Introduction to Programming and Computing for Scientists

Vytautas Vislavicius

Lund University

Lectures 6

C++ class

- A class is a container for **data** and **functions**.
- An instances of the class are called an ***objects***, e.g. you can create many instances of class `line` as (2,3), (1,0), (-3,2).
- Two mandatory functions: **constructor** creates new objects; **destructor** cleans up when an object is destroyed.

```
#ifndef LINE__H
#define LINE__H
#include <iostream>
class line
{
  public:
    line();
    ~line();
    line(line &lin);
    line(double fx1, double fx2);
    double GetX1();
    double GetX2();
    void SetX1(double newvalue);
    double GetLength();
  protected:
    double x1, x2;

};
#endif
```

Inheritance

- Inheritance is a way to share characteristics among similar types.

```
class triangle { //Let's take this chance to observe some good programming conventions
  public:
  triangle(double base = 0., double height = 0.);
  ~triangle();
  double area();
  double getBase(); //Getters and setters are used to handle private info
  double getHeight();
  void setBase(double base); //This function should make sure that the base is positive
  void setHeight(double height);

  private: //All member varibles should in general be private to facilitate encapsulation
  double base_; //Name private members with an underscore to avoid shadowing
  double height_;
};
```

```
class rectangle {
  public:
  rectangle(double base = 0., double height = 0.);
  ~rectangle();
  double area();
  double getBase();
  double getHeight();
  void setBase(double base);
  void setHeight(double height);

  private:
  double base_;
  double height_;
};
```

Inheritance

- We'd have to repeat a lot of code to write the `triangle` and `rectangle`.
- Inheritance simplifies this immensely. Both `triangle` and `rectangle` are really special cases of something more general. Let's call it a `shape`.

```
#include <cmath> //For fabs

class shape { //This class has the common characteristics of triangles and rectangles
  public:
  shape(double base = 0., double height = 0.);
  ~shape();
  double getBase() { return base_; } //Simple functions can be defined here in the header
  double getHeight() { return height_; }
  void setBase(double base) { base_ = fabs(base); }
  void setHeight(double height) { height_ = fabs(height); }

  protected: //Protected members can be accessed by this and whatever inherits from this
  double base_;
  double height_;
};
```

```
#include "shape.h"

shape::shape(double base, double height) {
  setBase(base); //Let's call these functions rather than duplicate the code
  setHeight(height);
}

shape::~shape() { }
```

Inheritance

- Now we'll make `triangle` inherit from `shape`. The only new code we have to write is whatever is specific to `triangle` (in this case the `area` function).

```
#include "shape.h" //Include the class that we want to inherit from

class triangle : public shape { //Triangle inherits from shape, access levels unchanged
  public:
  triangle(double base = 0., double height = 0.); //A ctor/dtor must still be provided
  ~triangle();
  double area() { return base_*height_/2.; } //This function is specific to triangle
};
```

```
#include "triangle.h"

triangle::triangle(double base, double height) : shape(base, height) {
  //The first (and in this case only) thing to do is initialize the parent object
}

triangle::~triangle() {
  //At the end of this destructor, the parent destructor is called automatically
}
```

- `shape` is the 'base' or 'parent' class, while `triangle` is the 'derived' class.
- When an object is created, the base part should always be constructed first. Destruction follows the opposite order - The base should be destroyed last.

Polymorphism

- Let's pretend the area of a shape is so crucial, we have functions to check it.

```
bool isBigEnough(rectangle& obj) {
  return obj.area() > 10.;
}

bool isBigEnough(triangle& obj) {
  return obj.area() > 10.;
}
```

- This is clunky because every new kind of shape needs its own function.
- Ideally, we'd like to have a single function that works with any kind of shape.

```
bool isBigEnough(shape& obj) {
  return obj.area() > 10.;
}
```

- This function will happily accept `triangle` and `reclangle` objects as arguments. They inherit from `shape`, so they *are* shapes.
- The problem is that `shape` does not declare an `area` function, so the compiler complains. We can try adding one to the class definition.

```
double area() { return 0.: } //Let's add this to shape.h
```

