Classes

15 October 2017 21:20

Class - default private:

- this hides inner details from outside world / different object types

Data - private
- if public - breaks encapsulation, **can lead to inconsistent state**

Functions:

- member functions - called methods
  - should be only things that can access data fields
  - e.g. read data x for object mO (getter)
    - use method - mO.get_x()
- in .cpp
  - use class name before function name
  - e.g.
    - class Point - function: double get_x();
    - in .cpp - double point::get_x(){
      }
- const functions
  - makes functions read-only
    - protection layer on object that the function was called on
  - if any other functions call this, compiler needs to know if function is const
    - if parameter was passed as const
- function overloading
  - two functions same name
  - different parameters
    - can have same types of paraments - but order must be different
    - e.g.
      - STRING INT
      - INT STRING
    - parameters act like a functions signature to the compiler

Constructors:

- all data fields should ALWAYS be initialised
  - default constructors
  - use default -

```
point my_point;          // use default constructor - no ()
point my_point(1,1);     // use constructor - (values)
```

Constructor - acquire resources
Destructor - release

Useful to declare an object within a loop

in .cpp

Class::Function

::

- binary scope resolution operator

Overloading, Operators & Fields

15 October 2017 22:46

Compile with headers

Pre-processor directives

In Header

- Header - just declarations
- #ifndef POINT_HPP
- #define POINT_HPP
- #endif
- guards from multiple class declarations

in .cpp

- #include "point.hpp"
  - do this in class source file
    - and in main.cpp

In Console

- Compile class source files - point.o
  - g++ -c point.cpp
  - do this for main.cpp as well
- Linking phase -
  - g++ point.o main.o -o NAME

Copy constructor

- in vector
  - vector<Point> vp(3, Point(0, 0));
    - 3 elements of 0, 0
- provided by default

Overloading

- implicit conversion
  - won't check double's for TYPE
- pointers work for overloading
- pointers are associated with a TYPE

Operator Overloading:

- not a member function unless a special operator (+=)
  - Some operators can never be overloaded
  - e.g. + on prime types
- bool operator== (relaxational)
  - due to LHS and RHS
    - try to avoid making overloaded operators member functions
      - as then LHS operand must be of the type of the class
- pre
  - operator++(variable) - pre
  - operator++() - post

Default Arguments:

- / / in point. hpp
- Point(double in_x = 0 , double in_y = 0) ;
  - set in function declaration
- If the default values can be used in the function in the exact same way as any other values (i.e. they are not a special case) use default arguments.
- Use overloading when default values impact behaviour of the class

#include <typeinfo>

- typeid(variable).name()
  - returns variable TYPE

Friend:

- global - not member
- allows access to the private fields

needed for insertion and extraction

```
friend ostream &operator<< (ostream &o, const Class &c){
    o << c.data;
    return o;
    }
```

Keeping in Memory

19 October 2017 16:39

const Point* p1 - p1 protected (kind of)
Point* const p2 - p2 pointer address value locked

Compiler:
-Wall
gives warnings

destructor
- **~Point()**
  - called when exiting a scope
  - called when 'delete' a variable

OOP:

- **don't make getters for memory address of data fields**
  - would allow external editing
    - breaks encapsulation
- if necessary
  - use const point*
    - gives memory address
    - but value cannot be changed via this pointer
    - **copy of this pointer can bypass protection**

Reference

can use . operator, unlike pointer which uses ->
int& new_a = a; - has to be initialised

reference to memory location of a
cannot change the address it points to

const reference:
- **read only security**
- **efficiency of not copying**

can have function return a reference as return type
can also make this const
return int& can return type int

return a pointer
- if want to return pointer to local variable - unreliable
- instead use **new**

to pass a temporary variable by ref,

must use a const ref

vs normal ref to a permanent variable

Copying, assigning, things like *this*

29 October 2017 23:41

<u>temporary variables</u>

destructor called end of scope
either end of loop
or as a parameter to a function / method

end of main - is end of a scope
calls destructor on all variables

int* foo = new int[#]

- **new array size #**

delete[] foo;

- deletes array

<u>Vector Capacity</u>

When doubling
new array
delete old array
calls destructor

Vector - a **container class**

- should be able to modify it's 'private data'
- hence why the class doesn't protect encapsulation

push_back:

- delete[] v
  - invalidates all memory addresses of v
    - v pointer is still safe
      - - will be pointed to head of tmpv
  - can assign v to the new (double capacity) array

