

```
int i;
if (i == 3) ... // [*]
```

We can also compare objects:

```
string s1, s2;
if (s1 == s2) ...
```

And similarly for **vectors**.

The == operator is **overloaded**:

special definitions have been given for **string**, **vector** and many other types.

[\*] Prefer **(3 == i)**, because “if (i = 3)” is valid C++ (and it's always true...)

## Expanding overloaded operators

An operator can be either an independent function or a member function, in each case with a special name starting with `operator`:

**Binary operators** An expression `a == b` could mean either of

- `operator==(a, b)` (independent function)
- `a.operator==(b)` (member function)

**a is the implicit 1<sup>st</sup> argument!**

**Unary operators** An expression `! a` could mean either of

- `operator!(a)` (independent function)
- `a.operator!()` (member function)

As with ordinary overloading, there must be a unique best match.

**a.method(b, c, d) is in reality: method(a, b, c, d)**

## Comparing points

```
class point {
 int _x, _y;
public:
 point(int x, int y) : _x(x), _y(y) { }
 int x() const {return _x;} // p1.x(); p1 as if const
 int y() const {return _y;}
 // p1 == p2; stands for p1.operator==(p2);
 bool operator==(const point &p) const {
 return _x == p._x && _y == p._y();
 } // methods can read private fields
};
```

- Use `const` as much as possible.
- Put it in by default, only remove it if you (*really*) need to.
- If you need a non-`const` version, see if you can also provide a `const` one (for use with constant objects).

## An alternative definition

We could instead have defined an independent function:

```
// p1 == p2; now stands for operator==(p1, p2);
bool operator==(const point &p1, const point &p2){
 return p1.x() == p2.x() && p1.y() == p2.y();
}
```

In either case we can then write

```
point p1, p2;
...
if (p1 == p2)
 ...
if (p1 == point(0, 0)) // temporary object
 // (only works if second parameter is *const*)
...

```

## A note on types

- The language does not enforce any constraints on the argument types and return type of `operator==`, or any other operator.
- It is conventional that the arguments have the same type and the result type is `bool`.
- It is also conventional that the `==` operator should define an equivalence relation.
- Departing from these conventions is permitted by the language, but will be very confusing for anyone trying to understand your code (including a future you).

Equivalence Relation  $R$ :

**Reflective**  $x R x$

**Symmetric**  $x R y \rightarrow y R x$

**Transitive**  $x R y \wedge y R z \rightarrow x R z$

## Other comparison operators

The `<utility>` header file (which is included by `<string>`, `<vector>` and other data types) defines

- $a \neq b$  as  $!(a == b)$
- $a > b$  as  $b < a$
- $a \leq b$  as  $!(b < a)$
- $a \geq b$  as  $!(a < b)$

So usually we need only define `==` and `<`, but we can also define the others if required.

You need to declare:

```
using namespace std::rel_ops;
```

## Operators available for overloading

Only built-in operators can be overloaded:

|        |    |    |     |    |    |    |    |    |     |     |
|--------|----|----|-----|----|----|----|----|----|-----|-----|
| unary  | ~  | !  | +   | -  | &  | *  | ++ | -- | ++  | --  |
| binary | +  | -  | *   | /  | %  | ^  | &  |    | <<  | >>  |
|        | += | -= | *=  | /= | %= | ^= | &= | =  | <<= | >>= |
|        | == | != | <   | >  | <= | >= | && |    |     |     |
|        | =  | ,  | ->* | -> | () | [] |    |    |     |     |

Their precedence and associativity can't be changed, so the expressions

$a + b + c * d$                        $(a + b) + (c * d)$

are always equivalent, no matter how the operators are overloaded.

```
++a; is a.operator++();
a++; is a.operator++(int); //dummy argument (ignored)
```

## Output of built-in types

Consider

```
cout << "Total = " << sum << '\n';
```

This is equivalent to

```
((cout << "Total = ") << sum) << '\n';
```

- The operator `<<` is overloaded in `ostream`, not in the C++ language.
- It associates to the left.
- It is defined as a member function of `ostream`, and returns the modified `ostream`.

## The << operator

- The built-in meaning of `<<` is bitwise left shift of integers, so that the expression  $5 \ll 3$  is equal to 40.
- It associates to the left, so  $5 \ll 2 \ll 1$  is also equal to 40.
- It was selected for stream output for its looks. Luckily it associated the right way.
- Different overloads of the same symbol need not have related meanings, or even related return types.

| Bitwise left shift |        |      |
|--------------------|--------|------|
| $5 \ll 0$          | 101    | = 5  |
| $5 \ll 1$          | 1010   | = 10 |
| $5 \ll 2$          | 10100  | = 20 |
| $5 \ll 3$          | 101000 | = 40 |

$$x \ll y = x * 2^y$$
$$x \gg y = x * 2^{-y} = x / 2^y$$



## The ostream class

```
class ostream {
public:
 ostream& operator<<(char c);
 ostream& operator<<(unsigned char c);
 ostream& operator<<(int n);
 ostream& operator<<(unsigned int n);
 ostream& operator<<(long n);
 ostream& operator<<(float n);
 ostream& operator<<(double n);
 ...
};
```

In the **string** header file an independent function:

```
ostream& operator<<(ostream &out, const string &s);
Why not define it as a member function???
ostream& operator<<(ostream &out);
```

## Programming in C++

2024-11-13

### The ostream class

```
The ostream class
class ostream {
public:
 ostream& operator<<(char c);
 ostream& operator<<(unsigned char c);
 ostream& operator<<(int n);
 ostream& operator<<(unsigned int n);
 ostream& operator<<(long n);
 ostream& operator<<(float n);
 ostream& operator<<(double n);
 ...
};
In the existing header file an independent function:
ostream& operator<<(ostream &out, const string &s);
Why not define it as a member function???
ostream& operator<<(ostream &out);
```

### Why not define printing string objects as a member function?

The writer of the **string** class cannot modify the **ostream** class. So if they want to declare it as a member function they can only do so within the class **string**.

But then the meaning of the operator changes – instead of writing `cout << s`; we would have to write `s << cout`; – not what we want!

Do you understand why we'd have to write `s << cout`; to print a **string** `s` on `cout` if we'd have defined **operator<<** as a member function of class **string**?

If you do not, start reading again from slide “Expanding overloaded operators” (slide 8) – repeat until it's clear.

## Output of a user-defined type

```
class point { int _x, _y;
public:
 point(x, y) : _x(x), _y(y) { }
 int x() const { return _x; }
 int y() const { return _y; }
};
```

The output operator for **points** is defined as a non-member function:

```
ostream& operator<<(ostream &s, const point &p) {
 return s << '(' << p.x() << ', ' << p.y() << ')';
}
```

Again – why as a non-member function ???

## Output of a user-defined type – ATTENTION!!!

- 1 Always output to the ostream parameter (**s**), NOT to **cout/cerr**!!!
  - **cout/cerr** might not exist!
  - May want to print to a socket, a **string** buffer...
- 2 Always return the stream received as parameter
  - To enable chaining: `cout << a << b`;
- 3 Return type and the parameters should all be references – the object should be a **const** reference.
  - We need to change the **ostream** (so a reference) and we want to avoid copying the object (so a **const** reference).
- 4 We NEVER print a newline at the end!
  - Some may need to print more things before the newline.
- 5 We output the bare minimum – nothing more!  
Never print things such as:  
“The point object is (3,4)”
- 6 We keep the output simple and easy to READ BACK.
  - We must be able to eat our own dog food!

## Using various versions of the << operator

Suppose we have an expression  $a \ll b$ , where  $a$  has type **A**, and  $b$  has type **B**. Then the relevant definition of  $\ll$  could be either

- a method of class **A** taking one argument of type **B**:  
`ReturnType A::operator<<(B x)`
- or an independent function (not a method in a class) taking two arguments of types **A** and **B**:  
`ReturnType operator<<(A x, B y)`

For example the following uses a mixture of these:

```
point p(2,3);
cout << "The point is " << p << '\n';
```

Can you identify which occurrences of the  $\ll$  operator are independent functions and which are member functions?

(Hint: Think which types were already known to whomever wrote the *ostream* class.)

## On accessing private state: Friend (or NOT)

An accidental consequence of the way operators are defined in C++:

- An operator defined as a member function has access to the private and protected fields of its first argument, but not its second (when the second is an object of a different class).
- Sometimes this is not what we want (e.g. for  $\ll$  and  $\gg$  of user-defined types).
- One work-around is to declare the operator as a **friend** of the second class.
- Even **better** to use a helper member function:

```
class point {
public:
 ostream& print_on(ostream &s) const { /* *CONST* !!!
 return s << ' (' << _x << ', ' << _y << ')'; }
};
ostream& operator<<(ostream &s, const point &p) {
 return p.print_on(s); }
```

## Input of built-in types

Input is almost the mirror image of output:

```
int x, y, z;
cout << "Please type three numbers: ";
cin >> x >> y >> z;
```

- Again  $\gg$  is overloaded: it knows what to look for based on the type of its argument.
- It also associates to the left, and returns an **istream**.
- By default,  $\gg$  will skip white space before the item; in this mode you will not see a space, newline, etc.

## The *istream* class

```
class istream : virtual public ios {
public:
 istream& operator>>(char &c);
 istream& operator>>(unsigned char &c);
 istream& operator>>(int &n);
 istream& operator>>(unsigned int &n);
 istream& operator>>(long &n);
 istream& operator>>(float &n);
 istream& operator>>(double &n);
 ...
};
```

In the **string** header file, as an **independent** function:

```
istream& operator>>(istream &in, string &s);
```

## The state of an `istream`

The following methods of `istream` test its state:

`bool eof()` the end of the input has been seen.

`bool fail()` the last operation failed.

`bool good()` the next operation might succeed.  
(Equivalent to `! eof() && ! fail()`.)

`bool bad()` the stream has been corrupted: data has been lost  
(data was read but not stored in an argument).  
(Implies `fail()`, but not vice-versa.)

A test “`if (s)`” is equivalent to “`if (! s.fail())`”

## Input of a user-defined type

```
istream& operator>>(istream &s, point &p) {
 int x, y;
 char lpar, comma, rpar;
 if (s >> lpar) { //not met EOF (End Of File)
 if ((s >> x >> comma >> y >> rpar) &&
 (lpar == '(' && comma == ',' && rpar == ')'))
 p = point(x, y); // *constructor*, not setters!
 else
 s.setstate(ios::badbit); //read failed
 }
 return s;
}
```

When “`if (s >> lpar)`” fails, that means there is no more input.  
We have not read any data so far, so have not corrupted the input.  
Therefore, we simply return the input stream.

## Input of a user-defined type – ATTENTION!!!

- ❶ Always read from the stream received as parameter – NEVER `cin`!
  - `cin` may not exist!
  - May want to read from a file/buffer/socket. . .
- ❷ Always return the stream received as parameter
  - To allow checking for input success.
  - To allow for chaining.
- ❸ Return all parameters should be references (non-`const`).
- ❹ Set the badbit if there's a problem (*i.e.*, you've read something but cannot use it to set your object) – failing to read **anything at all** because of an EOF is NOT a problem.
- ❺ Always read what you print – always (so, keep the format simple!).
- ❻ NEVER use `getline()` – you're corrupting the stream!
- ❼ NEVER read into a `string` and parse that – stream corruption!
- ❽ NEVER, EVER print anything!
- ❾ Prefer constructors over setter member functions.  
*Avoid setters altogether – not very OO. Same with getters. . .*

## Getters/Setters – Why Not

- **Avoid getters**
  - Objects should be asked to do tasks themselves:  
`point1.move(3,5);`  
`shape2.scale(.5);`  
`employee3.clock_in(log_register); etc.`
  - When you're using getters, you end up doing the task yourself using the state data you got.  
*But that's procedural, not OO programming. . .*
- **Avoid setters** (OK, you can write *Fortran* in any language. . .)
  - Object's state should only change because of actions **they**'ve performed **on your behalf**, not because you've done a task and are now giving them the results.  
Don't spoon-feed your objects – they can take care of themselves.
  - Setters need to preserve the class invariance.  
Much easier to get this right once (in the constructors) and re-use the constructors from that point on.
- **Delegate!** “What can I ask an object of this class to do for me?”

## Next session

- Genericity (parametric polymorphism)
- Template classes and functions in C++.
- Reading: Savitch 16.1–2; Stroustrup 13.2–3; Horstmann 13.5.
- Introducing the Standard Template Library: some container classes.
- Reading: Savitch 19.1; Stroustrup 16.2.3, 16.3; Horstmann 13.5.

## 2024-11-13 Programming in C++

Next session

### Next session

- Genericity (parametric polymorphism)
- Template classes and functions in C++
- Reading: Savitch 16.1–2; Stroustrup 13.2–3; Horstmann 13.5.
- Introducing the Standard Template Library: some container classes.
- Reading: Savitch 19.1; Stroustrup 16.2.3, 16.3; Horstmann 13.5.

## Final Notes

- **a + b**, can be either **a.operator+(b)** or **operator+(a, b)**. All methods receive the current object (**\*this**) as their implicit first argument.
- Avoid friend functions – use helper methods.  
*“Treat your friend as if he might become an enemy.”* – Publilius Syrus, 85-43 BC.
- Output: Read again slides 17–18. Repeat.
- Input: Read again slides 24–25. Repeat.
- More on Operators:  
<https://www.cplusplus.com/doc/tutorial/operators/>
- More on Operator overloading:  
<https://en.cppreference.com/w/cpp/language/operators>
- More on friends: <https://isocpp.org/wiki/faq/friends>

## Programming in C++

### Session 4 – Genericity, Containers

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

## Polymorphism

Code that works for many types.

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

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 code looks similar for different types
- Very common for *container* types (vectors, lists, *etc.*)  
Reversing vectors? Same whatever the element types
- Reuse: separate what varies (the type of the elements) from what doesn't (the code), and reuse the latter
- Instead of writing *N* similar versions, we will write 1 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;
}
```

**Cannot write this int swap in Java**

(if you don't get this, you don't get parameter passing)

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.

## Programming in C++

2024-11-13

### A generic swapping procedure

**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>**  
**T** is a type parameter, supplied upon function/class use
- Can also use **class** instead of **typename**
- Multiple parameters? Sure!  
**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`!!!

## Programming in C++

2024-11-13

### A generic reversal procedure

**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--
 }
}
```

**Practical 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](https://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.

## Programming in C++

2024-11-13

### A generic reversal procedure

**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--
 }
}
```

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

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"

[elements.ofprogramming.com/](https://elements.ofprogramming.com/)

Even better – use one of the standard C++ algorithms if applicable!

[en.cppreference.com/w/cpp/algorithm](https://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](https://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)

### Another example

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: Better because simpler.

```
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 `x` as a constant inside `member`.

`const Elem & x` does promise that (and avoids copying potentially large objects).

### Another example

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)

Optimisation 3: Best because simplest/most robust/potentially fastest

```
template <typename Elem>
bool member(const Elem & x, const vector<Elem> & v) {
 for (const auto & elem : v) //range-based for loop
 if (elem == x) return true;
 return false;
}
```

We're not programming in Fortran anymore – stop using array indices in loops.



## Bounded genericity

- Some generic definitions use functions not defined for all types (e.g. `member` uses `==`)
- C++ checks this when specializing the definition for a 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;`

## Programming in C++

2024-11-13

### Bounded genericity

#### Bounded genericity

Some generic definitions use functions not defined for all types (e.g. `member` uses `==`)  
C++ checks this when specializing the definition for a 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](https://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).

Or NOT: `auto p = make_pair(3, 4);`  
`auto n = make_pair(12, string("twelve"));` // can't always

## Programming in C++

2024-11-13

### A generic class

#### 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).  
NOT: auto p = make\_pair(3, 4);  
auto n = make\_pair(12, string("twelve")); // can't always

- 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)

**Same things? vector/list/set Different things? pair/tuple**

Did you notice the **initializer list constructors**?

```
vector<int> p1 = {3, 4}; int p2[2] = {3, 4};
```

[https://www.cplusplus.com/reference/initializer\\_list/](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`
    - `tuple`
  - Associative Containers
    - `set`
    - `map`

- Iterators

See [en.cppreference.com/w/cpp/container](http://en.cppreference.com/w/cpp/container)

Just taught you about *deque*, *tuple* 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();
};
```

### The vector class

```
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

- list: a **sequence** of items of the **same type**, that can be efficiently modified at the ends
- Can access the first/last **element** and add/remove elements at **either** end
- All these operations are fast, independently of the size of the list
- Lists are implemented as linked structures (with pointers)
- Other uses of lists require *iterators* (covered next session)

If you don't have an ordering (so no “sequence”), then don't use **vector**/**list**, use **set**/**multiset** instead!

“Fast” ==  $O(1)$

“Slow” ==  $O(n)$

## 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](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` !!!

## Programming in C++

2024-11-13

### Defining methods outside the class

#### 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 a 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 “ $days["March"] = 31;$ ” ? ]

### The map class

- 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 &!**).

```
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? => Use m.count(k)
[What does "days[m]" mean? Or "days["March"] = 31;" ?]
```

## 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.

## Programming in C++

2024-11-13

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 – 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!)

## Programming in C++

2024-11-13

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)

## Programming in C++

2024-11-13

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)

# Programming in C++

## Session 5 – Pointers and Arrays

### Iterators

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)



CITY  
ST GEORGE'S  
UNIVERSITY OF LONDON

Copyright © 2005 – 2024

## Introduction

- Pointers and arrays: from C with love  
(Savitch 10.1; Stroustrup 5.1–3; Horstmann 9.7)
- *Iterators*: for sequential access to container elements  
(Savitch 17.3 and 19.2; Stroustrup 19.2)
- STL iterator interface based on that of pointers and arrays
- STL: many generic functions operating on iterators.  
In the STL, these are called *algorithms* `<algorithm>`

## Pointers and arrays

- C's arrays, pointers and pointer arithmetic survive in C++.
- Arrays are mostly superseded by vectors.
- C/C++ pointers support arithmetic, but this is little used in C++.
- Many uses of pointers are superseded by references, but they still have their uses:
  - Subtype polymorphism.
  - Dynamically allocated objects (sessions 8 and 9).
  - Dynamic data structures.
  - Legacy interfaces.
  - Accessing hardware directly.

