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)

CITY ST GEORGE'S UNIVERSITY OF LONDON

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 *Gang of Three!!!*
- An iterator
- A `swap` method is also useful

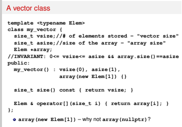
A vector class

```
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)`?



`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$(DIR) -x c++ -g -std=c++23 -pedantic -Wall -Wpointer-arith \
 -Wwrite-strings -Wcast-qual -Wcast-align -Wformat-security \
 -Wformat-nonliteral -Wmissing-format-attribute -Winline -funsigned-char
LDFLAGS=-L$(DIR) -lcity # 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:    cwk cwkt

clean:
        -rm *.o cwk cwkt *~ 2> /dev/null

cwk:    sample.o Makefile libcity.a
        $(CXX) sample.o -o cwk $(LDFLAGS)

cwkt:   cwkt.o Makefile libcityt.a
        $(CXX) cwkt.o   -o cwkt $(LDFLAGS)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 (`-Idir1 -Idir2`):

```
#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	`// what the class does`
include directives	`#include "xyz.h"`
class definitions	`class A { ... };`
class declarations	`class B;`
constant definitions	`const double pi = 3.14159;`
type definitions	`typedef double real;`
function declarations	`int sqr(int x);`

They should not contain code, except inline function definitions.

BE CAREFUL!

NEVER IN HEADER FILES!

global variable definition	`int counter = 0;`
function definition	`int foo() { return 3; }`

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

- 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 polution)**
  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
  $\Rightarrow$ ***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.

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)

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 – I

- Why not initialize member **`array`** in **`my_vector`**'s default constructor with **`nullptr`**? (slide 5)
  Because then we'd be violating the class ***invariant***:
  `vsize <= asize`
  If **`array`** is not pointing to an array, then **`asize`** isn't defined.
- **`my_vector`**'s assignment operator (slide 8) shows that sometimes we can reuse resources instead of always destroying the ones we've got and copying those of the other object.
  - Note the parameter type of the copy constructor and the assignment operator (and the operator's return type):

```
template <typename Elem>
class my_vector {
  public:
  my_vector( const my_vector<Elem> & o);
  my_vector<Elem> &
  operator=( const my_vector<Elem> & o);
  ...
};
```

  The type is a generic one, as the class is generic; type **`my_vector`** does not exist, only **`my_vector<Elem>`** exists!!!
- Outside the class:

```
template <typename Elem>
my_vector<Elem>:: my_vector( const my_vector<Elem> & o)
  : ... {
  ...
}
template <typename Elem>
my_vector<Elem> &
my_vector<Elem>:: operator=( const my_vector<Elem> & o) {
  ...
}
```

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 – II

- Implementation of the iterator type for class **`my_vector`** (slide 9)
- Slide 11 – the **`swap`** specialised for objects of type **`my_vector`**, is another example of partial specialization! The type of its arguments is still generic but now we know that it's a **`my_vector`** of some **`T`**.
- Things need to be declared (not necessarily defined) before they're used – slides 13–17.
- Separate compilation – CLASS DEFINITIONS with METHOD DECLARATIONS go into the HEADER file **`NAME.h`**, while the method IMPLEMENTATIONS into the SOURCE file **`NAME.cc`**. See slides 27–28.

Which file should include which?

- If there's no generic code, then we include **`NAME.h`** at the top of **`NAME.cc`** and compile the latter into **`NAME.o`**
- If there is generic code, then we include **`NAME.cc`** at the bottom of **`NAME.h`** (compiler needs to see the implementation of the generic code to be able to instantiate it where it's used) but do not ask the compiler to produce **`NAME.o`** (pointless – it'll be empty).

ALL other files that need to know the types defined in **`NAME.h`** include **`NAME.h`** (**NEVER `NAME.cc`**).

- To avoid "multiple definition" compiler errors, we surround the entire contents of **`NAME.h`** with include guards (*NOT* pragma's!!!):

```
// File: name.h – WITHOUT generic code
#ifndef NAME_H
#define NAME_H
...
#endif
```

  This ensures that the compiler will see the contents only the first time **`NAME.h`** is included (when **`NAME_H`** hasn't been defined).

```
// File: name.cc – WITHOUT generic code
// Get declarations
#include "name.h"
```

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 – III

- Things change a bit with generic code:

```
// File: name.h – WITH generic code
#ifndef NAME_H
#define NAME_H
...
// Compiler needs to see the implementation
// of the generic code.
#include "name.cc"
#endif
```

  and the source file:

```
// File: name.cc – WITH generic code
// No include of "name.h"!
...
```

  Afterwards **`NAME_H`** will get defined, so the contents between the **`#ifndef`** and the **`#endif`** will not be considered again.
- 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/`

2024-12-12
Programming in C++
Next Session

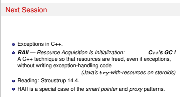
Next Session

- 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 – IV

- The C preprocessor (**`cpp`**) 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`
- X-Macros (for meta-programming with macros):
  - `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#`
  - `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#`

  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`



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
"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."