Polymorphism

- This small test program doesn't print the answer we want. The problem is of course that `isBigEnough` calls the `area` function in `shape`, which returns 0.

```
#include <iostream> //For cout
#include "triangle.h" //Both triangle.h and rectangle.h include shape.h
#include "rectangle.h" //I added #ifndef macros in shape.h, so it isn't doubly defined
using namespace std;

bool isBigEnough(shape& obj) { //A shape is big enough if its area is greater than 10
  return obj.area() > 10.;
}

int main() { //Create some shapes, print their area and see if they're big enough
  triangle tri(10., 10.);
  cout << "Triangle with area " << tri.area();
  if(isBigEnough(tri)) cout << " is big enough!" << endl;
  else cout << " is NOT big enough!" << endl;

  rectangle rec(5., 5.);
  cout << "Rectangle with area " << rec.area();
  if(isBigEnough(rec)) cout << " is big enough!" << endl;
  else cout << " is NOT big enough!" << endl;

  return 0;
}
```

```
Triangle with area 50 is NOT big enough!
Rectangle with area 25 is NOT big enough!
```

Polymorphism

- We can get the desired behavior by making the `area` function `virtual`.

```
virtual double area() { return 0.: } //Change the area function in shape.h to this
```

- When a function is `virtual`, derived classes are allowed to override it. If `isBigEnough` receives a `triangle`, the `triangle` version of `area` is called.
- The call to `area` thus behaves differently depending on whether the `shape` is a `triangle` or `rectangle`. Such a function is said to be *polymorphic*.
- After `area` is declared `virtual`, the test program gives the expected output.

```
Triangle with area 50 is big enough!
Rectangle with area 25 is big enough!
```

- You seldom need to tell a function what kind of `shape` it is dealing with at compile time. The proper behavior is achieved through polymorphism.
- Good use of inheritance and polymorphism will make code much easier to read, maintain and extend. One of the great strengths of OO programming!

Polymorphism

- Virtual functions work the same way with pointers as with references.

```
bool isBigEnough(shape* obj) {
  return obj->area() > 10.; //Works as expected - area is called in the derived class
}
```

- Pointers mesh well with inheritance, because a pointer of type base class can point to a derived class. Remember, triangles and rectangles *are* shapes.

```
shape* shapePtr = new shape(10, 10); //Nothing new - A shape pointer pointing to a shape
isBigEnough(shapePtr); //The area function in shape is called, so this returns false

shape* triPtr = new triangle(10, 10); //Perfectly OK - Any triangle is also a shape
isBigEnough(triPtr); //The area function in triangle is called, so this returns true

triangle* illegalPtr = new shape(10, 10); //Not OK - A shape isn't necessarily a triangle
```

- If a class is handled polymorphically, is should have a `virtual` destructor.

```
virtual ~shape(); //Make the destructor virtual in shape.h, or you're in for trouble
```

```
shape* triPtr = new triangle(10, 10); //A shape pointer that points to a triangle
delete triPtr; //Calls ~shape(). Make it virtual so the triangle part is destroyed too!
```

Polymorphism

- We made `shape` return an area of zero, but in reality it is undefined.
- This is a valid concern. In fact, it doesn't make sense to instantiate a `shape` in the first place. Only `triangle` and `rectangle` are meaningful objects.
- To avoid this logical inconsistency, make the `area` function pure `virtual` in `shape`. A class that has a pure `virtual` function can not be instantiated.
- A class that contains at least one pure `virtual` function is called *abstract*.

```
virtual double area() = 0; //Put this in shape.h to make area a pure virtual function
```

- Any class that inherits from `shape` must now either implement `area` or be abstract itself. This ensures that no one can misuse our `shape` class.

```
triangle t(10, 10); //No problem - A triangle is a meaningful object
shape s(10, 10); //This will not compile. Shape is abstract and can not be instantiated!
```

Memory Leak

- Hardest bugs to trace typically are related with memory managing and allocation.
- Most common mistake is Memory Leak - forgetting to free allocated memory.

```
#include <stdlib.h>     /* atof */
#include <unistd.h>     /* usleep */
using namespace std;

int main(int argc, char* argv[]) {

    const int sizeOfAlloccationInBytes = 300000000; //300 MB
    int numberOfAllocations = 10;
    double timeToSleepBetweenAllocations = 0.3 * 1000000; // microseconds

    for (int i=0; i<numberOfAllocations; i++){
        char *myChar = new char[sizeOfAlloccationInBytes]; // allocate memory in heap
        for (int j=0; j<sizeOfAlloccationInBytes; j++){
            myChar[j] = 'q';
        }
        usleep(timeToSleepBetweenAllocations);
        delete myChar; // <-- possible place of memory leak, if one forget to add this line
        myChar = NULL;
    }

    return 0;
}
```

The const keyword

- Declare a variable as `const` when you want to be certain that it is never modified. Trying to do so then results in a compile time error.

```
const int var = 10; //Remember to initialize at declaration time. Const variables can't be modified later
var = 20; //Nope! Because var is const, this results in a compile time error
```