## Pointers in C and C++

- Pointer variables are declared with `*`  
`int *ip;`  
This does **NOT** initialize the pointer. (*not null, **garbage** value!*)
- The address of a piece of storage, obtained with `&`, is a pointer:  
`int i;`  
`ip = &i;`
- Pointers are dereferenced with `*`  
`*ip = *ip + 3;`  
In general, `*` and `&` are inverses.
- `&` the **address-of** operator
- `*` the **dereference** operator

**Note:** Beware of multiple variable definitions!

`int *ip1, ip2;` // `ip1` is a pointer, `ip2` is an int

Why? `*ip1` is an int – so is `ip2`. The `*` operator binds with the name, not the type.

## Pointers vs References

Given the definition of two integer variables: `int i = 3, j = 4;`

|                              | References                      | Pointers                         |
|------------------------------|---------------------------------|----------------------------------|
| <b>Declaration</b>           | <code>int &amp;ref = i;</code>  | <code>int *ptr = &amp;i;</code>  |
| <b>Reading the integer</b>   | <code>cout &lt;&lt; ref;</code> | <code>cout &lt;&lt; *ptr;</code> |
| <b>Assigning the integer</b> | <code>ref = 5;</code>           | <code>*ptr = 5;</code>           |
| <b>Using another integer</b> | <b>N/A</b>                      | <code>ptr = &amp;j;</code>       |

- `ptr` is an **actual variable**, allocated somewhere in memory.
- A `ref` is more like a `const` pointer (`int * const r = &i;`), with an easier interface (no `*` and `&`), and the additional assertions that `r != nullptr` AND that `r` refers to a **VALID** object

(On a 16 bit computer:)

```
1024 | 3 | i, ref // i, j, ref, ptr:
1040 | 4 | j // names, not in memory
1056 | 1024 | ptr (holds the address of i)
1072 | ... | (other - possibly garbage)
1088 | ... | (other - possibly garbage)
```

## Undefined pointers

- The storage pointed to by a pointer may become undefined. There will be no warning from the compiler or runtime system:

```
int *p;
{
 int i = 5;
 p = &i;
} // i ceases to exist
*p = 3; // undefined behaviour
```

Like a telephone number that has gone out of use – calling it doesn't reach anyone (or may reach another person).

It is the **programmer's responsibility** to ensure that the pointer points at something **valid** whenever it is dereferenced.

- BTW, local variable pointers are **not initialized** (no basic type is).  
⇒ `p`'s initial value is **garbage**.

## Null pointers

- The value 0 in pointer types is distinct from any address.  
`int *ip = 0;`  
cf. `null` in Java.
- Since C++11 one should use `nullptr` instead of 0 – avoid using `NULL` (comes from C).
- Pointers that are global variables are initialized to `nullptr`
- Again, pointers that are local variables are not initialized.

## More pointers

- The following declaration

```
const int *p;
```

means that things pointed to by `p` cannot be changed through `p` (but `p` itself can be changed.)

- Read it from right to left till the `*`, then left to right:  
“`p` is a pointer (`*`) to a constant (`const`) integer (`int`).”

- It is possible to have pointers to pointers:

```
int i; // I solve problems...
int *p1 = &i; // I know a guy who solves problems
int **p2 = &p1; // I know a guy who knows a guy...
int ***p3 = &p2; // I know a guy who knows a guy who knows a guy...
```

- These may be qualified with `const` in various ways:

```
int * p1; // a pointer to an int
const int * p2; // a pointer to a const int
int * const p3; // a const pointer to an int
const int * const p4; // a const pointer to a const int
```



More pointers

**More pointers**  
• The following declaration  
const int \* p;  
means that things pointed to by p cannot be changed through p.  
(But p itself can be changed.)  
• Should have right to left of the =, not left to right.  
• p is a pointer (C) to a constant (number) (integer) (int).  
• It is possible to have pointers to pointers.  
int a; // 2 memory addresses.  
int \* p1 = &a; // 2 know a guy who knows a guy.  
int \* p2 = p1; // 2 know a guy who knows a guy.  
int \* p3 = p2; // 2 know a guy who knows a guy.  
int \* p4 = p3; // 2 know a guy who knows a guy.  
• These may be qualified with const to restrict usage.  
const int \* p1; // a pointer to a const int.  
const int \* const p2; // a const pointer to a const int.

```
int * * p1; // a pointer to a pointer to an int
const int * * p2; // ???
int * const * p3; // ???
int * * const p4; // ???
const int * const * p5; // ???
const int * * const p6; // ???
int * const * const p7; // ???
const int * const * const p8; // ???
```

Pointers to objects

Given a class

```
class point {
public:
 int x, y;
 point (int xx, int yy) : x(xx), y(yy) {}
};
```

We can refer to members as follows:

```
point my_point(2, 3);
point *p = &my_point;
cout << (*p).x << '\n';
```

or equivalently as

```
cout << p->x << '\n';
```

and similarly for member functions.

Arrays

We have already used vectors, but C++ also has arrays, which are fixed in size:

```
int arr[40];
for (std::size_t i = 0; i < 40; ++i)
 arr[i] = arr[i] + 5;
```

Unlike Java, there is no check that the index is in bounds.

Advice:

- Use `vector<T>` instead when the size is unknown
- With a fixed size use `array<T>` instead!

(Help the compiler – it'll pay you back!)

Arrays

**Arrays**  
We have already used vectors, but C++ also has arrays, which are fixed in size.  
int arr[40];  
for (std::size\_t i = 0; i < 40; ++i)  
 arr[i] = arr[i] + 5;  
Unlike Java, there is no check that the index is in bounds.  
**Advice:**  
• Use `vector<T>` instead when the size is unknown.  
• With a fixed size use `array<T>` instead!  
(Help the compiler – it'll pay you back!)

We can find the length of an array using the `sizeof` function:

```
int l = sizeof(arr) / sizeof(int);
Only works if arr is the name of the array, not if it's a pointer...
```

```
sizeof (Name of the array)
/ sizeof (Type of the elements)
```

## Pointers and arrays

When assigning or initializing from an array, a pointer to the first element is copied, not the array:

```
int arr[40];
int *p = arr; // What's arr ???
```

Now `*p` is equivalent to `arr[0]`, and indeed to `*arr`.  
The following are all equivalent:

```
arr[0] = arr[0] + 5;
*p = *p + 5;
*arr = *arr + 5;
```

## Parameter passing

Parameter passing is a form of initialization, so an array

```
int arr[40];
```

can be passed as a pointer parameter:

```
void f(int *p) { ... }
```

Functions that take a pointer to a **single** element look the same.  
(pointer passing less common in C++ than in C, thanks to references)

But it might be used if we want to:

- re-use a C library; or
- write a C++ library that may be used by C programs as well.

## C-style strings

- In C, strings are stored in `char` arrays, with the end of the string marked by a `'\0'` character delimiter  
`char name[]="Bill";` //array of 5 chars  
`char *name2="Fred";` //pointer to a `*const*` array of 5 chars
- Often `char *` indicates a C-style string, e.g.,  

```
int main(int argc, char **argv);
```
- C++'s `string` is **MUCH** safer
- A C-style string can be used where a `string` is expected, and is automatically converted  
That's done with constructor `string(char *s);`
- If you need a C-style string for some legacy interface, use the method `c_str()` of `string`  
For example, `string s; char *p = s.c_str(); foo(p);`

## Pointer arithmetic

When `p` has type `T *`, and points to the  $i^{\text{th}}$  element of an array of `T`s:

```
T arr[N];
T *p = arr + i; // WOULD BE NICE IF i < N ...
```

Then:

- `p + k` is a pointer to the  $(i + k)^{\text{th}}$  element.
- `++p` is equivalent to `p = p+1`
- `p - k` is a pointer to the  $(i - k)^{\text{th}}$  element.
- `--p` is equivalent to `p = p-1`
- `p[k]` is equivalent to `*(p+k)` **DEFINED IFF** `arr ≤ p+k < arr+N`

Again, there are no checks that anything is in bounds.

Can also subtract two pointers (`ptrdiff_t`), which should be pointers to the same array (\*NOT\* checked of course...).

```
T *p1 = arr + i;
T *p2 = arr + j;
ptrdiff_t diff = p2 - p1; // = j - i
```

## A Game!!!

Consider:

```
int arr[] = {1, 2, 3, 4, 5};
int *p = arr;
```

Which are *legal*, which are *illegal*?

- 1 `p[2]`
- 2 `2[p]`
- 3 `p + 2`
- 4 `arr[2]`
- 5 `2[arr]`
- 6 `arr + 2`

**ONLINE QUIZ NOW!** [t.ly/KHCjm](https://t.ly/KHCjm)

**You've got 3 mins!**

*What do the legal ones mean?*

## Looping over an array

Given an array of integers:

```
int arr[40];
```

The following are (functionally) equivalent:

- Using offsets (*slower*):

```
for (std::size_t i = 0; i < 40; ++i)
 arr[i] = arr[i] + 5;
```
- Using pointers (*faster*):

```
int *end = arr + 40;
for (int *p = arr; p != end; ++p)
 *p = *p + 5;
```

Notes:

- `arr + 40` **SHOULDN'T** be dereferenced (*undefined behaviour*)
- Pointer loop is **faster!** (*why?*)

## Iterators

Iterators are objects providing sequential access to container elements

- The Java interface is analogous to a linked list or a stream:

```
public interface java.util.Iterator {
 boolean hasNext();
 Object next();
 void remove(); // not always supported
}
```
- C++ STL iterators are modelled after array pointers

## Iterators in the STL

Iterating over a list of strings:

```
list<string> names;
...
for (list<string>::iterator p = names.begin();
 p != names.end(); ++p)
 cout << *p << '\n';
```

Sequences include a type **iterator** and two iterators:

**begin()** positioned **at the start** of the sequence, and  
**end()** positioned **just past the end** of the sequence.

Each iterator supports the operators ==, ++ and \*.

- For `int *p` we now have `list<int>::iterator p`.
- What about `const int *p`?  
`list<int>::const_iterator p` (one word, with a hyphen)  
`c.begin()/c.end()` become `c.cbegin()/c.cend()`

## Programming in C++

2024-11-13

### Iterators in the STL

**Iterators in the STL**  
Iterating over a list of strings:  
list<string> names;  
for (list<string>::iterator p = names.begin();  
 p != names.end(); ++p)  
 cout << \*p << '\n';  
Sequences include a type **iterator** and two iterators:  
**begin()** positioned **at the start** of the sequence, and  
**end()** positioned **just past the end** of the sequence.  
Each iterator supports the operators ==, ++ and \*.  
For int \*p, we now have list<int>::iterator p.  
What about const int \*p?  
list<int>::const\_iterator p (one word, with a hyphen)  
c.begin()/c.end() become c.cbegin()/c.cend()

- Prefer using `begin(container)` and `end(container)`
- Instead of `container.begin()` and `container.end()`
  - The former form works with arrays as well; \*and\*
  - It selects `container.begin()` or `container.cbegin()` automatically, depending on whether `container` is `const` or not.

## A variation: typedefs

In C++ we can define new names for types using **typedef**:

```
typedef int time;
typedef char * cstr;
typedef deque<string> phrase;
typedef vector<vector<double> > matrix;
```

(can also do this in C, but only outside functions)

With **typedef** we can introduce an abbreviation for the iterator type:

```
typedef list<string>::iterator iter;
for (iter p = begin(names), e = end(names);
 p != e; ++p)

 cout << *p << '\n';
// Or *better*: USE auto!
for (auto p = begin(names), e = end(names);
 p != e; ++p)

 cout << *p << '\n';
```

## The analogy

| C             |              | STL – C++98    |                      |
|---------------|--------------|----------------|----------------------|
| array         | arr          | container      | c                    |
| pointer       | p            | iterator       | p                    |
| start pointer | arr          | start iterator | c.begin() / cbegin() |
| end pointer   | arr + LENGTH | end iterator   | c.end() / cend()     |
| increment     | ++p          |                | ++p                  |
| dereference   | *p           |                | *p                   |

### Since C++11 – One API for all!

|               |            | Since C++11    |          |
|---------------|------------|----------------|----------|
| array         | arr        | container      | c        |
| pointer       | p          | iterator       | p        |
| start pointer | begin(arr) | start iterator | begin(c) |
| end pointer   | end(arr)   | end iterator   | end(c)   |
| increment     | ++p        |                | ++p      |
| dereference   | *p         |                | *p       |

**begin(c)** returns a const/non-const iterator as appropriate! :-)

## Iterator is a concept

- “Iterator” is an STL concept, not a C++ class.
- All iterators support the same operations in the same way:
  - Switching representations is relatively easy
  - Generic code can be written using these operations
- Special kinds of iterators support more operations
- Checking is done when generic code is instantiated

## Iterator concepts in the STL

Different containers have different iterator kinds, belonging to a hierarchy of iterator concepts:

**Input Iterator** supports `==`, `++`, (unary) `*` and `->`  
e.g., the **iterator** of **forward\_list** [`*`]

**Bidirectional Iterator** supports all these as well as `--`  
e.g., the **iterator** of **list**

**Random Access Iterator** supports all these as well as `<`, `+`, `-` and `[]`, which should behave similarly to operations on pointers  
e.g., the **iterator** of **vector** or **deque**

- Why isn't `<` supported for input/bidirectional iterators?
- What does `iter[3]` stand for?

[`*`] **forward\_list**: née **slist**, see issue:  
[stackoverflow.com/a/6885508](https://stackoverflow.com/a/6885508)

## A generic function

```
template <typename Iterator, typename Elem>
int count(Iterator start,
 Iterator finish, const Elem & v) {
 int n = 0;
 for (Iterator p = start; p != finish; ++p)
 if (*p == v)
 n++;
 return n;
}
```

There are several type requirements here (checked at instantiation):

- **Iterator** must be at least an **input** iterator type;
- **Iterator** must be an iterator with element type **Elem**; and
- The **Elem** type must support `==`.

## Using the generic count function

Function `count` is defined in `<algorithm>`.

Here is an example of its use:

```
list<string> names;
string s;
....
int n = count(begin(names), end(names), s);
cout << s << " occurs " << n << " times\n";
```

In the above use,

- **Iterator** is `list<string>::iterator`
- **Elem** is `string`

Check `<algorithm>` out! [en.cppreference.com/w/cpp/algorithm](https://en.cppreference.com/w/cpp/algorithm)

## Iterating over associative containers

- A **map** associates keys with values.
- The **iterator** of a map produces **pairs** of key and value.
- If **p** is a **map<K, V>** iterator, then **\*p** has type **pair<const K, V>**.

```
map<string, int> table; // How to print map's elements?
...
typedef map<string, int>::iterator Iter;
for (Iter p = begin(table); p != end(table); ++p)
 cout << p->first << " -> " << p->second << '\n'; // Or
for (auto p = begin(table); p != end(table); ++p)
 cout << p->first << " -> " << p->second << '\n'; // Or
for (const auto &pr : table) // range for
 cout << pr.first << " -> " << pr.second << '\n'; // Or
for_each(begin(table), end(table),
 [](const auto &pr) { // a lambda function
 cout << pr.first << " -> " << pr.second << '\n';
 }); // for_each can be ***PARALLELIZED***!!!
```

## Programming in C++

2024-11-13

### Iterating over associative containers

Iterating over associative containers

- A map associates keys with values.
- The iterator of a map produces pairs of key and value.
- If p is a map<K, V> iterator, then \*p has type pair<const K, V>.

```
map<string, int> table; // How to print map's elements?
...
typedef map<string, int>::iterator Iter;
for (Iter p = begin(table); p != end(table); ++p)
 cout << p->first << " -> " << p->second << '\n'; // Or
for (auto p = begin(table); p != end(table); ++p)
 cout << p->first << " -> " << p->second << '\n'; // Or
for (const auto &pr : table) // range for
 cout << pr.first << " -> " << pr.second << '\n'; // Or
for_each(begin(table), end(table),
 [](const auto &pr) { // a lambda function
 cout << pr.first << " -> " << pr.second << '\n';
 }); // for_each can be ***PARALLELIZED***!!!
```

```
#include <string>
#include <iostream>
#include <algorithm>
#include <execution>

std::map<std::string, int> table;
std::for_each(std::execution::par_unseq,
 // instance of parallel_unsequenced_policy
 std::begin(table), // start from.
 std::end(table), // end before.
 // a lambda (anonymous) function
 [](const auto &pair) {
 std::cout << pair.first
 << " -> "
 << pair.second
 << std::endl;
 }); // std::for_each ***PARALLELIZED***!!!
```

(doesn't make much sense to parallelize stream output, since the stream sequences the output from the different parallel printers)

Check out [en.cppreference.com/w/cpp/algorithm/reduce](https://en.cppreference.com/w/cpp/algorithm/reduce)

## Summary

- Some features inherited from C:
  - arrays** mostly superseded by **vector<T>** (& **array<T>**)
  - pointers** most useful for dynamic binding & structures  
Mostly superseded by references & smart pointers  
(**unique\_ptr<T>**, **shared\_ptr<T>**, **weak\_ptr<T>**)
- Iterators: sequential access to container elements
- STL iterators look like pointers (++ , \* , -- etc)
- Many generic functions use iterators **STUDY <algorithm> !!!**
- After the reading week:
  - inheritance in C++  
(Savitch 14, 15 and 16.3; Stroustrup 12; Horstmann 14)
  - Genericity and inheritance

## Programming in C++

2024-11-13

### Summary

Summary

- Some features inherited from C:
  - arrays** mostly superseded by **vector<T>** (& **array<T>**)
  - pointers** most useful for dynamic binding & structures  
Mostly superseded by references & smart pointers  
(**unique\_ptr<T>**, **shared\_ptr<T>**, **weak\_ptr<T>**)
- Iterators: sequential access to container elements
- STL iterators look like pointers (++ , \* , -- etc)
- Many generic functions use iterators **STUDY <algorithm> !!!**
- After the reading week:
  - inheritance in C++  
(Savitch 14, 15 and 16.3; Stroustrup 12; Horstmann 14)
  - Genericity and inheritance

(Area left empty on purpose)

- Learn how to write generic functions that take iterators (slide 23)

```
// *** The ONE, TRUE strcpy!!! ***
void strcpy3(const char *s, char *t) {
 while ((*t++ = *s++)) /* extra parentheses added
 to get rid of warning */
 ; /* do nothing in the body - loop condition
 does the job */
}
/*
 * Source: Kernighan & Ritchie, The C Programming
 * Language, 2nd Edition, Prentice Hall PTR, 1988,
 * p. 106
 *
 * strcpy: copy s(source) into t(target).
 * ASSUMPTION: t(target) has enough space for the
 * string inside s(source)!
 */
```

## Programming in C++

### Session 6 – Inheritance 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)



**CITY**  
ST GEORGE'S  
UNIVERSITY OF LONDON

Copyright © 2005 – 2024

The most important slide of the lecture

**Why use inheritance?**

## Reasons for Inheritance (revision)

- **Implementation Re-Use** *Bad-ish...* [+]
  - new classes *extend* existing classes with additional fields and methods, and can *override* the definitions of existing methods.
- **Interface/Type Hierarchies (Is-A relations [\*])** *Good!*
  - the new class is also a *subtype* of the old: its objects can be used wherever objects of the old class can (*subtype polymorphism*) with the appropriate method selected by *dynamic binding*.
  - *abstract classes* declare methods without defining them: the methods are defined in subclasses.