Subscript Operator Overloading - []

- two versions
  - one version for when called on const objects
- MUST BE MEMBER FUNCTION

<u>Copy Constructor</u>

- default **bitcopy** - provided by compiler - also in **assignment =**
- if dealing with memory addresses - need to make your own

this

- pointer used within method to point to object that called the method

Inheritance

subclass EXTENDS base

06 November 2017 02:16

Composition:

- Main class constructor - do this in .cpp
  - Triangle::Triangle (parameters)
    : // initialisation list
    {}
  - : list
    - comes before curly braces {}

subclass

- inherits from base class
  - inherits all members
  - can redefine base class functions

- public
- private
  - not accessible by subclass
- protected
  - accessible by subclass object or base class object
    , not global

Polymorphism

13 November 2017 11:03

friend - not inheritable
Redefined - function called on sub class object
Override - function called through pointer of base class

Polymorphism - can pass a subclass object in place of base

- say a funciton has parameter person
  - parent is subclass of person
  - can pass parent object instead of person as the argument
- also used on pointers to person

Binding:

- connecting a function call to a function body
- early / static
  - by compiler and linker
- late / dynamic
  - runtime, based on the actual type of object

Overriding needs to be enabled (in C++ by default it's disabled):
declaring member function virtual in base class.

virtual
in base class
regular - choice of overriding or not
will call funciton through reference of object
instructs compiler to perform **late binding**

Abstract class - just has pure virtual functions

=0 - pure virtual function
abstract class
no base implementation
must be overridden

virtual destructor
call derived class destructor instead

useful for using as reference / pointers
pass derived objects to them

Templates

20 November 2017 11:01

<type>
general type

Subtyping
template <class T1, class T2>
returns type on RHS

template <class Type>
template <typename Type>
placeholder
takes **any primitive or class**
write **above both defention and declaration**
keep in same file - usually header
can overload
different amount of type parameters
to fore the choice of one version:
- function<int,int> ()
  - forces version with int
- ranking algorithm choses normally

Class Templates
use type throughout class
Member definition still possible
in header file
use typename
for friend member functions

give default type
template <class T = double>
class myClass{

};

Compiler
compile time
fills in parameter type
once created a specific implementation
can reuse if required again
will choose specific parameter version over template
if available
uses matching algorithm
override:
function<int, int>(a,b)
forces template version of two ints
unless type conversion - weird

Object slicing
assigning extended object to base
discards extended properties

STL

- ::iterator
  - e.g. vector<int>::iterator
  - is a type specific to vector<int>
  - can dereference with *
  - use const_iterator
    if object is const
  - **why use? - more general, if use diff container, don't have to rewrite cope**

- **<algorithm> sort()**
  - **element type need to overload < operator**
  - **container being sorted needs to also have random access iterators**

typename:

```
typename std::list<T> l;
```

Exceptions

27 November 2017 11:05

Out of bounds in vector - cause a throw

exit(EXIT_FAILURE) - terminates program

Resource Acquisition is Initialization

- use an object for resources
- constructor locks the resource
- destructor frees the resource

throw "description"

try block
catch block

- When a function `throw`s an exception:
  - No other subsequent instructions in the function are executed.
  - Control goes immediately back to the caller.
- If in the caller the function call is in a `try` block:
  - No other subsequent instructions in the `try` block are executed.
  - Control goes to the `catch` block.
- Otherwise control goes up one more level.
  - Until a `try` block is encountered, or the top level is reached (which means the exception was not handled!).

- can catch various things
  - parameter defines what a catch block catches

Exam Stuff

03 June 2018 10:49

Structured vs OOP:

Structured:

- focus on flow of program execution
- single entry and exit point of groups of instructions
  - control by conditions, if..else..loops
- enclosed in functions - call each other and communicate via parameter and return value

OOP:

- focus on concept of state
  - represented by member data
  - manipulated and accessed from outside using member functions
- objects are instances of classes
- classes can be organised to express roles, relationships, hierarchies etc

Encapsulation: protecting state
Abstraction: keeping state consistent without higher level knowing the inner workings

UML:

Hollow arrow - inheritance
Shaded arrow - composition
Middle section: - data: type
Bottom section: +method(arg1:rg2..): type

Dynamic memory allocation requires the program to allocate memory when it is needed and deallocate it when it is not needed anymore. **Memory leaks** occur when dynamically allocated memory areas become unreachable before being deallocated and thus are wasted; the repeated occurrence of this may deplete the available memory.