- The `const` keyword acts on whatever word or symbol is to its immediate left. If there is nothing to its left, it acts on whatever is to its right instead.

```
int const var = 10; //These two lines are completely equivalent
const int var = 10; //Pick one usage and be consistent
```

- A const pointer (`int* const p`) must point to the same variable forever.
- A pointer to const (`int const* p`) can't be used to assign to a variable.

```
int foo = 10;
int bar = 20;

int* const p1 = &foo; //p1 is a constant pointer to int (so the pointer is const but not foo)
*p1 = 30; //No problem
p1 = &bar; //Error! p1 must forever point to foo

int const* p2 = &foo; //p2 points to a constant int (so foo is const but not the pointer itself)
*p2 = 30; //Error! foo can't be assigned to via p2. Assigning via e.g. p1 is still fine, though.
p2 = &bar; //No problem
```

The `const` keyword

- `const` variables and objects are picky about how they are used. They will only work with functions that have promised in advance not to change them.
- A function can promise not to change an argument by declaring it as `const`.
- This is relevant only when passing arguments by reference or pointer. When an argument is passed by value, any modifications are local to the function.

```
#include <iostream>

using namespace std;

void passByValue(int foo) { cout << foo << endl; } //None of these functions actually modify foo
void passByPtr(int* foo) { cout << *foo << endl; }
void passByConstPtr(int const* foo) { cout << *foo << endl; } //But this one explicitly promises not to!

int main() {
  const int foo = 10;
  passByValue(foo); //No problem! The function can only modify a local copy of foo
  passByPtr(&foo); //Compile time error! Function could in principle modify foo
  passByConstPtr(&foo); //OK! The function has promised not to modify foo
}
```

The const keyword

- Member functions of an object can be declared as const to promise that they won't try to modify any of the object's member variables.
- This promise must be made before a const object will use the functions.

```
class date { //This class represents a day, month and year
  public:
  date(int day, int month, int year); //You get the idea, so let's skip everything but the month part
  int getMonth() const; //This function is const - It promises not to change any of the member variables
  void setMonth(int month);

  private:
  int month_;
};
```

```
const date myBirthday(23, 8, 1986); //Changing my birthday makes no sense at all. I'll make it const!
int month = myBirthday.getMonth(); //No problem, getMonth has promised not to change anything
myBirthday.setMonth(8); //Results in a compiler error because setMonth is not const. It might make changes!
```

- Code that works as intended with const variables is called "const correct".
- If you want your code to be const correct, *do it right from the start!* It is extremely difficult to take a program that is not const correct and fix it.

Operator overloading

- When an operator does more than one thing, it is said to be overloaded.
- For example, the addition operator + is overloaded. It adds when acting on integers, but concatenates when acting on strings.

```
int aVal = 10;
int bVal = 20;
int cVal = aVal + bVal; //The addition operator adds two numbers and returns the sum, so cVal = 30

string aStr = "Hello";
string bStr = ", world!";
string cStr = aStr + bStr; //Now the same operator concatenates two strings, so cStr = "Hello, world!"
```

- Operators are really just convenient shorthands for function calls. The `operator` functions have silly names, but they are ordinary functions.

```
c = a.operator!(); //Equivalent to c = !a
c = a.operator+(b); //Equivalent to c = a + b
c = operator+(a,b); //Also equivalent to c = a + b. The operator doesn't have to be a member of a
```

- You might be tempted to try and call the `operator` functions directly. This will work for user-defined types, but not built-in ones (like `int` and `double`).

Operator overloading

- Any class can overload an operator by implementing the appropriate `operator` function. This can be convenient and make code more readable.
- Let's implement the += (increment), == (equality) and `!=` (inequality) operators for the `shape` class.

```
class shape {
  public:
  shape& operator+=(const shape& toAdd); //Increment
  bool operator==(const shape& toCompare) const; //Equality. Once we have this, defining != is easy
  bool operator!=(const shape& toCompare) const { return !(*this == toCompare); } //Use existing equality
};
```

```
shape& shape::operator+=(const shape& toAdd) { //We get to decide for ourselves what addition means
  base_ += toAdd.getBase(); //Let's decide that it means to add up the base and height
  height_ += toAdd.getHeight();
  return *this; //Return a reference to the object to enable chaining of operations (a + b + c + ...)
}

bool shape::operator==(const shape& toCompare) const { //We get to decide what it means to be equal
  if(base_ != toCompare.getBase()) return false; //Let's define it as having the same base and height
  if(height_ != toCompare.getHeight()) return false;
  return true;
}
```