[\*] **Is-A** vs **Has-A** relations:

- A car **Is-A** vehicle Inheritance
- A car **Has-A** steering wheel Composition

[+] Just have an object of that class as a field (composition) and have your methods forward calls to it.

## Inheritance in C++

The basic concept is similar to Java, but

- different syntax
- **objects** of subclasses may be assigned to **object** variables of superclasses, by *slicing* off the extra parts.
- interactions with:
  - overloading
  - pointers
  - template classes



## Inheritance syntax in Java and C++

- in Java:
 

```
public class holiday extends date {
```
- in C++:
 

```
class holiday : public date {
```

 we will always use `public` inheritance.
- C++ terminology: `date` is a *base class*; `holiday` is a *derived class*.
- multiple inheritance (in C++):
 

```
class child : public parent1, public parent2 {
```
- there are no interfaces in C++.

## A base class

Recall the class `date` from session 2:

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

public:
 date(); // today's date
 date(int d, int m);
 date(int d, int m, int y);
 int get_day() const { return day; }
 int get_month() const { return month; }
 int get_year() const { return year; }
};
```

## Inheritance and initialization

The members of base class(es) are initialized similarly to subobjects:

```
class holiday : public date {
 string name;
public:
 holiday(string n) : date(), name(n) {}

 holiday(string n, int d, int m) :
 date(d, m), name(n) {}

 string get_name() const { return name; }
};
```

Members of the base class can't be initialized directly:

**use constructor date  
(can only use it in the initialisation list!)**

## Order of initialization – IMPORTANT!

Initialization is done in the following order:

- 1 constructors for base classes
- 2 members (in order of declaration) – WARNING!!! BE CAREFULL!!!
- 3 body of constructor

Principle: **The constructor body needs a fully initialised object!**

Danger: **Order of initializers has \*NO\* effect,  
only declaration order matters!**

So: Initialise members with order-independent expressions

## Initialization and assignment

- As in Java, we can initialize and assign from derived classes, but here **objects** are copied, not pointers:

```
holiday h("Anzac Day", 25, 4);
date d = h;
```

initializes **d** as a copy of the **date** part of **h**

```
d = h;
```

copies the **date** part of **h** into **d**

- In both cases, the object is **sliced**

**Note:** *Call-by-value initialises a new variable, so it also involves copying (and slicing)*

## Method overriding in Java and C++

The default in C++ is the **opposite** to that in Java:

- in Java:

```
final int non_redefinable_method() { ... }
 int redefinable_method() { ... }
abstract int undefined_method();
```

- in C++:

```
int non_redefinable_method() { ... }
virtual int redefinable_method() { ... }
virtual int undefined_method() = 0;
```

The latter is called a **pure virtual** function.

When a method is declared **virtual** in a base class, it is also **virtual** in derived classes (the keyword there is optional).

### Why is it the opposite?

### Method overriding in Java and C++

The default in C++ is the opposite to that in Java:

```

In Java:
class int non_redefinable_method() { ... }
 int redefinable_method() { ... }
abstract int undefined_method();

In C++:
int non_redefinable_method() { ... }
virtual int redefinable_method() { ... }
virtual int undefined_method() = 0;

The latter is called a pure virtual function.
When a method is declared virtual in a base class, it is also
virtual in derived classes (the keyword there is optional).

```

Why is it the opposite?

- It's the opposite because non-redefinable member functions are faster than redefinable (virtual) ones. (C++'s #1 aim is speed!)
- Redefinable member functions are actually *pointers* to functions – at run time the code has to dereference the pointer held in the class information of the current object to figure out which code to execute.
- This also explains the bizarre syntax for abstract (pure virtual) member functions:
  - " = 0" means that the function pointer is the **nullptr**, i.e., there's no respective code for it!

## Method overriding

Overridable methods must be declared **virtual**:

```
class date {
 ...
 virtual string desc() const { ... }
};
```

Overriding in a derived class:

```
class holiday : public date {
 ...
 virtual string desc() const {
 return name + " " + date::desc();
 }
};
```

**Note:** *Qualify* with the class name to get the base version.

## Static and dynamic binding

Given functions

```
void print_day1(date d) {
 cout << "It's " << d.desc() << '\n';
}

void print_day2(date &d) {
 cout << "It's " << d.desc() << '\n';
}
```

then

```
holiday xmas("Christmas", 25, 12, 2004);
print_day1(xmas); // It's 25/12/2004
print_day2(xmas); // It's Christmas 25/12/2004
```

### Why the different behaviour?!

(the answer is on slide 9)

## Programming in C++

### Inheritance

#### Static and dynamic binding

### Dynamic Binding

In order to get dynamic binding we need:

- 1 a type hierarchy (inheritance)
- 2 some virtual member functions
- 3 references or pointers to objects  
(so that the compiler isn't sure what the real object type is)

If you don't need Dynamic Binding, then you don't need Inheritance!

You can simply use composition and (implicit) conversion:

```
class holiday {
 date d;
 ...
 operator date() const { return d; } // convert to date
};

void print_month(date d) { // works with holiday objects too
 cout << d.get_month() << endl;
}
```

Static and dynamic binding

```

// Static binding
void print_day1(date d) {
 cout << "It's " << d.desc() << '\n';
}

// Dynamic binding
void print_day2(date &d) {
 cout << "It's " << d.desc() << '\n';
}

// Test
holiday xmas("Christmas", 25, 12, 2004);
print_day1(xmas); // It's 25/12/2004
print_day2(xmas); // It's Christmas 25/12/2004

// Why the different behaviour?
// (the answer is on slide 9)
```

## Abstract classes

A class containing a pure virtual function is *abstract*, though this is not marked in the syntax.

```
class pet {
protected:
 string _name;
public:
 pet(string name) : _name(name) {}
 virtual string sound() const = 0;
 virtual void speak() const {
 cout << _name << ": " << sound() << "!\n";
 }
};
```

As in Java, abstract classes may not be instantiated, so no variable may have type `pet`, but we can declare a reference (or a pointer).

## Derived classes

```
class dog : public pet {
public:
 dog(string name) : pet(name) {}
 string sound() const { return "woof"; }
 void speak() const { // virtual is optional
 pet::speak();
 cout << '(' << _name << " wags tail)\n";
 }
};

class cat : public pet {
public:
 cat(string name) : pet(name) {}
 virtual string sound() const { return "miao"; }
};
```

## Subtype polymorphism and dynamic binding

We cannot pass `pets` by value, but we can pass them by reference:

```
void speakTwice(const pet &a_pet) {
 a_pet.speak();
 a_pet.speak();
}
```

Then we can write

```
dog a_dog("Fido");
speakTwice(a_dog);
cat a_cat("Tiddles");
speakTwice(a_cat);
```

**Why can't we pass `a_pet` by value to `speakTwice`?**

## Programming in C++

### └ Inheritance

2024-11-13

### └ Subtype polymorphism and dynamic binding

**Subtype polymorphism and dynamic binding**

We cannot pass `pets` by value, but we can pass them by reference:

```
void speakTwice(const pet &a_pet) {
 a_pet.speak();
 a_pet.speak();
}
```

Then we can write:

```
dog a_dog("Fido");
speakTwice(a_dog);
cat a_cat("Tiddles");
speakTwice(a_cat);
```

**Why can't we pass `a_pet` by value to `speakTwice`?**

Because

- call-by-value involves **instantiating** a new local object, which is initialised using the original parameter (see slide 9); and
- `a_pet` is an abstract class, so we cannot instantiate it. .

## Caution: inheritance and overloading

```
class A {
 virtual void f(int n, Point p) { ... }
};
```

Now suppose we intend to override `f` in a derived class, but make a mistake with the argument types:

```
class B : public A {
 void f(Point p, int n) { ... }
};
```

***`f` will be accepted as a definition of a new and different member function.***

***Even forgetting a single `const` or changing a `*` to a `&` means it's a different function!***

```
class B : public A {
 void f(Point p, int n) override { ... }
};
```

## Programming in C++

### └ Inheritance

2024-11-13

### └ Caution: inheritance and overloading

**Caution: inheritance and overloading**

```
class A {
 virtual void f(int n, Point p) { ... }
};
```

Now suppose we intend to override `f` in a derived class, but make a mistake with the argument types:

```
class B : public A {
 void f(Point p, int n) { ... }
};
```

***`f` will be accepted as a definition of a new and different member function.***

***Even forgetting a single `const` or changing a `*` to a `&` means it's a different function!***

```
class B : public A {
 void f(Point p, int n) override { ... }
};
```

How can you protect yourself against such mistakes?

Since C++11 there's a new keyword **`override`** that you can use to state that you're trying to override a member function of one of your base classes:

```
class B : public A {
 void f(Point p, int n) override { ... }
 // Now the compiler catches the error
};
```

There's also a keyword **`final`** to state that derived classes should not be allowed to further override the member function:

```
class A {
 virtual void f(int n, Point p) { ... }
 virtual int g(Point p) const { ... }
};
class B : public A {
 void f(int n, Point p) override { ... }
 int g(Point p) const final { ... }
};
```

## Which version is selected?

If more than one **overloaded** function or method matches, the best (most specific) is chosen:

```
class pet {};
class cat : public pet {};

void wash(pet &x) { ... }
void wash(cat &x) { ... }

int main() {
 cat felix;
 wash(felix); // both functions match; second is used
}
```

Overload: **STATIC** (i.e., compile-time) decision

Override: **DYNAMIC** (i.e., run-time) decision

## Overloading – Write fewer **if**'s with OOP!

Overloading – STATIC/COMPILATION TIME:

```
void f(pet & x) {
 if (x isA cat) {}
 else if (x isA dog) {}
 else if (x isA hamster) {}
 else {assert(0);} /*ERROR*/
}

void f(cat &x) {...}
void f(dog &x) {...}
void f(hamster &x) {...}
/*NO* (runtime) *ERROR*!!!
```

Overriding – DYNAMIC/RUN TIME:

```
void move(person &p) {
 if (p isA driver) {}
 else if (p isA cyclist) {}
 else if (p isA pilot) {}
 else { /*DEFAULT* } }

class person { /*DEFAULT*
 virtual void move() {...} }
class driver :person{
 void move(){...} }
class cyclist :person{
 void move(){...} }
class pilot :person{
 void move(){...} }
```

Write better if/then/else's – let the compiler do it!

## Pointers and subtyping

Pointers to derived classes are subtypes of pointers to base classes (i.e., if I can point to a base class, I can also point to a derived class):

```
cat felix;
pet *p = &felix;
```

No slicing occurs here, because pointers are copied not objects (a memory address is the same size as another memory address):

```
p->Speak(); // miao
```

The **speak** method uses the virtual method **sound**, which is defined in the **cat** class, and selected by dynamic binding (see slides 6–13).

## Containers of pointers

Often a container holds pointers to a base type:

```
vector<pet *> pets;
cat felix("Felix");
dog fido("Fido");
pets.push_back(&felix);
pets.push_back(&fido);
```

When we access elements of the vector, dynamic binding is used:

```
for (std::size_t i = 0; i < pets.size(); ++i)
 pets[i]->Speak(); // miao, woof
```

## Introducing dynamic allocation

- Typically the number of things in the collection is unpredictable
- So allocate objects dynamically (as in Java) on the heap:

```
cat *cp = new cat("tiddles");
pets.push_back(cp);
```

Here the pointer `cp` is local, but the object it points at is on the heap (so it outlasts the current block)

- Major difference: in C++ the programmer is responsible for deallocation, but we'll ignore that till session 8

- Better (C++11):

```
#include <memory>
vector<shared_ptr<pet>> pets;
// shared_ptr<cat> cp = make_shared<cat>("Tom");//Old
auto cp = make_shared<cat>("Tom");//New, simpler!!!
pets.push_back(cp);
```

## Templates and subtyping (I)

When `cat` is a subtype of `pet`,

- `cat *` **IS** a subtype of `pet *`, but
- `vector<cat *>` **IS NOT** a subtype of `vector<pet *>`!

Why not? Consider this code fragment:

```
vector<cat *> cats;
vector<pet *> *p = &cats; // illegal
dog fido;
p->push_back(&fido); // would be trouble
```

See Stroustrup 13.6.3(3<sup>rd</sup> ed.)/27.2.1(4<sup>th</sup> ed.) for more.

## Templates and subtyping (II)

- It is possible to inherit from a template class

```
template <typename T>
class history { ... };
```

```
template <typename T>
class my_history : public history<T> { ... };
```

- The parameters need not be the same

```
class browser_history : history<string> { ... };
```

```
template <typename T>
class pointer_history : history<T *> { ... };
```

Why not

```
template <typename T>
class browser_history : history<string> { ... };
```

???

Because `browser_history` is NOT a template class, it simply inherits from a (specialised) template class.

## Next session: multiple inheritance

- In many object-oriented languages, including C++ and Eiffel, a class may derive from more than one base class.
- Java supports a common special case: a class may extend only one class, but may implement any number of interfaces.
- Multiple inheritance is very useful, but raises the question of what to do when the base classes conflict.
- Reading: Stroustrup 15.2

## Final Notes – I

- Inheritance is used for:
  - ① Code re-use (**bad, bad, bad!** That's why Java allows us to inherit from at most one class)
  - ② Defining type-hierarchies through the IsA relation between types: car Is-A vehicle, cat Is-A pet (**good, good, good!** That's why Java allows us to inherit from as many interfaces as we want)
- Inheritance is required if we need **\*dynamic binding\***, *i.e.*, code that behaves differently at run-time depending on the real type of the objects involved.
  - For dynamic binding we also need to use references or pointers (they keep the real type of the objects and don't cause slicing to happen).
  - And of course we need some member functions to be virtual, otherwise the compiler will plug-in direct calls to the superclass member functions (static binding) instead of checking the object's real type and using dynamic binding.
- Slicing: If we try to assign an object of a derived class (like holiday) into an object of a base class (like date), then there's not enough room for all the information, so we need to slice the object of the derived class - we throw away its new members and keep just the members of the base class.
- We can initialize the base class part of a derived object by calling the constructor of the base class in the initialization list of the derived object's constructor (only there can we call it):
 

```
holiday(string n, int d, int m) : date(d, m), name(n) {}
```
- Initialization order:
  - ① Constructors of base classes
  - ② Constructors of members
  - ③ Body of constructor of the derived class

Principle: **The constructor body needs a fully initialised object!**

The destruction follows the **opposite** order (destructor body, destructors of members, destructors of base classes).

Principle: **The destructor body needs a fully initialised object!** (same principle)

- Overriding behaviour: The base class must have declared the member function as virtual for us to be able to override it in the derived class:
 

```
virtual string desc() const {...}
```
- Pure virtual member functions (aka abstract methods):
 

```
virtual string sound() const = 0; // no code!
```

  - Virtual functions are essentially pointers (to functions).
  - Pure virtual (abstract) functions are null (**nullptr**) pointers (no code to point to). That should explain the bizarre syntax (= 0).
  - A class with at least one pure virtual member function is an abstract class - cannot instantiate it (but we can have references and pointers to it – for dynamic binding, see below).
  - A class with no members (fields) and all of its member functions pure virtual is equivalent to a Java interface.
  - If your class has a virtual function then it probably needs a virtual destructor.

## Final Notes – II

- Static vs Dynamic binding - check out slide 12.
  - Function **print\_day1** uses call-by-value (so the real object passed is copied and sliced in order to initialize the local parameter and the function always operates on a date object).
  - Function **print\_day2** uses call-by-reference (so the real object is passed without copying/slicing, initializing the local reference parameter to refer to it whatever it may be, and the function operates on any kind of date object).
  - To get dynamic binding, *i.e.*, different behaviour at runtime depending on the real type of an object, one needs two things:
    - To have virtual member functions, which have been overridden in derived classes (the implementation of the different behaviour according to the type of the object)
    - To allow these virtual member functions to be selected dynamically at runtime, by passing objects either by reference or by pointer. Otherwise (*i.e.*, in pass-by-value) static binding is used.

- Java has **super (...)**; to call the same method in the parent class. A C++ class may have multiple parent (base) classes, so to call one of their member functions that we've overridden, we must name the base class explicitly:

