Programming in C++

Session 8 – Memory Management

Dr Christos Kloukinas

City, UoL

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

(slides originally produced by Dr Ross Paterson)

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?

  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

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 is guaranteed.

inflexible Must determine limits at compile-time.

wasteful We often allocate more than needed. Also, the storage is held for the entire execution, even if it is not being used.

Static function/class variables are 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 parameters of a function: initialized from the actual parameters.
- Variables local to a function or block, optionally initialized. The value of an uninitialized variable is 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"?

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...

Dynamically allocated **arrays** in C++

A pointer can also address a dynamically allocated array:

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

Note the special syntax for deletion syntax, which is required because C++ doesn't distinguish a pointer to an `int` from a pointer to an array of `int`s.

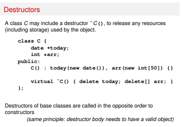
Destructors

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; }
};
```

Destructors of base classes are called in the opposite order to constructors

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



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; }
};
```

Destructors of base classes are called in the opposite order to constructors

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

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:

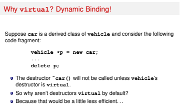
```
#include <memory>
#include <utility>
using namespace std;
class C {
 unique_ptr<pair<float,float>> upair;// prefer unique_ptr
 shared_ptr<pair<float,float>> spair;// over shared_ptr
 unique_ptr<float[]> uarr;// unique_ptr supports arrays
      // as well in C++11/14 - shared_ptr only in C++17
public: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

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. . .



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. . .

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**
static object	***before*** main starts	***after*** main terminates
local object	when the declaration is executed	on exit from the function or block
free object	when **new** is called	when **delete** is called
subobject ***[*]***	when the containing object is created (constructed ***before*** the containing object is constructed)	when the containing object is destroyed (deleted ***after*** the containing object is destructed)

[*] 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(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:

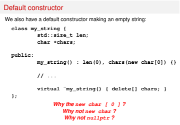
```
my_string(const char *s) : len(strlen(s)), chars(0) {
  chars = new char[len];
  for (std::size_t i=0; i<len; ++i) chars[i]=s[i];
}
```

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(0), chars(new char[0]) {}

        // ...

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

Why the `new char [ 0 ]` ?
Why not `new char` ?
Why not `nullptr` ?



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(0), chars(new char[0]) {}
// ...
virtual ˜my_string() { delete[] chars; }
};
Why the new char [0] ?
Why not new char ?
Why not nullptr ?

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`. . .

More reasonable code would have been:

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

Code in slide used to highlight the importance of the class invariant!

Initialization of objects

- Initialization is not assignment: the target is empty.
- Initialization invokes a constructor with arguments of the appropriate type, *e.g.*,
  `my_string foo = "bar";`
  invokes the above constructor: `my_string(char *)`
- Initialization from another `my_string` object invokes the ***copy constructor***, which is a constructor with signature
  `my_string(const my_string &s);`
- If no copy constructor is supplied for a class, the compiler will generate one that does a memberwise copy.
  This may not always be the right thing. . .

Here:

```
my_string(const my_string &s)
  : len(s.len), chars(s.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 the last initialization, **`s1`**, **`s2`** and **`s3`** all point at the same array of characters.
- The array will be deleted **three times**!

(Bad, bad karma. . .)

Solution: define a copy constructor

We define a copy constructor to copy the character array:

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

- This copying (“*deep copy*”) is typical:
  With explicit deallocation, it is 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 &s);
```

- If no assignment operator is supplied for a class, the compiler will generate one that does a memberwise copy.
- The compiler’s code for it is

```
    my_string & operator= (const my_string &s) {
         len   = s.len;
         chars = s.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:

- The original array pointed to by **`s1`** is discarded without being deleted.
- After the assignment, both **`s1`** and **`s2`** point at the same array of characters, which is thus deleted twice.

Solution: define an assignment operator

We define an assignment operator inside the `my_string` class:

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

      len = s.len;    // II: COPY CONSTRUCTOR ACTIONS
      chars = new char[len];
      for (std::size_t i = 0; i < len; ++i)
              chars[i] = s.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;
    }
};
```

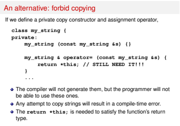
(1) Looping over a C string.
(2) What does that line do?
** Why do we destroy our string parameter `s` by doing `++s`?!?

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.
- The `return *this;` is needed to satisfy the function's return type.



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.

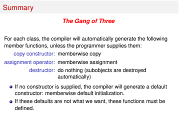
Summary

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 is supplied, the compiler will generate a default constructor: memberwise default initialization.
- If these defaults are not what we want, these functions must be defined.



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 is supplied, the compiler will generate a default constructor: memberwise default initialization.
If these defaults are not what we want, these functions must be defined.

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 26).

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

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!

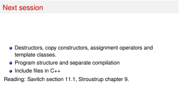
- Use smart pointers
  (**`unique_ptr`**, **`shared_ptr`** from **`<memory>`**)
- No more need for:
  - Copy constructors
  - Assignment operators
- Destructors can now be empty
  (and **`virtual`** if sub-classing possible)

(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.



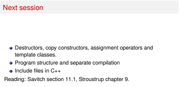
Next session

- 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 12–13).
- Pay attention to the order of allocation/construction and destructor/deallocation (slide 14).



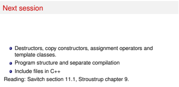
Next session

- 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 – 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 17–19.
- 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 20–22.
  - 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.



Next session

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

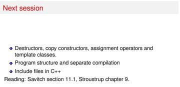
Reading: Savitch section 11.1, Stroustrup chapter 9.

Final Notes – 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`



Next session

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

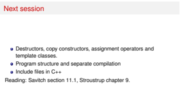
Reading: Savitch section 11.1, Stroustrup chapter 9.

Final Notes – 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:

  7. 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)
  8. Memory fragmentation: INABILITY TO ALLOCATE MEMORY even though there are enough free bytes; can be combatted with specialized memory allocators



Next session

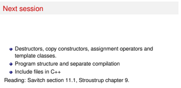
- Destructors, copy constructors, assignment operators and template classes.
- Program structure and separate compilation
- Include files in C++

Reading: Savitch section 11.1, Stroustrup chapter 9.

Final Notes – IV

- 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`**.

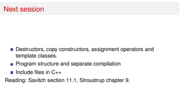


Next session

- 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.

Empty page – Check next!

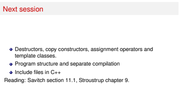


Next session

- 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.

Empty page – Check next!

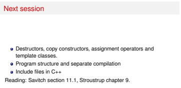


Next session

- 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.

Empty page – Check next!



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 – V

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

```
// Unsafe version!
#include <cstring>
#include <iostream>
class my_string {
 std::size_t len;
 char *chars;
public:
 my_string(const char *s)
   : len(std::strlen(s)), chars(0) {
   chars = new char[len];
   for (std::size_t i=0; i<len; ++i) chars[i] = s[i];
}
 my_string() : len(1), chars(new char[1]) {*chars = ’\0’;}

 virtual ˜my_string() { delete[] chars; // print below used for demo
                        std::cerr << "˜my_string\n"; }
};

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;
}
```

```
// Safe version!
#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)), chars(0) {
   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’;}

 virtual ˜my_string() // = delete; // impl below used for demo
                      { std::cerr << "˜my_string\n"; }
};

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;
}
```