Operator overloading

- Implementing == and `!=` as member functions works fine. But they don't *have* to members, because they don't need access to private variables.
- The usual convention is to implement operators that don't change the state of the object as non-member functions.

```
class shape {
  //Put the shape class here as usual
};

bool operator==(const shape& lhs, const shape& rhs); //Then define the operators outside
bool operator!=(const shape& lhs, const shape& rhs); //They are not members of the class
```

```
bool operator==(const shape& lhs, const shape& rhs) { //Implement the functions. Note lack of shape::
  if(lhs.getBase() != rhs.getBase()) return false;
  if(lhs.getHeight() != rhs.getHeight()) return false;
  return true;
}

bool operator!=(const shape& lhs, const shape& rhs) { //The inequality can again make use of the equality
  return !(lhs == rhs);
}
```

What is coding style?

Style == Readability

- How and when to use comments,
- Tabs or spaces for indentation (and how many spaces),
- Appropriate use of white space,
- Proper naming of variables and functions,
- Code grouping an organization,
- Patterns to be used/avoided.

Coding style

- **Style == Readability**
- Why Coding Style Matters?
  - Make Errors Obvious.
  - Easy to understand logic of your own old codes.
  - Style is mandatory in any sw developer team.
- EasyToStart tip: use editor plugins, e.g. flymake-google-cpplint for Emacs (accompany with iedit, google-c-style).

```
#include <float.h>
#include <stdarg.h>

// Let's try typing 'while'

int main(int argc, char *argv[]) {
  int variable_index = 0;
  while ( variable_index != 10) {
    printf("This is good");
    variable_index ++;
  }

  return 0;
}
```

Clean Code: **Meaningful names**

- Use meaningful (intention-revealing) names

```
const int size;
int nCycles;
double time;
```

Better

```
const int sizeOfAlloccationInBytes;
int numberOfAllocations;
double timeToSleepBetweenAllocations;
```

Clean Code: **Functions**

- Keep functions **small** (no more than 20 lines)

```
public void renderWebPage() {
  StringBuilder content = getContentBuilder();
  content.append("<html>");
  content.append("<head>");
  for (HeaderElement he : getHeaderElements()) {
    String headerEntry = he.getStartTag() + he.getContent() +
      he.getEndTag();
    content.append(headerEntry);
  }
  content.append("</head>");
  content.append("<body>");
  for (BodyElement be : getBodyElements()) {
    String bodyEntry = /* .. */
      content.append(bodyEntry);
  }
  content.append("</body>");
  content.append("</html>");
  OutputStream output = new OutputStream(response);
  output.write(content.toString().getBytes());
  output.close();
}
```

Clean Code: **Functions**

- Keep functions **small** (no more than 20 lines)

```
public void renderWebPage() {
  startPage();
  includeHeaderContent();
  includeBodyContent();
  endPage();
  writePageToResponse();
}

private void startPage() { /* ... */ }

private void includeHeaderContent() { /* ... */ }

private void includeBodyContent() { /* ... */ }

private void endPage() { /* ... */ }

private void writePageToResponse() { /* ... */ }
```

Clean Code: **Functions**

Function

- should do **one thing**,
- should do it **well**,
- should do it **only**,
- with less arguments is easier to use.

```
calendar.SetDate(2014,2,3);
```

```
calendar.SetDate(todaysDate);
```

Clean Code: **Comments**

- Intuitively understandable code is better than complex code with a lot comments.

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) != 0 && (employee.age > 65))
```

```
if (employee.isEligibleForFullBenefits())
```

- Describe why you do something - not how!

```
// We need to remove duplicates from the names because
// a person cannot have the same name more than once.
Set<String> uniqueNames = new HashSet<String>(names);
```

- Don't comment obvious things

```
// Check if members have been initialized. If not, do it!
if (members == null) {
  members = new ArrayList<Member>();
}
```

Final words

- Aim for simpilicity, whenever possible.
- Stick to one coding style. Importance of code readability usually are underestimated.
- Use Coding Tools.
- Use Google.

Backup Slides

Type Casting

- Type casting is when a variable of one type is converted into another type.
- Casts that aren't specifically requested by the programmer are called *implicit*. Built-in types are often implicitly converted into each other.

```
int a = 10;
double b = a; //The int is implicitly converted into a double
```

- Any type can be implicitly converted if the appropriate constructor exists.