```
class dog : public pet {
 void speak() const override
 /* "override" - C++11 keyword to show that we want
 to override some base class' speak */ {
 pet::speak(); // call pet's speak
 cout << ' ' << _name << " wags tail\n";
 }
};
```

- Containers of pointers:
  - Want to have a collection of objects but your class doesn't have a default constructor?
  - Want to avoid copying objects around?
  - Want to store different sub-types of some base class and get dynamic binding when you use them (and avoid slicing them)?

Then use a container of pointers – slide 20.

- Beware that **vector< cat \* >** isn't a sub-type of **vector< pet \* >**, even though **cat \*** is a sub-type of **pet \*** when **cat** is a sub-type of **pet** (slides 22–23).
- Inheritance and templates: slides 22–23. Partial specialization (**PointerHistory** partially specializes the type of **History** to be a pointer to some still unknown type **T**).
  - More on template specialization (and partial specialization) [www.cprogramming.com/tutorial/template\\_specialization.html](http://www.cprogramming.com/tutorial/template_specialization.html)
  - Did you notice in the template specialization article that a template parameter does not have to be a typename? Welcome to Template Meta-Programming [www.codeproject.com/Articles/3743/A-gentle-introduction-to-Template-Metaprogramming](http://www.codeproject.com/Articles/3743/A-gentle-introduction-to-Template-Metaprogramming) No need to thank me. (DEFINELY \*NOT\* IN THE SCOPE OF THE MODULE/EXAM!)

And some interesting further reading that may help you better understand how virtual member functions work (and don't work sometimes) – not part of the exam but **highly** helpful:

- Vee Table <https://wiki.c2.com/?VeeTable>
- Fragile Binary Interface Problem <https://c2.com/cgi/wiki?FragileBinaryInterfaceProblem>



**Composition & Conversion instead of Inheritance**

```

#include <iostream>
#include <string>
using namespace std;

class date {
 int day, month, year;
public:
 date(int d = 1, int m = 1, int y = 1) : day(d), month(m), year(y) {}
 int get_day() const { return day; }
 int get_month() const { return month; }
 int get_year() const { return year; }
};

// class holiday : public date {
class holiday {
 date d;
 string name;
public:
 holiday(string n) : d(), name(n) {}
 holiday(string n, int day, int mon) :
 d(day, mon), name(n) {}
 string get_name() const { return name; }
 operator date() const {return d;}
};

void print_month_passbyvalue(date d) {
 cout << d.get_month() << endl;
}

void print_month_passbyreference(date & d) {
 cout << d.get_month() << endl;
}

void print_month_passbyconstreference(const date & d) {
 cout << d.get_month() << endl;
}

int main() {
 holiday xmas("Xmas", 25, 12);

 print_month_passbyvalue(xmas);

 // print_month_passbyreference(xmas); // ERROR - why?

 print_month_passbyconstreference(xmas);

 // cout << xmas.get_month() << endl; // ERROR - why?

 cout << ((date) xmas).get_month() << endl;

 return 0;
}

```

## Programming in C++

### Session 7 – Multiple Inheritance

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

## Major Differences between Java and C++

*These are the main pain points to understand C++ [★]  
(why did Java “simplify” them?)*

- **call-by-reference** (session 1 and since)
- operator overloading (session 3)
- genericity or template classes (sessions 4–6)
- **memory management**
  - local allocation of objects (sessions 1–2 and since)
  - **pointers** (sessions 5–6)
  - dynamic allocation & **de-allocation** (sessions 8–9)
- **multiple inheritance** (this session)

[★] (*and to answer in job interviews*)

## Multiple Inheritance

- In many object-oriented languages, including C++ and Eiffel, a class may derive from more than one base class.
- Java supports a common special case: a class may extend only one class, but may implement any number of interfaces.
- Multiple inheritance is very useful, but raises some questions:
  - What if both happen to **define** the same names?
  - What if both **derive** from a common class?

Both these are **implementation** (code reuse) problems, nothing to do with the type hierarchies.

**That's why interfaces (i.e., abstract classes) don't have problems...**

## The simple case

- A common, simple, use of multiple inheritance to combine two essentially unrelated classes:  
class read\_write : public reader, public writer { **An IS-A relation.**  
...  
};
- We can also combine classes using sub-objects:  
class chess\_game : public window { **An IS-A relation.**  
protected:  
board board; **A HAS-A relation.**  
...  
};

**Key question:** should the new class be usable by clients of the old?  
That is, do we need an IS-A relation (yes) or a HAS-A relation (no) ?  
This question is about the type relation – nothing to do with code reuse.

## An asymmetrical case

Often a class extends a concrete base class and an abstract one, using the concrete class to implement the undefined methods from the second class, and possibly a bit more:

```
class active_grid : public grid,
 public button_listener {
public:
 void mouse_pressed(button_event & e) {
 // use grid stuff
 }
};
```

Java supports only this special case.

## Name clashes (ambiguity)

What if two base classes define the same name?

```
class A { public: int f(); };

class B { public: int f(); };

class AB : public A, public B {
public:
 int g() {
 return f() + 1; // which one?
 }
};
```

## Possible solutions

- The language chooses one, using some rule (some LISP dialects).
- The language permits the programmer to rename methods of a base class in a derived class, thus avoiding the clash (Eiffel).
- The programmer must explicitly qualify the names with the class from which they come (C++).

Renaming the methods in the original classes is often not an option, as they may be part of a library or fixed interface.

## Ambiguity resolution by qualification

In C++, ambiguous names must be qualified: [t.ly/c90gP](https://t.ly/c90gP) QUIZ now!

```
class A { public: int f() {return 1;} };
class B { public: int f() {return 2;} };
class AB : public A, public B {
public:
 int f() { return 3; }
 int g() {
 return A::f() + B::f() + f() + 1; // 7
 }
};

void fa(A &a){ cout << a.f() << endl; }
void fb(B &b){ cout << b.f() << endl; }
...
AB ab;
fa(ab); // prints what? why?
fb(ab); // prints what? why?
```

## └─ Ambiguity resolution by qualification

**Ambiguity resolution by qualification**  
 In C++, ambiguous names must be qualified to help the compiler resolve the ambiguity.

```

class A { public: int f(); virtual ~A(); };
class B { public: int f(); virtual ~B(); };
class AB { public: A, B; };

public:
 int f() { return A.f(); }
 int f() { return B.f(); }
};

void f(A a) { cout << a.f() << endl; }
void f(B b) { cout << b.f() << endl; }

int main() {
 AB ab;
 f(ab); // prints what? what?
}

```

- Will print 1 & 2 respectively, because **f** is **NOT** virtual!
- So there's no dynamic binding – compiler chooses the appropriate **f** statically (at compilation time), by considering the interface of the object.
- If **A's f()** was **virtual**, then **fa()** would print 3 if its argument was of class **AB**...
- What if **f** was **virtual only** inside class **A**?

## Replicated base classes

```
class storable { int width; ... }; // I HAVE *width*!
```

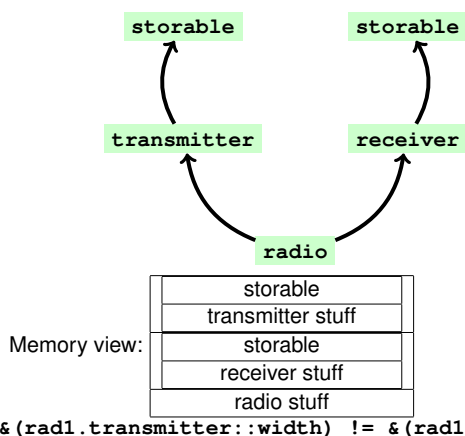
```
class transmitter : public storable { ... };
```

```
class receiver : public storable { ... };
```

```
class radio : public transmitter,
 public receiver { ... };
```

- A **radio** object will contain *two* distinct **storable** components, and thus two versions of each member.
- All references to **storable** members in **radio** must be qualified with either **transmitter** or **receiver**.

## Replicated base classes, graphically



```
&(rad1.transmitter::width) != &(rad1.receiver::width)
```

## Virtual functions in the base class

```

class storable {
public:
 virtual void write() = 0;
};

class transmitter : public storable {
public:
 virtual void write() { ... }
};

class receiver : public storable {
public:
 virtual void write() { ... }
};

```

## Overriding virtual methods

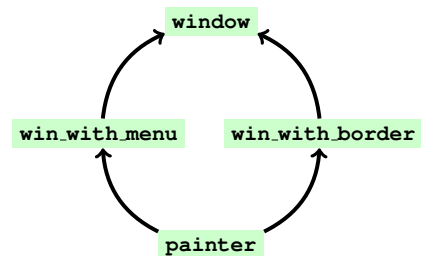
A virtual function in the replicated base class can be overridden:

```
class radio : public transmitter,
 public receiver {
public:
 virtual void write() {
 transmitter::write();
 receiver::write();
 // write extra radio stuff
 }
};
```

The use of the base class versions, plus a bit more, is common.

## Virtual inheritance (sharing)

Suppose we want:



## Virtual base class

If we write

```
class window { ... };

class win_with_border : public virtual window {...};
class win_with_menu : public virtual window {...};

class painter : public win_with_border,
 public win_with_menu { ... };
```

Then a `painter` object includes a *single* `window`.

Class `window` is a **virtual base class** of class `painter`.

- Virtual method – you have a **pointer** to the method.
- Pure virtual method (= 0) means a `nullptr` pointer (no code)

⇒ **Virtual base class – you have a pointer to it!**

(just like “virtual memory” in OSs uses indirection to real memory)

## Virtual inheritance – Memory view

|                       |
|-----------------------|
| shared window w       |
| window *wptr = &w     |
| win_with_menu stuff   |
| window *wptr = &w     |
| win_with_border stuff |
| painter stuff         |

## Constructors

```
class window {
public:
 window(int i) { ... }
};
class win_with_border : public virtual window {
public:
 win_with_border() : window(1) { ... }
};
class win_with_menu : public virtual window {
public:
 win_with_menu() : window(2) { ... }
};
```

**PROBLEM:** The base classes of `painter` want to initialise the common `window` object in a different way – they don't know it's shared!

**SOLUTION:** Ignore them – class `painter` is the one best placed to decide how the common `window` object should be initialised.

## Constructors for a virtual base class

The class in the hierarchy that *knows* that a common virtual base class is shared decides how to construct it (intermediate classes don't know if the virtual base class is shared or not).

```
class painter : public win_with_border,
 public win_with_menu {
public:
 painter(int i) : window(i),
 win_with_border(),
 win_with_menu() { ... }
 ...
};
```

- Avoids conflicts between intermediate class constructors
- Language ensures each constructor is called exactly once

## Ensuring other methods are called only once

When the virtual base class has a method redefined by each class?

```
class window {
public:
 virtual void draw() {
 // draw window
 }
};
```

## Drawing, first attempt

```
class win_with_border : public virtual window {
public:
 virtual void draw() {
 window::draw();
 // draw border
 }
};

class win_with_menu : public virtual window {
public:
 virtual void draw() {
 window::draw();
 // draw menu
 }
};
```

## Disaster!!!

But then if we write:

```
class painter : public win_with_border,
 public win_with_menu {
 void draw() {
 win_with_border::draw();
 win_with_menu::draw();
 // draw painter stuff
 }
};
```

The **window** gets drawn twice!

## Solution: auxiliary methods

We put the drawing of the extra stuff in a method of its own:

```
class win_with_border : public virtual window {
protected:
 void own_draw() { ... }
public:
 virtual void draw() {
 window::draw();
 own_draw();
 }
};
```

And similarly for **win\_with\_menu**.

## Calling each method once

```
class painter : public win_with_border,
 public win_with_menu {
protected:
 void own_draw();
public:
 void draw() {
 window::draw();
 win_with_border::own_draw();
 win_with_menu::own_draw();
 own_draw();
 }
};
```

Then each part is drawn exactly once.

## Virtual inheritance: Summary

- *Good news*: Virtual inheritance is a rare case.
- *Even better*: Language ensures constructors called exactly once.
- *Bad: Code Re-use Kills*  
If a method is defined in the virtual base class and overridden in more than one derived class (a rare case), considerable care is required to ensure that each method is called exactly once.
- If a method is pure virtual in the virtual base class, the issue does not arise (because the derived versions cannot call it).
- If the method is overridden in only one branch, the issue does not arise (because only that version need be called).

## I/O stream classes

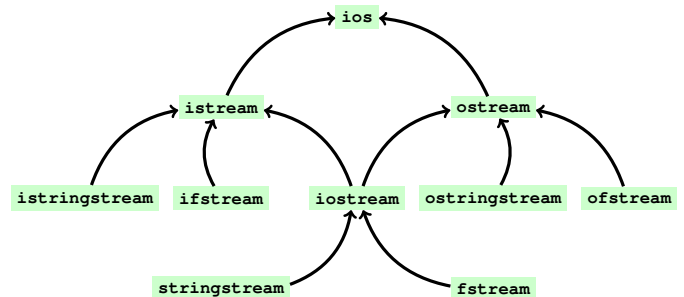
```
class ios {
 // private state
public:
 bool good() const { ... }
 bool eof() const { ... }
 bool fail() const { ... }
 bool bad() const { ... }
};

class istream : virtual public ios { ... };

class ostream : virtual public ios { ... };

class iostream : public istream, public ostream {};
```

## Stream class hierarchy



The state of `ios` is *not* duplicated.

## Next session: memory management

- Both Java and C++ have dynamic/heap allocation (**new**), but
  - In Java, heap objects are automatically recycled when no longer needed.
  - In C++, this is the programmer's responsibility.
- In C++, we can have these kinds of bugs (Java only #1):
  - Freeing too late: overusing memory
  - Forgetting to free: memory leak
  - Freeing things twice: mysterious program crashes
  - (And freeing things prematurely...)
  - (And freeing things the wrong way...)
  - (And freeing things that were not created with **new**...)
- Alternative strategies:
  - Use local allocation instead (not always appropriate).
  - Use C++11 smart pointers:
    - `unique_ptr<T>`, `shared_ptr<T>`, `weak_ptr<T>`
- Reading: Stroustrup section 10.4, Savitch 10.3, Horstmann 13.2.

## Programming in C++

2024-11-22

Next session: memory management

### Final Notes – I:

- Multiple inheritance is a major difference between Java and C++.
- Java doesn't allow it – inheriting fields and code from multiple classes is problematic:
  - What if multiple parent classes define the same fields or functions?
  - What if multiple parent classes inherit from a common class themselves?
- Both these problems are caused by code reuse, not by introducing a type hierarchy.
- That's why in Java you can inherit from only one class and ... multiple interfaces (that don't have any code).
- C++ allows multiple inheritance – it gives you all the tools you need to solve the issues (enough rope to hang yourself...).
- Sometimes you can avoid inheritance altogether – the key question to ask is:
  - Should class A be usable at all settings where class B is usable? If so, then A should inherit from B (A is-A B). Otherwise A can simply contain a B (A Has-A B).
- Name ambiguity is resolved by qualification:
  - `ClassName::MemberName()`

**Need operation: memory management**

- Both Java and C++ have dynamic heap allocation (new), but
  - In Java, heap objects are automatically recycled when no longer needed.
  - In C++, this is the programmer's responsibility.
- In C++, we can have these kinds of bugs (Java only #1):
  - Freeing too late: overusing memory
  - Freeing too late: memory leak
  - Freeing things twice: mysterious program crashes
  - (And freeing things prematurely...)
  - (And freeing things the wrong way...)
  - (And freeing things that were not created with new...)
- Alternative strategies:
  - Use local allocation instead (not always appropriate).
  - Use C++11 smart pointers:
    - `unique_ptr<T>`, `shared_ptr<T>`, `weak_ptr<T>`

Reading: Stroustrup section 10.4, Savitch 10.3, Horstmann 13.2.



Next session: memory management

**Need operator: memory management**

- Both Java and C++ have dynamic heap allocation (new, but
  - In Java, heap objects are automatically recycled when no longer needed.
- In C++, it's the programmer's responsibility
  - In C++, we can have three kinds of bugs (Java only #!):
    - Pointing too low: accessing memory
    - Pointing to free: memory leak
    - Pointing things twice: mysterious program crashes
  - And having things that were not created with new...
  - And having things that were not created with new...
- Alternative strategies
  - Use heap allocation instead (not always appropriate)
  - Use C++ smart pointers (shared\_ptr, weak\_ptr, atomic\_ptr)

★ Reading: Stroustrup section 10.4, Section 10.5, Herlihy section 10.2

## Final Notes – II:

- Two types of multiple inheritance:

- Replicated inheritance:

```
#include <cassert>
// struct's a class with everything public.
struct A {int x;};
class B: public A {};
class C: public A {};
class D: public B, public C {};
int main() {
 D d1;
 d1.B::x = 1; // assign d1's x from the B side
 d1.C::x = 2; // assign d1's x from the C side
 assert(&(d1.B::x) != &(d1.C::x));
 assert(d1.B::x == 1);
 assert(d1.C::x == 2);
 // restricted view of d1 - B interface (B & ...)
 B & b_view_of_d1 = d1;
 // restricted view of d1 - C interface (C & ...)
 C & c_view_of_d1 = d1;
 assert(&(b_view_of_d1.x) != &(c_view_of_d1.x));
 assert(b_view_of_d1.x == 1);
 assert(c_view_of_d1.x == 2);
 /* ALWAYS */
 assert(&(d1.B::x) == &(b_view_of_d1.x));
 assert(&(d1.C::x) == &(c_view_of_d1.x));
 return c_view_of_d1.x - b_view_of_d1.x; // 1
}
```

D contains two copies of A – one from the B side and one from the C side (like persons having two grandfathers – one from their mother's side and one from their father's side).

- Virtual inheritance:

```
#include <cassert>
struct A {int x;};
class B: virtual public A {}; // virtual public =
class C: public virtual A {}; // public virtual
class D: public B, public C {};
int main() {
 D d1;
 d1.B::x = 1;
 assert(&(d1.B::x) == &(d1.C::x) && d1.B::x == 1);
 d1.C::x = 2;
 assert(&(d1.B::x) == &(d1.C::x) && d1.B::x == 2);
 B &b_view_of_d1 = d1;
 C &c_view_of_d1 = d1;
 assert(&(b_view_of_d1.x) == &(c_view_of_d1.x));
 assert(b_view_of_d1.x == 2);
 /* ALWAYS */
 assert(&(d1.B::x) == &(b_view_of_d1.x));
 assert(&(d1.C::x) == &(c_view_of_d1.x));
 return c_view_of_d1.x - b_view_of_d1.x; // 0
}
```

D contains only one copy of A – the B and C side have virtual A's.

- Compiler ensures constructors work as expected (only called once).
- You need auxiliary methods to get this version of inheritance work for other methods.

Next session: memory management

**Need operator: memory management**

- Both Java and C++ have dynamic heap allocation (new, but
  - In Java, heap objects are automatically recycled when no longer needed.
- In C++, it's the programmer's responsibility
  - In C++, we can have three kinds of bugs (Java only #!):
    - Pointing too low: accessing memory
    - Pointing to free: memory leak
    - Pointing things twice: mysterious program crashes
  - And having things that were not created with new...
  - And having things that were not created with new...
- Alternative strategies
  - Use heap allocation instead (not always appropriate)
  - Use C++ smart pointers (shared\_ptr, weak\_ptr, atomic\_ptr)

★ Reading: Stroustrup section 10.4, Section 10.5, Herlihy section 10.2

## Checking if two objects are the same

As we saw in slide 10, to check if two objects are the same (and not just equal to each other) we need to check their addresses in memory. Here's a simple example of that using int:

```
#include <cassert>

int main() {

 int i = 3, j = 3;

 assert(i == j); // same values

 assert(&i != &j); // BUT different objects
 // (cause different addresses!)

 return 0;
}
```

Comparing addresses is closer to what the === operator does in languages like JavaScript, which is (mainly) used to check if two objects are the same ([developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict\\_equality](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_equality)). In Java, the == operator when applied to Object references compares the references directly (i.e., the pointers), so can be used to figure out if two objects are the same. In Java we need to call equals() to compare object contents instead.

```
public class SameObj {
 public int i = 1;
 public boolean equals(SameObj other) {
 return i == other.i;
 }

 public static void main(String[] args) {
 SameObj o1 = new SameObj(), o2 = new SameObj();
 // Comparing VALUES: .equals()
 if (o1.equals(o2))
 System.out.println("o1 & o2 are equal");
 else
 System.out.println("o1 & o2 are *not* equal");
 // Comparing POINTERS: ==
 if (o1 == o2)
 System.out.println("o1 & o2 are the same");
 else
 System.out.println("o1 & o2 are *not* the same"); /**/
 if (o1 == o1) // should always be true
 System.out.println("o1 & o1 are the same"); /**/
 else
 System.out.println("o1 & o1 are *not* the same");
 }
}
```

## Programming in C++

### Session 8 – Memory Management

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

## The issues

Programs manipulate data, which must be stored somewhere.

- How is the storage allocated?
- How is this storage initialized?
- Can the storage be reused when no longer required?
  - If so, how?
- What is required of the programmer?

## The issues – Java keeps things simple...

Programs manipulate data, which must be stored somewhere.

- How is the storage allocated?  
*On the heap, with **new***
- How is this storage initialized?  
*With constructors – basic types to 0 by default*
- Can the storage be reused when no longer required?  
*Sure*
  - If so, how?  
*With **new***
- What is required of the programmer?  
*Erm... to call **new**?!? 🤖*

Java: Peace! 😊

C++: *I don't want peace – I want problems, always!* 🤖

## Common storage modes

(This is different from *scope*, which is a compile-time attribute of identifiers.)

- static** exists for the duration of program execution.
- local (or stack-based)** exists from entry of a block or function until its exit.
- free (or dynamic, or heap-based)** explicitly created, and either
  - explicitly destroyed, or
  - automatically destroyed when no longer in use.
- temporary** for intermediate values in expressions.

## Static storage in C++

- variables declared outside any class or function. 🤔🤔
- **static** class members. 🤔🤔
- **static** variables in functions. 🤔🤔

(don't use **static** elsewhere – it's something **completely** different [\*)

Variables may be initialized when defined:

```
// global variables
int i; // implicitly initialised to 0
int *p; // implicitly initialised to 0 = nullptr
int area = 500;
double side = sqrt(area);
double *ptr = &side;
int f(int i) {
 static std::size_t times_called = 0;
 return ++times_called;
}
```

[\*) internal linkage [en.cppreference.com/w/cpp/language/storage\\_duration](https://en.cppreference.com/w/cpp/language/storage_duration)

## Implicit initialization of static variables

Static variables that are not explicitly initialized are implicitly initialized to 0 converted to the type.

```
int i;
bool b;
double x;
char *p;
```

is equivalent to

```
int i = 0;
bool b = false;
double x = 0.0;
char *p = 0; // null pointer
```

## Evaluation

Static storage is

**simple** No extra effort from the programmer

**safe** Storage guaranteed

**inflexible** Must determine limits at compile-time

**wasteful** Often allocate more than needed (and then, run out...)

Also, storage held for the entire execution, used or not

Static variables allocated even if not used



**Global/static variables are thread unsafe!**

## Local storage in C++

```
int f(std::size_t start, std::size_t size) {
 int total = 0;
 int tmp;
 for (std::size_t i = start; i < size; ++i) { ... }
}
```

- Formal function parameters: initialized from the arguments
- Function/block local variables  
Uninitialized variable values are undefined
- Variables introduced in **for** loops

## Evaluation

Local storage is

- efficient** The implementation merely adjusts a **stack pointer**
- often suitable** If the data is being used in a block-structured way.
- not enough** What if we wish to construct some data in a function and return it to the caller?

```
int foo() { int i = 3; return i; } // OK
int &bar() { int i = 3; return i; } // KO!
#include <iostream>
using namespace std;
int main() {
 cout << "foo() returns " << foo() << endl;
 cout << "bar() returns " << bar() << endl;
 return 0;
}
```

*Hey – what's a “stack pointer”?*

## Caller creates, passes by reference?

Can't the caller create the object and pass it to us by reference? 🤔

UML calls this an **out** parameter type

Possible if the size of the object is known to the caller  
But if size depends on another parameter (*e.g.*, array of length  $f(N)$ , list/tree of  $g(N)$  nodes, *etc.*), then it doesn't work...

We need more flexibility!

## Free storage in C++

Class types:

```
{
 point *p; // uninitialized pointer
 p = new point; // default constructor
 p = new point(1,3);
 cout << p->x << ' ' << p->y << '\n';
 delete p;
}
```

and similarly for primitive types.

- Created with “**new** type”.
- Programmer's responsibility to **delete** the storage.
- Attempts to access the storage after deletion are potentially disastrous, but not checked by the language.

*Houston, we've had a problem here...*

## Free storage in C++ – II

```
{
 auto p = make_unique<point>(); // default constructor
 p = make_unique<point>(1,3);
 cout << p->x << ' ' << p->y << '\n';
}
```

*Houston, never mind...*

## Dynamically allocated arrays in C++

Pointers can also address dynamically allocated arrays

```
{
 int *arr;
 arr = new int[n];
 for (size_t i = 0; i < n; ++i) arr[i]=f(i) + 3;
 delete[] arr;
}
```

**Note** Special deletion syntax!

Cause C++ doesn't distinguish pointers to an `int`/array of `ints` 🧑

```
{ // safe:
 auto arr = make_unique<int[]>(n);
 for (size_t i = 0; i < n; ++i) arr[i]=f(i) + 3;
}
```

## Destructors

A class `C` may include a destructor `~C()`, to release any resources (including storage) used by the object.

```
class C {
 date *today;
 int *arr;
public:
 C() : today(new date()), arr(new int[50]) {}

 virtual ~C() { delete today; delete[] arr; }
};
```

Destruction: opposite order to construction!

(same principle: destructor body needs to have a valid object)

(NOT exception safe code – check notes!)

### Exception Safety

The constructor of class `C` is not exception safe...

What will happen if the first `new` succeeds but the second one throws an exception?

Then the object is not initialised – its destructor will not run and the memory allocated by the first `new` will not be reclaimed (a memory leak).

To make it exception-safe we'd need to use smart pointers:

```
#include <memory>
#include <utility>
using namespace std;
class C {
 unique_ptr<pair<float,float>> upair; // prefer unique_ptr
 shared_ptr<pair<float,float>> spair; // over shared_ptr
 unique_ptr<float[]> uarr; // unique_ptr supports arrays
 // as well in C++11/14 - shared_ptr only in C++17
public:
 C() : upair(make_unique<pair<float,float>>(1.1, 2.2)),
 spair(make_shared<pair<float,float>>(3.3, 4.4)),
 uarr(make_unique<float[]>(50)) {}

 virtual ~C() {}
};
int main() {
 C c1;
 return 0;
}
```

**Destructors**  
A class `C` may include a destructor `~C()`, to release any resources (including storage) used by the object.

```
class C {
 date *today;
 int *arr;
public:
 C() : today(new date()), arr(new int[50]) {}
 virtual ~C() { delete today; delete[] arr; }
};
```

Destruction: opposite order to construction!  
(same principle: destructor body needs to have a valid object)  
(NOT exception safe code – check notes!)

## Why **virtual**? Dynamic Binding!

Suppose `car` is a derived class of `vehicle` and consider the following code fragment:

```
vehicle *p = new car;
...
delete p;
```

- The destructor `~car()` will not be called unless `vehicle`'s destructor is **virtual**.
- So why aren't destructors **virtual** by default?
- Because that would be a little less efficient...

Virtual needed even if used with smart pointers

Why **virtual**? Dynamic Binding!Why **virtual**? Dynamic Binding!

Suppose `car` is a derived class of `vehicle` and consider the following code fragment:

```
vehicle *p = new car;
delete p;
```

- The destructor `~car()` will not be called unless `vehicle`'s destructor is `virtual`.
- So why would destructors `virtual` by default?
- Because that would be a little less efficient...

Virtual needed even if used with smart pointers

## ATTENTION!!!

- Always make the destructor **virtual** if there's a chance that the class will serve as a base class.
- When there's a **virtual** member function then it's certain that the class will serve as a base class at some point – make the destructor **virtual** as well!!!
- **virtual** is needed even if your fields are smart pointers. If your class will be inherited from, then the constructor *MUST* be **virtual**, no matter what.
- **virtual** `~C() {}` is enough.
- Even better: **virtual** `~C() = default;`  
(if using defaults, state so!)

## Construction and destruction

|                                                                     | Storage allocated, constructor initializes it                                                          | Destructor is called, storage is reclaimed                                                        |
|---------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <b>static object</b>                                                | <b>before</b> main starts                                                                              | <b>after</b> main terminates                                                                      |
| <b>local object</b>                                                 | when the declaration is executed                                                                       | on exit from the function or block                                                                |
| <b>free object</b>                                                  | when <b>new</b> is called                                                                              | when <b>delete</b> is called                                                                      |
| <b>subobject</b> <span style="background-color: yellow;">[*]</span> | when the containing object is created (constructed <b>before</b> the containing object is constructed) | when the containing object is destroyed (deleted <b>after</b> the containing object is destroyed) |

## [\*] Principle:

The constructor/destructor body needs to deal with a valid object.

## Example: a simple string class

```
#include <cstring>
class my_string {
 std::size_t len; // BUG IF YOU CHANGE THE ORDER!!!
 char *chars;
public:
 my_string(const char *s)
 : len(1+std::strlen(s)), chars(new char[len]) {
 for (std::size_t i=0; i<len; ++i) chars[i] = s[i];
 }
 // more to come later ...
};

Better:
my_string(const char *s):len(1+strlen(s)), chars(0){
 chars = new char[len]; // "len" exists here for sure
 for (std::size_t i=0; i<len; ++i) chars[i]=s[i];
}
```

## Default constructor

We also have a default constructor making an empty string:

```
class my_string {
 std::size_t len;
 char *chars;

public:
 my_string() : len(1), chars(new char[1])
 {chars[0] = '\0';}

 // ...

 virtual ~my_string() { delete[] chars; }
};
```

**Why the new char [ 1 ] ?**  
**Why not new char ?**  
**Why not nullptr ?**

## └ Default constructor

**Default constructor**  
 We also have a default constructor making an empty string:

```
class my_string {
 int len = 0;
 char* chars;

public:
 my_string() : len(0), chars(new char[1])
 (chars[0] = '\0') {
 // ...
 }
 virtual ~my_string() { delete[] chars; }
};
```

*Why the new char? ( 0, 1, 2  
 Why not new char? ?  
 Why not new char? ?*

## Why?

## CLASS INVARIANT: “chars points to an array of size len”

- Therefore, **chars** cannot be initialised with **new char** since then it'll not be pointing to an **ARRAY** of characters – we will not be able to do **delete [] chars**; in that case.
- I can do **delete [] nullptr**; – that works fine (does nothing, just like **delete nullptr**;  
 But I'd be breaking the invariant, since **chars** would not be pointing to an array of length **len**...

The importance of the class invariant – if you don't know the invariant, your code is wrong (no ifs, not buts...)

## Initialization of objects

- Initialization is not assignment: target is empty
- Initialization calls some constructor, e.g.,  

```
my_string foo = "bar";
```

 calls the constructor **my\_string(char\*)**
- Initialization from another **my\_string** object calls the **copy constructor**  

```
my_string(const my_string &other);
```
- If no copy constructor supplied,  
 compiler generates a memberwise copying one  
*This may not always be the right thing...*

Here:

```
my_string(const my_string &other)
: len(other.len), chars(other.chars) { }
```

*But this copy constructor is PrObLeMaTiC...*

## A problem

Here are some initializations:

```
{
 my_string empty;
 my_string s1("blah blah");
 my_string s2(s1); // initialized from s1
 my_string s3 = s1; // initialized from s1
} // all four strings are destroyed here
```

- After last initialization, **s1**, **s2** & **s3** all point at same array
- The array will be deleted **three times!**

*(Bad, bad karma...)*

## Solution: define a copy constructor

So define a copy constructor to copy the character array:

```
my_string(const my_string &other)
: len(other.len),
 chars(new char[other.len]) { //other.len, NOT len!
 for (std::size_t i = 0; i < len; ++i)
 chars[i] = other.chars[i];
}
```

- This copying (“*deep copy*”) is typical:  
 With explicit deallocation, generally unsafe to share
- In this case, Java is *more* efficient

## Assignment

- Assignment (=) isn't initialization: target already has data
- Each type **overloads** the assignment operator
- For `my_string` it's a member function with signature  
`my_string & operator= (const my_string &other);`
- If no assignment operator supplied, compiler generates a memberwise copying one  
`my_string & operator= (const my_string &other) {  
 len = other.len;  
 chars = other.chars;  
 return *this; // <---- enable chaining!!!  
}  
// chain: a = b = c; (a = (b = c));`

## More problems

Consider

```
{
 my_string s1("blah blah");
 my_string s2("do be do");
 s1 = s2; // assignment
} // the two strings are destroyed here
```

Problems after assignment:

- Original `s1` array discarded but **\*NOT\*** deleted
- Both `s1` & `s2` point at same array, which is deleted **\*TWICE\***



## Solution: define an assignment operator

So define an assignment operator for `my_string`

```
my_string & operator= (const my_string &other) {
 if (&other != this){// DON'T COPY ONTO SELF!!!
 delete[] chars; // I: DESTRUCTOR ACTIONS

 len = other.len; // II: COPY CONSTRUCTOR ACTIONS
 chars = new char[len];
 for (std::size_t i = 0; i < len; ++i)
 chars[i] = other.chars[i];
 }
 return *this; // III: RETURN YOURSELF
}
```

## Programming in C++

2024-11-28

└ Solution: define an assignment operator

### Define an assignment operator – II

So define an assignment operator for `my_string`

```
my_string & operator= (const my_string other)
 // II: COPY CONSTRUCTOR ACTIONS
{
 len = other.len;
 std::swap(chars, other.chars);
 return *this; // III: RETURN YOURSELF
} // I: DESTRUCTOR ACTIONS
```

Solution: define an assignment operator

```
So define an assignment operator for my_string
my_string & operator= (const my_string &other) {
 if (&other == this){// DON'T COPY ONTO SELF!!!
 delete[] chars; // I: DESTRUCTOR ACTIONS

 len = other.len; // II: COPY CONSTRUCTOR ACTIONS
 chars = new char[len];
 for (std::size_t i = 0; i < len; ++i)
 chars[i] = other.chars[i];
 }
 return *this; // III: RETURN YOURSELF
}
```



## The `this` pointer

In C++,

- `this` is a pointer to the current object (as in Java),
- So the “current object” is “`*this`”

```
class ostream {
 ...
public:
 ostream & operator<<(const char *s) {
 for (; *s != '\0'; ++s) // (1)
 *this << *s; // (2)
 return *this;
 }
};
```

(1) Looping over a C string.

(2) What does that line do?

\*\* Why do we destroy our string parameter `s` by doing `++s`!?

## An alternative: forbid copying

If we define a private copy constructor and assignment operator,

```
class my_string {
private:
 my_string (const my_string &s) {}

 my_string & operator= (const my_string &s) {
 return *this; // STILL NEED IT!!!
 }
 ...
}
```

- The compiler will not generate them, but the programmer will not be able to use these ones
- Any attempt to copy strings will result in a compile-time error
- `return *this;` needed to satisfy the function's return type

### └ An alternative: forbid copying

#### C++11

Since C++11 we can write:

```
my_string(const my_string &) = delete;
my_string & operator= (const my_string &s) = delete;
```

Explicitly tell the compiler (and other programmers!) that the copy constructor/assignment operator does not exist and should not be auto-generated.

#### An alternative: forbid copying

If we define a private copy constructor and assignment operator,

```
class my_string {
private:
 my_string (const my_string &s) {}

 my_string & operator= (const my_string &s) {
 return *this; // STILL NEED IT!!!
 }
 ...
}
```

• The compiler will not generate them, but the programmer will not be able to use these ones  
• Any attempt to copy strings will result in a compile-time error  
• `return *this;` needed to satisfy the function's return type

## Summary

### *The Gang of Three*

For each class, the compiler will automatically generate the following member functions, unless the programmer supplies them:

**copy constructor:** memberwise copy

**assignment operator:** memberwise assignment

**destructor:** do nothing (subobjects are destroyed automatically)

- If **\*NO\*** constructor supplied, compiler generates a default constructor: memberwise default initialization
- If defaults not what desired, define functions yourself

## Summary

## C++11

Since C++11, it's the Gang of Five...

## + Move constructor

```
my_string (my_string && o); // no const,
 // && instead of &
```

## + Move assignment operator