```
class effort { //This empty class is just used for the sake of example
  //We'll let the compiler automatically generate a constructor and destructor
};
```

```
class success {
  public:
  success(effort& toConvert) {} //Success can be constructed from effort
};
```

```
effort e;
success s = e; //The effort is implicitly converted into success
```

Type Casting

- There are four types of explicit casts in C++:
  - dynamic_cast
  - static_cast
  - reinterpret_cast
  - const_cast
- Let's pretend, for the sake of example, that shape was never made abstract.

```
shape* shapePtrToShape = new shape(); //shape pointer pointing to a shape
shape* shapePtrToTri = new triangle(); //shape pointer pointing to a triangle
triangle* triPtrToTri = new triangle(); //triangle pointer pointing to a triangle
```

- dynamic_cast is used to cast pointers and references within class hierarchies. Downcasting only works on polymorphic classes.
- The legality is checked at run time. Be careful! You will not get any warnings at compile time. An illegal cast of a pointer returns NULL.

```
shape* upCast = dynamic_cast<shape*>(triPtrToTri); //Always OK
triangle* downCast = dynamic_cast<triangle*>(shapePtrToTri); //OK if polymorphic
triangle* illegalCast = dynamic_cast<triangle*>(shapePtrToShape); //Not OK, returns NULL
```

Type Casting

- `static_cast` works like `dynamic_cast` except the legality of the cast is not checked at all. This avoids some overhead at run time. Faster but less safe.
- In addition, `static_cast` can do any conversion that could have been done implicitly. Use it freely when converting between built-in types.

```
int a = 9;
int b = 10; //Dividing two ints returns another int, rounded down
double ratio = static_cast<double>(a)/b; //Would return 0 if not for the cast
```

- `reinterpret_cast` can convert between references and pointers of (almost) any type. Even unrelated classes can be converted into each other.
- There are valid uses for `reinterpret_cast` (such as interfacing identical classes from several third party libraries), but having to `reinterpret_cast` generally indicates that your code is not well designed.

Type Casting

- `const_cast` turns a `const` variable into non-`const`. Can be used to modify `const` variables or pass them to functions that take non-`const` arguments.

```
void printThis(char* str) { //str should be declared const because it's not modified
  cout << str << endl; //As is we couldn't pass a const char*, which is bad design
}

const char* constString = "Text to print";
printThis(const_cast<char*>(constString)); //A const_cast is required to use the function
```

- There are also C-style casts. But it's not recommended to use them!

```
//Base to derived? Derived to base? Getting rid of const? Integer to pointer?
quantity* qPtr = (quantity*) var; //It's impossible to tell because I used a C-style cast
```

- C++ casts do only one thing, making the intention of the programmer clear. A C-style cast will try every type of cast until it finds one that succeeds.
- This makes C-style casts dangerous! They might not do what you intend, but since they still perform legal operations the compiler does not complain.

Templates

- Sometimes you want a function or class that works with more than one type.
- For example, we might want a function `compare` that determines which out of two inputs is greater. It should work with any type, including custom ones.
- The naive approach is to copy the source code for each type, but that is not sustainable in the long run. A better solution is to write a template function.

```
template <typename T> //typename is a C++ keyword, T is a name that you choose to represent the type
int compare(const T& val1, const T& val2) {
  if (val2 < val1) return 1; //The first argument is greater
  if (val1 < val2) return -1; //The second argument is greater
  return 0; //The arguments are equal
}
```

- Now any type that implements `operator<` can use the function! In general, templates should try to place as few requirements on the types as possible.

Templates

- Classes can also be templates. The syntax is the same as for functions.

```
template <typename T> //Put the template keyword before the class, just like for functions
class calculator { //This simple calculator class can add and multiply objects
  public:
  T multiply(T val1, T val2);
  T add(T val1, T val2);
};

template <typename T> //The function implementations should also be preceded by the template keyword
T calculator<T>::multiply(T val1, T val2) {
  return val1*val2; //For this to work, the type must implement operator*
}

template <typename T>
T calculator<T>::add(T val1, T val2) {
  return val1 + val2; //For this to work, the type must implement operator+
}
```

- Now we can make calculators for any type. The syntax for creating templated types is the familiar one from vectors, maps and so on.

```
calculator<double> doubleCalc; //This calculator works with doubles
calculator<shape> shapeCalc; //This would work with shapes if they defined operator* and operator+
```

References

- http://indico.cern.ch/event/404359/timetable
- http://www.smashingmagazine.com/2012/10/why-coding-style-matters/