```
my_string & operator= (my_string && o);
// no const, && instead of &
```

Compare these with the copy constructor and (copy) assignment operator declarations on the slide to the right (slide 28).

The move versions don't copy the members of the other object – they *move* them (i.e., steal them)!

(more on this at the last lecture)

[https://en.cppreference.com/w/cpp/language/rule\\_of\\_three](https://en.cppreference.com/w/cpp/language/rule_of_three)

## Summary

## The Gang of Five

For each class, the compiler will automatically generate the following member functions, unless the programmer supplies them:

- copy constructor: memberwise copy
- assignment operator: memberwise assignment
- destructor: by calling destructors as destroyed automatically

\* If "NO" constructor supplied, compiler generates a default constructor: memberwise default initialization

\* If defaults not what desired, define functions yourself

## Default Copy Constructor and Assignment Operator

```
XYZ (const XYZ & other)
: field1(other.field1),
 field2(other.field2),
 ...
 fieldN(other.fieldN) {
}
```

```
XYZ & operator= (const XYZ & other) {
 field1 = other.field1;
 field2 = other.field2;
 ...
 fieldN = other.fieldN;

 return *this;
}
```

## Default Default Constructor

```
XYZ ()
: field1(), // if it exists
 field2(), // if it exists
 ... // if it exists
 fieldN() { // if it exists
}
```

Basic types don't have a default constructor, so... you get garbage.

## Summary, continued

- If a class needs a nontrivial destructor (because it holds resources), you probably also need to define a copy constructor and an assignment operator, even if **private**. Or, = **delete** them, so they cannot be used.
- The copy constructor for class **XYZ** will have signature

```
XYZ(const XYZ & other);
```

Typically, it copies any resources that would be destroyed by the destructor

## Summary, concluded

- The assignment operator **YOU** would write should be like:  

```
XYZ & operator= (const XYZ & other) {
 if (&other != this) { // DON'T COPY ONTO SELF!!!
 // PART I: DESTRUCTOR ACTIONS

 // PART II: COPY CONSTRUCTOR ACTIONS

 }
 return *this; // PART III: RETURN YOURSELF
}
```

  
but may do something smarter (e.g., reuse instead of deleting).

## Summary – Avoid pointer fields!

- Use smart pointers  
(`unique_ptr`, `shared_ptr` from `<memory>`)
- No more need for:
  - Copy constructors
  - Assignment operators
- Destructors can now be empty  
(and `virtual` if sub-classing possible)

(check end of handouts for `mystring.cc` without (*unsafe*) & with (*safe*) smart pointers)

## Next session

- Destructors, copy constructors, assignment operators and template classes.
- Program structure and separate compilation
- Include files in C++

Reading: Savitch section 11.1, Stroustrup chapter 9.

## Programming in C++

2024-11-28

Next session

Next session

• Destructors, copy constructors, assignment operators and template classes  
• Program structure and separate compilation  
• Include files in C++  
Reading: Savitch section 11.1, Stroustrup chapter 9.

## Final Notes – I

- There are four main modes of storage: static, local/stack, free/dynamic/heap, and temporary.
  - Static storage is the simplest and safest (used a lot in safety-critical real-time systems) but at the same time is extremely inflexible and wasteful.
  - Local storage is quite efficient and often just what we need; sometimes though it's not enough – we need our data to outlive the functions that created them.
  - Free storage uses new to allocate objects on the heap – these outlive the function that was active when they were created and stay on until someone calls delete on them explicitly.
- `delete p;` (destroy ONE object) vs `delete[] p;` (destroy an ARRAY of objects)
- Destructors for releasing resources – need for them to be virtual if the class is to be sub-classed (slides 14–15).
- Pay attention to the order of allocation/construction and destructor/deallocation (slide 16).

2024-11-28
Programming in C++

Next session

- Destructors, copy constructors, assignment operators and template classes
- Program structure and separate compilation
- Inside Makefile C++

Reading: Switch section 11.11, Stroustrup chapter 9.

## Final Notes – II

- Copy constructor – compiler always generates one if we haven't defined one.
- Why the compiler-generated copy constructor doesn't always do the right thing (and how to do it ourselves): slides 19–21.
- Assignment operator – compiler always generates one if we haven't defined one.
- Why the compiler-generated assignment operator doesn't always do the right thing (and how to do it ourselves): slides 22–24.
  - See also file `strings.cc` (<https://www.staff.city.ac.uk/c.kloukinas/cpp/src/lab08/strings.cc>) file from the lab for another alternative implementation of the assignment operator, that uses call-by-value and swap, so as to get the compiler to call the copy-constructor and the destructor implicitly instead of us re-writing the same code.
- Make sure you understand how to use the `this` pointer and that you understand that `*this` is the current object itself.

2024-11-28
Programming in C++

Next session

- Destructors, copy constructors, assignment operators and template classes
- Program structure and separate compilation
- Inside Makefile C++

Reading: Switch section 11.11, Stroustrup chapter 9.

## Final Notes – III

- **"The Gang of Three"** – you need one, you need all of them:
  - copy constructor
  - assignment operator
  - destructor
- Learn what THE COMPILER generates for them for some class `XYZ`.
- Also learn what the usual USER-DEFINED version of the assignment operator is for some class `XYZ`.
- **Note: (advanced)** Since C++11 it's the "Gang of Five"...
  - move constructor
  - move assignment operator

These "move", *i.e.*, steal the data, from the object that you're using to initialise/assign the current object instead of copying them.

[https://en.cppreference.com/w/cpp/language/rule\\_of\\_three](https://en.cppreference.com/w/cpp/language/rule_of_three)

2024-11-28
Programming in C++

Next session

- Destructors, copy constructors, assignment operators and template classes
- Program structure and separate compilation
- Inside Makefile C++

Reading: Switch section 11.11, Stroustrup chapter 9.

## Final Notes – IV

- You need to do delete explicitly – what could possibly go wrong?
  - 1 Do it too late (USE TOO MUCH MEMORY) (*in Java too*)
  - 2 Forget to do it (MEMORY LEAK)
  - 3 Do it too soon – still using the deleted memory (UNDEFINED BEHAVIOUR – usually crash)
  - 4 Do it more than once (UNDEFINED BEHAVIOUR – usually crash)
  - 5 Delete something that hadn't been new-ed (UNDEFINED BEHAVIOUR – usually crash)
  - 6 Use the wrong form of delete (UNDEFINED BEHAVIOUR – potential crash when `delete[] pointer_to_an_object;` or crash/memory leak when `delete pointer_to_an_array;`)
- **ADVANCED MEMORY MANAGEMENT ISSUES:**
  - 1 When you delete an object in C++ there is an LONG CASCADE OF DESTRUCTORS that is executed for its subobjects that can severely impact real-time systems (especially if deleting a container)
  - 2 Memory fragmentation: INABILITY TO ALLOCATE MEMORY even though there are enough free bytes; can be combatted with specialized memory allocators

2024-11-28
Programming in C++

Next session

- Destructors, copy constructors, assignment operators and template classes
- Program structure and separate compilation
- Inside Makefile C++

Reading: Switch section 11.11, Stroustrup chapter 9.

## Final Notes – V

- A number of garbage collectors suffer from #1 delayed collection (which freezes your program for quite some time), unpredictability (you have no idea when the GC will start working and can rarely control it, unlike manual deallocation), and sometimes #8 memory fragmentation (though some compact memory too). There are some real-time garbage collectors but none that can solve everybody's problems (perfection is not of this world...)
  - At least Java's GC protects you from all the other problems of C++'s manual memory deallocation (2 – 7 and sometimes from 8).
  - When a GC cannot help...
    - What if you need to control when destructors (Java's finalizers – deprecated!!!) run?
    - What if you need to reclaim another resource (DB, file, etc.)? You'd still need to do it manually in a GC-ed language. :- (
- Java does this with its new "try-with-resources" statement, where the "destructor" is called `close()`, see <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>
- The "try-with-resources" is syntactic sugar over `try-finally`.

2024-11-28

## Programming in C++

Next session

Next session

- Destructors, copy constructors, assignment operators and template classes
- Program structure and separate compilation
- Inside std::C++

Reading: Switch section 11.11, Stroustrup chapter 9.

## Final Notes – VI pointer, shared\_ptr

Don't use basic pointers as fields – use smart pointers!!!

```
// Pointer version. String arrays SHOULD NOT BE SHARED!

// comment out next line to see why we need the copy Xtor
// & the assignment operator.
#define SAFE

// g++-14 -std=c++20 (or c++23)
#include <cstring>
#include <iostream>

class my_string {
 std::size_t len;
 char* chars;
 my_string(int alen, const char *s)
 : len(alen), chars(new char[alen]) {
 for (std::size_t i=0; i<len; ++i) chars[i] = s[i];
 }
public:
 my_string(const char *s)//strlen doesn't count the last '\0'
 : len(std::strlen(s)+1), chars(nullptr) {
 my_string tmp(len, s);
 std::swap(chars, tmp.chars);
 }
 my_string() : len(1), chars(new char[1]) {chars[0] = '\0';}

#ifdef SAFE
 my_string(const my_string &other)
 : len(other.len), chars(nullptr) {
 // copy into your own internal array
 my_string tmp(other.len, other.chars);
 std::swap(chars, tmp.chars);
 }
 my_string &operator= (my_string other) {
 len = other.len;
 std::swap(chars, other.chars);
 return *this;
 }
#endif

 virtual ~my_string() // = default; // impl below used for demo
 { std::cerr << "~my_string:_\n"
 << (void *) chars << '_\n'
 << (chars?chars:"") << '\n'; }
};

#include "safe-string-main.cc"

// Safe version! String arrays are SHARED!
#include <cstring>
#include <memory>
#include <iostream>

class my_string {
 std::size_t len;
 std::shared_ptr<char[]> chars;
public:
 my_string(const char *s)
 : len(std::strlen(s)+1), chars(nullptr) {
 chars = std::make_shared<char[]>(len);
 for (std::size_t i=0; i<len; ++i) chars[i] = s[i];
 }
 my_string() : len(1), chars(std::make_shared<char[]>(1))
 {chars[0] = '\0';}

 // shared_ptr allows sharing, so copy Xtor & assignment op
 // just do shallow copy.

 virtual ~my_string()// = default; //impl below used for demo
 // chars.get() returns the actual pointer.
 { std::cerr << "~my_string:_\n"
 << (void *) chars.get() << '_\n'
 << (chars.get()?chars.get():"")
 << '\n'; }
};

#include "safe-string-main.cc"
```

2024-11-28

## Programming in C++

Next session

Next session

- Destructors, copy constructors, assignment operators and template classes
- Program structure and separate compilation
- Inside std::C++

Reading: Switch section 11.11, Stroustrup chapter 9.

## Final Notes – VII unique\_ptr, main

// Safe version! String arrays are NOT SHARED!

```
// g++-14 -std=c++20 (or c++23)
#include <cstring>
#include <memory>
#include <iostream>

class my_string {
 std::size_t len;
 std::unique_ptr<char[]> chars;
 my_string(int alen, const char *s)
 : len(alen), chars(std::make_unique<char[]>(alen)) {
 for (std::size_t i=0; i<len; ++i) chars[i] = s[i];
 }
public:
 my_string(const char *s) // strlen doesn't count the last '\0'
 : len(std::strlen(s)+1), chars(nullptr) {
 my_string tmp(len, s);
 std::swap(chars, tmp.chars);
 }
 my_string(): len(1), chars(std::make_unique<char[]>(1))
 { chars[0] = '\0'; }

 // unique_ptr don't allow sharing - by default deletes copy
 // Xtor & assignment op, so must do deep copying ourselves.
 my_string(const my_string &other)
 : len(other.len), chars(nullptr) {
 // copy into your own internal array
 my_string tmp(other.len, other.chars.get());
 std::swap(chars, tmp.chars);
 }
 my_string &operator= (my_string other) {
 len = other.len;
 std::swap(chars, other.chars);
 return *this;
 }

 virtual ~my_string() // = default; // impl below used for demo
 // chars.get() returns the actual pointer.
 { std::cerr << "~my_string:_\n"
 << (void *) chars.get() << '_\n'
 << (chars.get()?chars.get():"")
 << '\n'; }
};

#include "safe-string-main.cc"

/*****
 * This is safe-string-main.cc *
 *****/

int main() {
 {
 my_string empty;
 my_string s1("blah_blah");
 my_string s2(s1); // initialized from s1
 my_string s3 = s1; // initialized from s1
 } // all four strings are destroyed here

 {
 my_string s1("blah_blah");
 my_string s2("do_be_do");
 s1 = s2; // assignment
 } // the two strings are destroyed here

 return 0;
}

/*
 * If you have multiple pointer fields, then the smart pointer
 * versions are safe under exceptions, while the normal pointer
 * version is NOT.
 */
```

## Programming in C++

Session 9 – A generic class with dynamic allocation  
Declarations and definitions  
Program structure

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)



## This session

Two parts:

- 1 Completing memory management:  
A generic class with dynamic allocation
- 2 Program structure & separate compilation
  - Revision: declarations & definitions
  - Separate compilation in C++

## Part I

### A Generic Class with Dynamic Allocation

## Writing our own vector class

- Array to hold elements
  - (efficiency) Array often longer than #elements
  - Various vector operations
  - Array dynamically allocated, so destructor must free it
  - Since a non-trivial destructor, must have copy constructor & assignment operator
  - An iterator
  - A **swap** method is also useful
- Gang of Three!!!**

## A vector class

```
template <typename Elem>
class my_vector {
 size_t vsize; // # of elements stored - "vector size"
 size_t asize; // size of the array - "array size"
 Elem *array;
 // INVARIANT: 0 <= vsize <= asize && array.size() == asize
public:
 my_vector() : vsize(0), asize(1),
 array(new Elem[1]) {}

 size_t size() const { return vsize; }

 Elem & operator[](size_t i) { return array[i]; }
};

• array(new Elem[1]) – why not array(nullptr)?
```

## Programming in C++

2024-12-12

### A vector class

```
A vector class
template <typename Elem>
class my_vector {
 size_t vsize; // # of elements stored - "vector size"
 size_t asize; // size of the array - "array size"
 Elem *array;
 // INVARIANT: 0 <= vsize <= asize && array.size() == asize
public:
 my_vector() : vsize(0), asize(1),
 array(new Elem[1]) {}
 Elem & operator[](size_t i) { return array[i]; }
};
• array(new Elem[1]) – why not array(nullptr)?
```

`array(new Elem[1])` – why not `array(nullptr)`?

Because of the **invariant**!

For the invariant `vsize <= asize` to hold, `array` must be an actual array, otherwise `asize` is not defined. And `array.size()` must be equal to `asize`.

Why not `asize(0)`, `array(new Elem [0])`? Invariant is satisfied.

⇒ Because of the implementation of `push_back` on the next slide. (and because it'd be silly – avoid 0-length arrays)

## Shrinking and growing the vector

```
void pop_back() { vsize--; } // "forget" last elem

void push_back(const Elem & x) {
 if (vsize == asize) {
 asize *= 2; // Why *= 2 instead of ++? [*]
 Elem *new_array = new Elem[asize];
 for (size_t i = 0; i < vsize; ++i)
 new_array[i] = array[i];
 delete[] array;
 array = new_array;
 }
 array[vsize] = x;
 ++vsize;
}
```

[\*] try adding 1000 elements into a vector...

## Destructor and Copy constructor

This class allocates dynamic memory, so it should reclaim it:

```
virtual ~my_vector() { delete[] array; }
```

A non-trivial destructor ⇒ need a copy constructor & assignment operator

**Gang of Three!!!**

```
my_vector(const my_vector<Elem> & other) :
 vsize(other.vsize), asize(other.asize),
 array(new Elem[other.asize]) {
 for (size_t i = 0; i < vsize; ++i)
 array[i] = other.array[i];
}
```

## Assignment operator

```
my_vector<Elem> &
operator=(const my_vector<Elem> & other) {
 if (&other != this) {
 vsize = other.vsize;
 if (asize < vsize) { // Reuse if possible!
 delete[] array;
 asize = other.asize;
 array = new Elem[asize];
 }
 for (size_t i = 0; i < vsize; ++i)
 array[i] = other.array[i];
 }
 return *this;
}
```

REUSE!!! Compare with 8-24 & 8-31 !

## \*Solution: define an assignment operator

So define an assignment operator for `my_string`

```
my_string & operator= (const my_string &other) {
 if (&other != this){// DON'T COPY ONTO SELF!!!
 delete[] chars; // I: DESTRUCTOR ACTIONS

 len = other.len; // II: COPY CONSTRUCTOR ACTIONS
 chars = new char[len];
 for (std::size_t i = 0; i < len; ++i)
 chars[i] = other.chars[i];
 }
 return *this; // III: RETURN YOURSELF
}
```

## An iterator

Recall: a C++ *iterator* supports `==`, `++`, `*` and `->`

A simple iterator for this type is "pointer to elements":

```
typedef Elem *iterator; // I.e., iterator is a
 // pointer to an Elem
typedef const Elem *const_iterator;

 iterator begin() {return array;}
 iterator end() {return array + vsize;}
const_iterator cbegin() const {return array;}
const_iterator cend() const {return array + vsize;}
}; // end of my_vector class
```

Alternative: define `[*]` a class & overload operators `++`, `==`, `*`, `->`  
`[*]` Can be an internal class !

## Swap function

*Designing classes?*

*Think how they'll behave with standard algorithms  
(so we should know the standard algorithms...)*

The header `<utility>` defines a general swap function:

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

- Works for vectors too (`T` is `my_vector<Elem>`)
- But is **\*very\*** inefficient

(why?)



## Efficient swap function for vectors

Add a member function to the `my_vector` class:

```
void fast_swap(my_vector<Elem> & other) {
 std::swap(vsize, other.vsize);
 std::swap(asize, other.asize);
 std::swap(array, other.array); // <3 <3 <3
}
```

Overload `swap` for vectors outside the class:

```
template <typename T> //"C++ template specialization"
void swap(my_vector<T> & x, my_vector<T> & y) {
 x.fast_swap(y);
}
```

( **template specialization:** *constraining the function parameter type to `my_vector<T>` means this applies to our class only* )

We're done! :-)

## Part II

## Program Structure — Declarations vs Definitions

## Program structure

- In C++, X (class, function, variable) **must be declared before use**
    - Can *declare* X, and ...
    - *Define* it fully later
  - C++ programs can have *millions* of lines
    - Impossible (too slow) to recompile everything all the time
- ⇒ Programs are partitioned into several files for *separate compilation*
- Common declarations and partial class definitions are placed in *header files* (they serve as interfaces)

## Declaration before use

C++ designed for *one-pass* compilers: must declare entities before use

```
class A { ... };
```

```
class B { A *p; ... }; // OK
```

Defining these classes in the opposite order is illegal. Problems:

- limits presentation.
- prohibits recursion.

## Forward declarations

Solution: *Declare* first, and fully *define* later:

```
class A; // declare A as a type

class B { // define B
 A *p; // OK - pointer size is known
 ...
};

class A { B b1; ... }; // fully define A - OK
```

## Limitations

However, this is *NOT* allowed:

```
class A; // declare A

class B { // define B
 A a; // don't know the size of A here
 ...
};

class A { ... }; // define A
```

Because the size of a member must be known when it's used

## Recursive class definitions

This is allowed:

```
class A; // declare A

class B { // define B
 A *p; // pointer size is known
 ...
};

class A { // define A
 B b1; // size of B is known here
 ...
};
```

## Part III

### Separate Compilation

## Separate compilation

### General Idea

- Avoid recompiling a huge program after each change
  - Break it into “*modules*”, each with an interface
- Ideally: only recompile “modules” when the interfaces they use have changed
- If a “module” implementation (*but not its interface*) is changed, that “module” must be recompiled, but its clients need not be
- This should be **automated** (e.g., with **make**)

## Separate compilation in C++

- Implementations go into source files, usually ending in “.cc”
- Interfaces go into header files, usually ending in “.h”
  - Header files are included in source files and other header files
- **Never** duplicate declarations (include them instead)
- Recompilation decisions are based on inclusion relationships and timestamps on files

(Other suffixes: .cpp, .cxx, .hh, .hpp, .hxx, ...)

Inclusion relationships (as used by **make**) — try:

- **g++ -MM file.cc**
- **g++ -M file.cc**

## The compilation process

- Compiling a source file **x.cc** yields an object file **x.o**  
(like a .java file yields a .class file)
- **x.cc** must be recompiled if it (or any of the header files it uses) has changed more recently than **x.o**  
(so don't include header files unnecessarily)
- Object files are linked together to make an executable program  
(like an executable .jar file)
- Re-compiling source files means the program must be re-linked
- In Unix, this is all managed by the **make** command

## A Makefile

```
COMMANDS (e.g., rm) MUST START WITH A TAB CHARACTER!!!

DIR=.
CXX=g++-14 # or CXX=g++
CXXFLAGS=-I$(DIR) -x c++ -g -std=c++23 -pedantic -Wall -Wpointer-arith \
-Wwrite-strings -Wcast-qual -Wcast-align -Wformat-security \
-Wformat-nonliteral -Wmissing-format-attribute -Winline -funsigned-char
LDLAGS=-L$(DIR) -lc -lstdc++ # Linking flags
CC=$(CXX) # Use the C++ compiler as the C compiler
(ensures linking is done according to C++)
CFLAGS=$(CXXFLAGS) # C flags are now C++ flags

all: cwkt cwkt

clean:
 -rm *.o cwkt cwkt *~ 2> /dev/null

cwkt: sample.o Makefile libcity.a
 $(CXX) sample.o -o cwkt $(LDLAGS)

cwkt: cwkt.o Makefile libcityt.a
 $(CXX) cwkt.o -o cwkt $(LDLAGS)t

...
```

## Include directives

- `#include` includes the text of another file at that point.
- To include a file from the **system** directories:

```
#include <vector>
#include <iostream>
```
- To include a file from the **local** directories (`-I dir1 -I dir2`):

```
#include "point.h"
```
- `g++`: You can see what the result is with `-E`  
(`-E` runs only the C preprocessor on your file, doesn't compile)  
(and `-c` runs only the C compiler, doesn't link)
- Any file can be included, but the following rules are recommended

## Part IV

2024: Lecture 9 ended here

## Header files

These approximate interfaces, and may contain:

|                       |                                         |
|-----------------------|-----------------------------------------|
| comments              | <code>// what the class does</code>     |
| include directives    | <code>#include "xyz.h"</code>           |
| class definitions     | <code>class A { ... };</code>           |
| class declarations    | <code>class B;</code>                   |
| constant definitions  | <code>const double pi = 3.14159;</code> |
| type definitions      | <code>typedef double real;</code>       |
| function declarations | <code>int sqr(int x);</code>            |

**They should not contain code, except inline function definitions.**

## BE CAREFUL!

### NEVER IN HEADER FILES!

|                            |                                      |
|----------------------------|--------------------------------------|
| global variable definition | <code>int counter = 0;</code>        |
| function definition        | <code>int foo() { return 3; }</code> |

### INSTEAD YOU SHOULD

**DECLARE** global variables `extern int counter;`

**INLINE** function definitions `inline int foo() { return 3; }`

Or **DECLARE** functions `int foo();`

Otherwise, global variables/functions are defined multiple times from each source file that includes the header file **& linker complains!**

## The header file `point.h`, first version

```
// File: point.h
class point {
protected:
 int _x, _y;
public:
 point(int x, int y);
 int x() const;
 int y() const;
 void move(int dx, int dy);
};
```

Often, a header file and source file correspond to a single class, but there are many other possibilities.

## The implementation `point.cc`

```
// File: point.cc
#include "point.h"

point::point(int x, int y) : _x(x), _y(y) {}

int point::x() const { return _x; }
int point::y() const { return _y; }

void point::move(int dx, int dy) {
 _x += dx; _y += dy;
}
```

This is why we're so interested in defining methods **outside** a class!

## Separate compilation and templates?

NO

[isocpp.org/wiki/faq/templates#templates-defn-vs-decl](https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl)

- C++ DOES NOT support separate compilation of template code
- Generic method definitions must be included in the header file *WITH* the template class definition

*Wat Do?*

## Generic code separation

```
// File: pointt.h
template <typename T>
class pointt {
 pointt(T _x, T _y);
};
#include "pointt.cc" // <---- includes .cc !!!
// *End* of file pointt.h

// File: pointt.cc
// *NOT* including pointt.h! <---- !!!
// Definitions for pointt
template <typename T>
pointt<T>::pointt(T _x, T _y) {
 ...
}
```

## Code separation: Normal vs Generic

```
// point.h NORMAL
class point {
 point(int _x, int _y);
};

// *End* of file point.h

// File point.cc
#include "point.h"
// Definitions for point
point::point(int _x, int _y){
 ...
}

// pointt.h GENERIC
template <typename T>
class pointt {
 pointt(T _x, T _y);
};
#include "pointt.cc" // !!!
// *End* of file pointt.h

// File pointt.cc
// *NOT* including pointt.h!!!
// Definitions for pointt
template <typename T>
pointt<T>::pointt(T _x, T _y){
 ...
}
```

## Repeated inclusion

- Suppose `point.h` is included by both `line.h` and `polygon.h`. Some drawing program might begin:

```
#include "line.h"
#include "polygon.h"
```

- This includes `point.h` twice, causing the compiler to complain about a repeated definition of `point`
- Seems reasonable to expect the language to take care of this, **BUT**
  - C++ doesn't care about reasonable
  - We must add *include guards* to our header files

## The header file `point.h` with a proper include guard

```
#ifndef POINT_H
#define POINT_H

class point {
protected:
 int _x, _y;
public:
 point(int x, int y);
 int x() const;
 int y() const;
 void move(int dx, int dy);
};

#endif
```

**Don't use bloody `#pragma's`! (non-standard/portable)**

## Typical structure

- For each “public”/“modules” class `Foo`, two source files:
  - `Foo.h` containing the class definition, but including only very small methods. This is the place for comments describing the interface of the class.
  - `Foo.cc` containing the method definitions for the class (unless the class is very simple). This should always include `Foo.h`.
- Include header files only if necessary:
  - `Bar.h` should **ONLY** include `Foo.h`, when `Foo` is needed for defining class `Bar`
  - But when class `Foo` is only needed for defining methods of `Bar`, then include `Foo.h` only in `Bar.cc`
- Never **use** namespaces inside header files (**namespace pollution**). Instead use full names: `std::string`, `std::ostream`, etc. Exercise: break up `date.cc` in this way.

Summary

- In C++, things must be **declared** before use
- Often, a partial declaration (interface) will suffice (but the compiler needs to know how big things are)
- Large programs are broken up into several source files  
⇒ **separate compilation**
- **Common declarations** are placed in **header files** , to be included by several source files
- Shared generic code must also be placed in header files

Learn how to use **make**  
<https://www.gnu.org/software/make/manual/>

Next Session

- Exceptions in C++.
- **RAII** — *Resource Acquisition Is Initialization*: **C++'s GC !**  
A C++ technique so that resources are freed, even if exceptions, without writing exception-handling code  
(Java's **try-with-resources** on steroids)
- Reading: Stroustrup 14.4.
- RAII is a special case of the *smart pointer* and *proxy* patterns.

2024-12-12

Programming in C++

Next Session

Next Session

- Exceptions in C++
- **RAII** — *Resource Acquisition Is Initialization*: **C++'s GC !**  
A C++ technique so that resources are freed, even if exceptions, without writing exception-handling code  
(Java's **try-with-resources** on steroids)
- Reading: Stroustrup 14.4.
- RAII is a special case of the *smart pointer* and *proxy* patterns.

Final Notes – (empty)

2024-12-12

Programming in C++

Next Session

Next Session

- Exceptions in C++
- **RAII** — *Resource Acquisition Is Initialization*: **C++'s GC !**  
A C++ technique so that resources are freed, even if exceptions, without writing exception-handling code  
(Java's **try-with-resources** on steroids)
- Reading: Stroustrup 14.4.
- RAII is a special case of the *smart pointer* and *proxy* patterns.

Final Notes – (empty)

The type is a generic one, as the class is generic; type `my_vector` does not exist, only `my_vector<Elem>` exists!!!

```
// File: name.h - WITH generic code
#ifdef NAME_H
#define NAME_H
...
// Compiler needs to see the implementation
// of the generic code.
#include "name.cc"
#endif
```

```
// File: name.cc - WITH generic code
// No include of "name.h"!
...
```

- Separate compilation is automated with the **make** tool. On the terminal type: **info make**  
Or read the GNU documentation of **make** on-line:  
<https://www.gnu.org/software/make/manual/>

- Exceptions in C++.
- RAB** — Resource Acquisition Is Initialization: C++'s GC  
A C++ technique so that resources are freed, even if exceptions,  
without writing exception-handling code  
(Java's *try-with-resources* on steroids)
- Reading: Stroustrup 14.4.
- RAII is a special case of the smart pointer and proxy patterns.

- The C preprocessor (`cpre`) can do quite a lot of things (e.g., give you a headache...— advanced, not to be examined):
  - [en.wikibooks.org/wiki/C\\_Programming/Preprocessor](https://en.wikibooks.org/wiki/C_Programming/Preprocessor)
- X-Macros (for meta-programming with macros):
  - [en.wikibooks.org/wiki/C\\_Programming/Preprocessor#X-Macros](https://en.wikibooks.org/wiki/C_Programming/Preprocessor#X-Macros)
  - [www.embedded.com/design/programming-languages-and-tools/4403953/C-language-coding-errors-with-X-macros-Part-1#](http://www.embedded.com/design/programming-languages-and-tools/4403953/C-language-coding-errors-with-X-macros-Part-1#)
  - [www.embedded.com/design/programming-languages-and-tools/4405283/Reduce-C--language-coding-errors-with-X-macros---Part-2#](http://www.embedded.com/design/programming-languages-and-tools/4405283/Reduce-C--language-coding-errors-with-X-macros---Part-2#)
  - [www.embedded.com/design/programming-languages-and-tools/4408127/Reduce-C-language-coding-errors-with-X-macros--Part-3#](http://www.embedded.com/design/programming-languages-and-tools/4408127/Reduce-C-language-coding-errors-with-X-macros--Part-3#)

Hello headache! (No, I don't understand these either... but that doesn't mean that you cannot use them!

- *Outta This World!!!*  
<https://github.com/pfultz2/Cloak/wiki/C-Preprocessor-tricks,-tips,-and-idioms>



## Next Session

• Exceptions in C++  
• RAII – Resource Acquisition to Initialization  
• C++ techniques to manage resources in functions (avoiding leaks, avoiding resource leaks)  
• Reading: Stroustrup 14.4  
• RAII is a special case of the smart pointer and proxy patterns.

## Final Notes – V

Someone who knows much better [Rob Pike; last paragraph], argued (in 1989, so things may have changed) that most of the compilation time is spent doing lexical analysis (breaking input into tokens). Therefore, inclusion guards are sub-optimal, as the compiler reads the whole header file, then discards it. So he suggested this instead:

```
// File: header.h
#ifndef _HEADER_H
#define _HEADER_H
#include "_header.h"
#endif
```

```
// File: _header.h
// What you'd normally place in between
// the header guards above.
```

You're welcome.

[Rob Pike] "Notes on Programming in C" Feb 21, 1989

[https://doc.cat-v.org/bell\\_labs/pikestyle](https://doc.cat-v.org/bell_labs/pikestyle)

*"There's a little dance involving #ifdef's that can prevent a file being read twice, but it's usually done wrong in practice - the #ifdef's are in the file itself, not the file that includes it. The result is often thousands of needless lines of code passing through the lexical analyzer, which is (in good compilers) the most expensive phase."*

# Programming in C++

## Session 10 – When things go wrong: Exceptions and Resource management

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)



**CITY**  
ST GEORGE'S  
UNIVERSITY OF LONDON

Copyright © 2005 – 2024

## Outline

- Exceptions in C++
- **Resource acquisition is initialization (RAII)**
  - A **fundamental** C++ technique
  - Ensures that resources are freed, even in the presence of exceptions, without writing lots of exception-handling code.  
(Java's **try-with-resources** on steroids)
- RAII: a special case of the *smart pointer* and *proxy* patterns
- Plus Revision!

## Part I

## Exceptions

## Failures (revision)

- Method cannot meet its specification?  
⇒ Communicate this to its caller!
- May cause the caller to fail, and so on  
But sometimes the caller can work around the failure
- Might be necessary to clean up in the event of failure
- Traditional (C) approach – an **if** on a status variable – is very cumbersome (and often left out)
- Disciplined use of *exceptions* makes error-handling clearer and more robust

## Throwing an exception in C++

- Any object can be thrown (even basic types)  

```
class my_exception { ... };
```
- **throw** statements typically take TEMPORARY OBJECTS  

```
throw my_exception("Bad date");
```
- Exceptions should be caught BY REFERENCE
  - This is the “best practice”
  - Can also be caught by value  
But avoid it, since catch-by-value:
    - Slices derived exceptions
    - Requires copying (so extra memory)

## Catching an exception in C++

- C++ has **try/catch** statement (as in Java)  

```
try {
 // do something that might fail
} catch (my_exception &e) { // or derived
 // deal with the exception
} catch (AnotherException &ae) { // or derived
 // deal with the exception
}
```
- Like Java, exceptions may form hierarchies
  - A **catch** clause also handles any derived classes
- C++ has no **finally** clause  
*(we don't need no filthy finally clauses!)*

## The C++ treatment of exceptions

- If (inside a **try** block  
&& there's a matching **catch** clause)  
Then execute the first matching **catch** clause  
  
“matching” = the exception type or some base type of it
- Otherwise
  - Exit from the current block or function  
Destroying any locally allocated variables in the process, and
  - Continue searching for a matching **try** block
- If the **main** function is exited in this way  
Halt the program with an error message.

This is called **unwinding the stack**

## Exception – What do? QUICK QUIZ!!!

At a family party: cousin Jim starts to choke on a piece of meat!

- 1 Catch exception & ignore it  
Hide Jim in a closet & pretend nothing's happened
- 2 Catch exception & log it  
“Dear diary, Jim ruined the party once again. . .” (& into a closet)
- 3 Catch exception & fix the problem  
Help Jim spit what is choking him
- 4 Not catch the exception – propagate it to your caller, who might know how to fix it  
Call 999 and let 'em know someone's choking; they'll deal with it (if they can)

**HINT:** One should do neither #1 nor #2 . . .

## Clean up and rethrow

Often exception handlers are used to free resources on failure:

```
// acquire resource
try {
 // do something that might fail
 // free resource
} catch (...) { // any exception
 // free resource
 throw; // rethrow the exception
}
```

This can often be avoided, using the **RAII** technique

**"Resource Acquisition Is Initialization"**

**Note on syntax:**

- Catch any exception: **catch (...)**
- Rethrow an exception: **throw;**

## Resource management

- Programs acquire resources  
Allocate memory, open files, create windows, acquire locks, etc.
- These resources should be released  
*Even if there are exceptions!*
- Some resources are freed when a program terminates  
:-)
- But some are not, e.g., some kinds of lock  
:- (
- Releasing resources properly is tricky & easy to get wrong

## A typical pattern of resource use

Resources must often be released in the opposite order to acquisition:

```
// acquire resource 1
// ...
// acquire resource n

// use resources

// release resource n
// ...
// release resource 1
```

Wait – that's just like locally allocated data!

## Resource acquisition is initialization (RAII)

Introduce a resource management class with

- A constructor to acquire the resource (or just to record it)
- A destructor to release the resource
- Possibly an access method

Locally allocate an object of this class when acquiring the resource, and the resource will be **automatically** released!

Moreover, resources will be released in the correct order!

```
// Without RAII :- (// With RAII :-) :-)
// acquire resource {
try { // acquire resource
 // this might fail try {
 // now free resource // this might fail
} catch (...) { //any exception }
 // free resource } // resource freed here!
 throw; //rethrow exception
} //Single try in main is enough!
```

## Example: file streams

- `ifstream/ofstream`'s constructors open streams  
`ifstream inp("file.txt");`
- Their destructors close the streams (though one can do it earlier if required)
- Hence code safely like this:

```
{
 ifstream inp("file.txt");
 // read and process file
} // inp is destroyed here (IF inside a try{}!!!)
```

Whether control leaves the block normally or due to an exception, the file stream will be closed.  
*(must have a surrounding `try` somewhere!)*

## Example: storage management

This class manages the deletion of dynamically allocated `point` objects

```
class point_manager {
 point *ptr;

public:
 point_manager(point *p) : ptr(p) {}

 ~point_manager() { delete ptr; }
 point_manager(const point_manager &) = delete;
 point_manager &operator=(const point_manager &)
 = delete;
};
```

## Using the `point_manager`

Whenever a `point` that is only required for this block is dynamically allocated, make a local `point_manager` to manage it:

```
point *p1 = new point(20,30);
point_manager m1(p1);

point *p2 = window->get_middle();
point_manager m2(p2);
```

On leaving the block (normally, via `return`, or by an exception), then `m2` will be destroyed, which will `delete p2`, and then `m1`, which will `delete p1`.

## Generic storage management

The standard header `<memory>` **provided [\*)** a class `auto_ptr`. Here is a simplified version:

```
template <typename T> class auto_ptr {
 T *_ptr;

public:
 auto_ptr(T *ptr) : _ptr(ptr) {}

 ~auto_ptr() { delete _ptr; }
};
```

(more to come later)

**[\*) Until C++11 – deprecated since!!!**

## Using `auto_ptr` – The promise

- To ensure that dynamically allocated storage is reclaimed, create a local `auto_ptr` to manage it:

```
point *p = new point(20,30);
auto_ptr<point> p_ptr(p);
```

- On leaving the block, `p` is automatically deleted.
- One can also use `auto_ptr` as a subobject  
No need to write our own destructors!
- Since all methods are inline, there is very little overhead.

**IT'S A LIE!!!**

**IT'S A LIE!!!**

## More convenience

We add the following operator definitions to the `auto_ptr` class:

```
T & operator*() { return *_ptr; }
T * operator->() { return _ptr; }
```

Then we can use the `auto_ptr` as a *proxy* for the pointer:

```
auto_ptr<int> ip(new int);
*ip = 3;

auto_ptr<point> pp(new point(20,30));
pp->x = 4;
pp->y = 5;
```

## Completing `auto_ptr`

Gang of Three!

- Since `auto_ptr` has a non-trivial destructor, it requires
  - A copy constructor; and
  - An assignment operator
- **Only one of the copies of an `auto_ptr` should call `delete`.**
- Might as well add a default constructor too.

**Let's do it!**

```
template <typename T>
auto_ptr() : _ptr(nullptr) {}
```

```
template <typename T>
auto_ptr(auto_ptr<T> &other) { // *** NOT const & !!!
 _ptr = other._ptr;
 other._ptr = nullptr; // *** other loses pointer!
}
```

## Completing `auto_ptr` – II

```
template <typename T>
auto_ptr<T> &
operator=(auto_ptr<T> &other) { // *** NOT const & !!!
 if (this != &other) {
 delete _ptr;
 _ptr = other._ptr;
 other._ptr = nullptr; // *** other loses pointer!
 }
 return *this;
}
```

## (Smart pointers

`auto_ptr` is a so-called “*smart pointer*”

It looks like a pointer, but does something extra

Some other examples:

**reference counting** proxy counts references to a dynamically allocated object, and deletes it when count reaches zero

**persistent data** proxy reads data from a file on first use, and saves it in the file on destruction

**virtual/lazy object** proxy delays creating a complex object until it is used (and if the object is never used, avoids creating it)

## The Proxy pattern)

More generally, a *proxy* is any object that is interposed between the client and some other object. Some other uses:

**wrapper** proxy provides consistent access to foreign language data

**protection** proxy provides more limited access to the object, for greater security

**handle** proxy represents an object in a different address space, e.g., an operating system object, a graphical system object, or an object on another machine

...

*May you live  
in interesting times. . . : – (*

(2019: This 2011 statement did not age well at all!)

## C++11

- 1 `auto_ptr` deletes its pointer using `delete` !
  - So cannot manage a pointer to an array (needs `delete[]`)
- 2 `auto_ptr`'s “*copy*” constructor *steals* the other object's pointer!
  - That's not copying, that's moving! (*polite version of “stealing”*)
  - So cannot use `auto_ptr` inside STL containers (*containers think they copy elements when they don't*)
  - C++11: Use `unique_ptr` instead (or `shared_ptr`)
  - `unique_ptr` offers a *move* constructor but no copy constructor:

```
unique_ptr(unique_ptr<T> && x); // rvalue reference...
unique_ptr(unique_ptr<T> & x) = delete; // reference...
```

- You need to know how `auto_ptr` works, as old code uses it (BUG!)
- And to understand “*rvalue references*” (*and why we need them*)
- You need to learn the others for your coding
- These also work with arrays by the way:

```
unique_ptr<int []> array(new int[30]);
```

## C++11 – II

Advanced – not assessed (neither is `unique_ptr` nor rvalue references/move constructors).

- **shared\_ptr**:  
*"It's complicated"* (see [stackoverflow t.ly/1XveD](https://stackoverflow.com/questions/11347773/using-sharedptr))  
And the class documentation:  
[https://en.cppreference.com/w/cpp/memory/shared\\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr)  
Especially the constructors:  
[https://en.cppreference.com/w/cpp/memory/shared\\_ptr/shared\\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr/shared_ptr)
- **!!! Avoid temporary smart pointers.**  
*Why? See Boost t.ly/MfyGQ*
- Or BETTER YET use **make\_shared**  
*See [stackoverflow t.ly/bN-1L](https://stackoverflow.com/questions/11347773/using-sharedptr)*

```
C++11
auto_ptr deletes its pointer using delete.
// So cannot manage a pointer to an array (invalide delete[])
auto_ptr<T> is not a smart pointer. It's just a wrapper around object's pointer.
// So it's not strong, that's moving (poor version of "stealing")
// Can't not use auto_ptr, it's inside STL containers
// contains them (they copy elements when they don't)
C++11: Use unique_ptr instead (or shared_ptr)
unique_ptr, weak_ptr offers a more constructor but no copy constructor.
unique_ptr(unique_ptr<T> &u) {} // value reference...
unique_ptr(unique_ptr<T> &u) = delete; // reference...
// You need to know how auto_ptr works, as old code uses it (BUGS)
// You need to understand "strong" and "weak" (and why we need them)
// You need to learn the others for your coding
// These also work with arrays by the way:
unique_ptr<int>() { array<new int[100]> }
```

## Further reading

- Exceptions: Stroustrup 14, Meyer 12.
- Resource acquisition is initialization (RAII): Stroustrup 14.4.
- Smart pointers: Stroustrup 14.4.2, 11.10.

## Part II

## Revision

## Major Differences between Java and C++

**C++ allows direct access to objects!!!**

- **[?]** call-by-value & call-by-reference (session 1 and since)
- operator overloading (session 3)
- genericity or template classes (sessions 4–6)
- **[?]** slicing of derived objects on copying (session 6)
- memory management
  - local allocation of objects (sessions 1–2 and since, esp. 9–10)
  - pointers (sessions 5 and 6)
  - dynamic allocation (sessions 8–9) **[?]** No GC
- multiple inheritance (session 7)
- **[?]** gang of three (session 8)
- **[?]** Rvalue references (& call-by-rvalue-reference – session 10)

**[\*] Because C++ allows direct access to objects...**



## Things you should be able to do

- Write simple C++ classes/functions
- Use STL containers/iterators to write compact (& correct!) code
- Understand how call-by-value & call-by-reference differ
- The various meanings of **const** & know when to use it
- Read programs using overloaded operators; identify which methods/independent functions are called
- Overload operators for new types
  - As member functions
  - As independent functions

(continued)

## More things you should be able to do

- Distinguish between objects & pointers (& how each behaves)
- Know how to use static, local, dynamic and temporary allocation, appreciating their properties and distinctive features
- Understand the properties of subobjects (= fields of other objects)
- Use inheritance, method redefinition and abstract classes in C++
  - Know the order of initialisation (parents [\*], fields [\*], constructor) and destruction (opposite) [\*] IN THE ORDER OF DECLARATION!!!!

BE CAREFUL WITH FIELD INITIALISATION!!!

- Write generic C++ classes/functions
- Use the standard generic algorithms!!!

(continued)

## Even more things you should be able to do

- Multiple inheritance – both replicated & virtual inheritance
- Explain — Gang of Three
  - ① What the automatically generated constructors, destructors & assignment operators do
  - ② When they are inadequate, and if so
  - ③ How they should be replaced
- Use the exception syntax of C++ (**try**, **catch**, **throw**, **rethrow**)
- **Use RAI** ("resource acquisition is initialization")  
to safely release resources,  
*even in the presence of exceptions*
  - Use **unique\_ptr** (and less often **shared\_ptr** [\*]) to automatically manage your pointers  
[\*] sharing makes it harder to parallelise)

## Programming in C++

2024-12-13

└ Even more things you should be able to do

Empty On Purpose

Even more things you should be able to do

- Multiple inheritance – both replicated & virtual inheritance
- Explain — Gang of Three
  - ① What the automatically generated constructors, destructors & assignment operators do
  - ② When they are inadequate, and if so
  - ③ How they should be replaced
- Use RAI ("resource acquisition is initialization") to safely release resources, even in the presence of exceptions
- Use **unique\_ptr** (and less often **shared\_ptr** [\*]) to automatically manage your pointers  
[\*] sharing makes it harder to parallelise)

## Even more things you should be able to do

Even more things you should be able to do

- Multiple inheritance – both replicated & virtual inheritance
- Explicit – C++ is Strong
- What the automatically generated constructors, destructors & assignment operators do
- Other things are interesting, and so
- How they should be replaced
- Use the exception system of C++ (try, catch, throw, noexcept)
- Use RAI (Resource Acquisition Is Initialization) to safely release resources
- Use unique\_ptr (and lots other smart\_ptr) to automatically manage your pointers (C++ sharing makes it harder to parallelize)

### Final Notes – I

- Java has **Exception** (or some such) from which all exceptions MUST derive.
- C++ doesn't impose such a constraint (though it does have **std::exception** that you could derive from)
  - So you can throw/catch an object of ANY class in C++ (even basic types – but avoid this).
- **Good practice:** throw a TEMPORARY object!  
**throw my\_exception("Not your lucky day!");**
- How can I catch it?  
 The same way I can receive a parameter – EITHER BY VALUE (exception is \*COPIED\* and \*SLICED\* – BAD!) or BY REFERENCE (GOOD!)

```
try {
 // dangerous stuff
} catch (problem1 p1) { // catch BY VALUE – BAD! BAD! >:-(
 // exception object COPIED and POTENTIALLY SLICED
 // treat p1
} catch (problem2 & p2) { // catch BY REFERENCE – GOOD! :-)
 // exception object NOT COPIED
 // treat p2
}
```

## Even more things you should be able to do

Even more things you should be able to do

- Multiple inheritance – both replicated & virtual inheritance
- Explicit – C++ is Strong
- What the automatically generated constructors, destructors & assignment operators do
- Other things are interesting, and so
- How they should be replaced
- Use the exception system of C++ (try, catch, throw, noexcept)
- Use RAI (Resource Acquisition Is Initialization) to safely release resources
- Use unique\_ptr (and lots other smart\_ptr) to automatically manage your pointers (C++ sharing makes it harder to parallelize)

### Final Notes – II

- A **catch** clause catches all exceptions of derived classes too – be careful to place clauses for these classes before the clauses of their superclasses.
- If no **catch** clause matches, then the function is terminated, destroying all its local stack-allocated variables, and the system looks for a matching catch clause in its caller.
- As exceptions can belong to ANY class (even basic types...), we cannot write **catch (Exception &e)** to catch any kind of exception. Instead we need to use the ellipsis notation in C++  
**catch (...)** matches any exception
- In order to state that we want to re-throw the same exception we simply write: **throw;** (EVEN when we have a name for the exception – it makes explicit that we're re-throwing)

- Resource allocation very often uses a pattern similar to stack-based allocation (acquire, use, release), thus the pattern:  
**"Resource Acquisition Is Initialization (RAII)"**  
**Introduce a local manager object for the resource that releases the resource in its destructor.**

In this way it is released whether the code block is terminated normally or through an exception, avoiding boiler-plate code with try/catch clauses.

- Simple example of that: **point\_manager** (slides 14–15)

## Even more things you should be able to do

Even more things you should be able to do

- Multiple inheritance – both replicated & virtual inheritance
- Explicit – C++ is Strong
- What the automatically generated constructors, destructors & assignment operators do
- Other things are interesting, and so
- How they should be replaced
- Use the exception system of C++ (try, catch, throw, noexcept)
- Use RAI (Resource Acquisition Is Initialization) to safely release resources
- Use unique\_ptr (and lots other smart\_ptr) to automatically manage your pointers (C++ sharing makes it harder to parallelize)

### Final Notes – III

- Standard manager class: **auto\_ptr** (slides 16–20)  
 An example of a **"smart pointer"** (which are examples of the **"proxy"** pattern)
- **auto\_ptr** copy constructor:  

```
template <typename T>
auto_ptr<T>::auto_ptr(/*NO const!*/ auto_ptr<T> & other)
: _ptr(other._ptr) { other._ptr = nullptr; }
```
- **auto\_ptr** assignment operator:  

```
template <typename T>
auto_ptr<T> &
auto_ptr<T>::operator=(/*NO const!*/ auto_ptr<T> & other)
{
 if (&other != this) {
 delete _ptr;
 _ptr = other._ptr; // MOVE (STEAL) THE POINTER
 other._ptr = nullptr;
 }
 return *this;
}
```

## Even more things you should be able to do

Even more things you should be able to do

- Multiple inheritance – both replicated & virtual inheritance
- Explicit – C++ is Strong
- What the automatically generated constructors, destructors & assignment operators do
- Other things are interesting, and so
- How they should be replaced
- Use the exception system of C++ (try, catch, throw, noexcept)
- Use RAI (Resource Acquisition Is Initialization) to safely release resources
- Use unique\_ptr (and lots other smart\_ptr) to automatically manage your pointers (C++ sharing makes it harder to parallelize)

### Final Notes – IV

- **auto\_ptr** is badly broken...
  - It calls **delete**, so cannot handle arrays of objects (these need **delete []**)  
*(OK, can always have a pointer to a vector)*
  - It says it has a copy constructor but it doesn't copy, it **"moves"** the value from the other object into itself – major breakage!  
 Cannot use them in standard containers!!!
- In C++11 **auto\_ptr** has been deprecated and replaced by **unique\_ptr**
- You still need to learn how to implement **auto\_ptr** and understand it and its problems  
 Only then you'll understand why we need rvalue references

└ Even more things you should be able to do

## Final Notes – V

- What to do when you receive an exception?  
You're at a family party and cousin Jim starts to choke on a piece of meat!
  - ❶ Catch the exception and ignore it – hide Jim in a closet and pretend nothing's happened.
  - ❷ Catch the exception and log it – “Dear diary, Jim once again ruined the party. . .” (after having hidden Jim in a closet).
  - ❸ Catch the exception and fix the problem – Help Jim spit the piece of meat that is choking him.
  - ❹ Not catch the exception but let it propagate instead to your caller (or catch/rethrow), who might know how to fix it – Call 999 and let them know there's someone choking; they'll deal with it (if they can).

**HINT:** It's neither #1 nor #2 that you should be doing...

- Multiple inheritance – both replicated & virtual inheritance
- Explain – Gang of Three
  - ❑ What the automatically generated constructors, destructors & assignment operators do
  - ❑ When they are inadequate, and if so
  - ❑ How they should be replaced
- Use the exception syntax of C++ (`try`, `catch`, `throw`, `nothrow`)
- Use RAII ("resource acquisition is initialization")  
to safely release resources,  
*even in the presence of exceptions*
- Use `auto_ptr`, `weak_ptr` [and less often `shared_ptr`] to  
automatically manage your pointers  
[?] sharing makes it harder to parallelise

└ Even more things you should be able to do

## Final Notes – VI

Further pointers:

- “What should I throw?”  
A temporary object.  
<https://isocpp.org/wiki/faq/exceptions#what-to-throw>
- “What should I catch?”  
Catch by reference if given the choice (avoids copying).  
<https://isocpp.org/wiki/faq/exceptions#what-to-catch>
- “But MFC seems to encourage the use of catch-by-pointer; should I do the same?” (aka When in Rome...)  
When working with MFC yes, otherwise no as it's not clear who's responsible for deleting the pointed-to object.  
<https://isocpp.org/wiki/faq/exceptions#catch-by-ptr-in-mfc>
- “What does throw; (without an exception object after the throw keyword) mean? Where would I use it?”  
Re-throw.  
<https://isocpp.org/wiki/faq/exceptions#throw-without-an-object>
- “How do I throw polymorphically?”  
To catch derived exceptions instead of base exceptions, make sure you're throwing derived exception objects! Use virtual functions.  
<https://isocpp.org/wiki/faq/exceptions#throwing-polymorphically>
- “When I throw this object, how many times will it be copied?”  
Nobody knows (zero up to some N) but the exception object must have a copy-constructor (even if the compiler will never copy it).  
<https://isocpp.org/wiki/faq/exceptions#num-copies-of-exception>
- Check out on StackOverflow the iterator proxy I created for implementing **copy\_if\_and\_transform**  
<https://stackoverflow.com/questions/23579832/why-is-there-no-transform-if-in-the-c-standard-library-74288551#74288551>  
or **t.ly/1LCtT**  
*(it tries to make **\*from** behave differently, depending on the context)*

- Multiple inheritance – both replicated & virtual inheritance
- Explain – Gang of Three
  - What the automatically generated constructors, destructors & assignment operators do
  - When they are inadequate, and if so
  - How they should be replaced
- Use the exception syntax of C++ (try, catch, throw, rethrow)
- Use RAII ("resource acquisition is initialization")
  - to safely release resources.
  - even in the presence of exceptions
- Use `unique_ptr` (and less often `shared_ptr`) to
  - automatically manage your pointers
  - (if sharing makes it harder to pass them)