Introduction to Programming and Computing for Scientists

Anders Floderus

Lund University

Lectures 3 and 4

Hello, world!

```
#include <iostream> //Standard input/output library

int main() {
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

- The `main` function is mandatory in every program.
- It returns an integer, where 0 means that the program execution finished successfully. Anything else indicates failure.
- `iostream` is the name of a library. It defines many objects and functions, like `cout` and `endl`, which print to the standard output.
- `std::` denotes the namespace where `cout` and `endl` live. Different functions can share a name if they reside in different namespaces.

Control structures - if, else

```
if(condition) {
  statement;
}
else if(condition) {
  statement;
}
else {
  statement;
}
```

- if evaluates the condition. If it is true, the statement is executed.
- If it is false, the statement in the optional else clause is executed.
- if and else can be nested.

```
if(5 == 10) {
  std::cout << "This computer is insane" << std::endl;
}
else if(5 == 5) {
  std::cout << "Everything is fine" << std::endl;
}
else {
  std::cout << "This will never happen" << std::endl;
}
```

Control structures - for, while

```
for(initialization; condition; statement) {
  statement;
}

while(condition) {
  statement;
}
```

- The `for` and `while` loops execute statements while some condition is met. They are functionally equivalent.
- Use a `for` loop when you know how many iterations you want to do.
- Use a `while` loop when the number of iterations is unknown, for example if the stopping condition depends on user input.

```
for(int i = 0; i < 10; ++i) {
  std::cout << "i equals " << i << std::endl;
}

bool keepGoing = true;
while(keepGoing) {
  std::cout << "Still going!" << std::endl;
  keepGoing = readUserInput(); //This magical function returns true or false
}
```

Control structures - `continue`, `break`

- The `continue` statement is used in loops to skip directly to the next iteration. It works in both `for` and `while` loops.

```
for(int i = 0; i < 10; ++i) {
  if(i == 5) continue; //5 won't be printed
  std::cout << "i equals " << i << std::endl;
}
```

- The `break` statement is used to exit the loop entirely. It works in `for` and `while` loops as well as `switch` clauses (next slide).

```
while(true) {
  std::cout << "Still going!" << std::endl;
  if(readUserInput() != true) break;
}
```

Control structures - switch, do-while

- The `switch` clause can be used to replace many `if` statements.

```
switch(variable) {
  case 0:
  std::cout << "variable is 0" << std::endl;
  break;

  case 1:
  std::cout << "variable is 1" << std::endl;
  break;

  default:
  std::cout << "variable is neither 0 nor 1" << std::endl;
}
```

- The `do-while` loop works like a `while` loop, except the condition is checked at the end of the loop instead of the beginning.
- This guarantees that the statement will be executed at least once.

```
bool keepGoing = true;
do {
  std::cout << "Still going!" << std::endl;
  keepGoing = readUserInput(); //This magical function returns true or false
} while(keepGoing);
```

Scopes

- A scope is a region within a program where a variable is visible.
- When a variable falls out of scope, it can no longer be used.
- This program will not compile because of scope errors.

```
#include <iostream> //For cout

int globalScope = 0; //This is a global variable. It is visible everywhere.

void foo() {
  int funcScope = 1; //Only visible within this function
  std::cout << localScope << std::endl; //Error! localScope is only visible in main()
}

int main() {
  std::cout << globalScope << std::endl; //OK! Will print 0

  { //Any block declares a scope, even this useless one
    int localScope = 3; //Only visible within this block
    std::cout << localScope << std::endl; //OK! Will print 3

    foo();
    std::cout << funcScope << std::endl; //Error! funcScope is out of scope
  }
  std::cout << localScope << std::endl; //Error! localScope is out of scope
}
```

Namespaces

- A namespace is a place where variables, classes and functions live.
- They can share names as long as they live in different namespaces.
- Typing `std::` in front of all standard functions soon gets tiresome. The `using` keyword allows them to be used without a qualifier.
- If you use an entire namespace, beware of collisions (e.g `std::count` exists).

```
#include <iostream> //For cout

using std::cout; //Now we don't have to type std::cout. Just cout will do.
using namespace std; //Like the above but for everything in the std namespace

namespace first {
  int a = 10;
}

namespace second {
  int a = 20;
}

int main() {
  cout << first::a << endl; //Will print 10
  cout << second::a << endl; //Will print 20
  first::a = 30;
  std::cout << first::a << std::endl; //Will print 30. Using std:: still works.
}
```

I/O - Standard input and output

- We already know how to use `std::cout` to write to standard output. To read from standard input, use `std::cin`.

```
#include <iostream> //For cin and cout

int main() {
  int userInput = 0;
  std::cin >> userInput;
  std::cout << "The user provided " << userInput << std::endl;
  return 0;
}
```

- The read operation evaluates to true if successful. A common trick is to read many times by putting it in a `while` loop.

```
#include <iostream> //For cin and cout

int main() {
  int userInput = 0;
  int sum = 0;
  while(std::cin >> userInput) {
    sum += userInput;
    std::cout << "The sum of inputs is " << sum << std::endl;
  }
  return 0;
}
```

I/O - Reading and writing files

- Reading and writing files is done using the `ifstream` and `ofstream` classes defined in the `fstream` library. The following program reads numbers from a file (input.txt) and prints the sum to another file (output.txt).

```
#include <iostream> //For cout
#include <fstream> //For ifstream and ofstream

int main() {
  std::ifstream inFile("input.txt"); //Name of the file to read from
  if(!inFile) {
    std::cout << "Error: could not read from file input.txt" << std::endl;
    return 1; //A nonzero return value indicates failure
  }
  double variable = 0.;
  double sum = 0.;
  while(inFile >> variable) { //Read numbers until we hit the end of file
    sum += variable;
  }
  inFile.close();

  std::ofstream outFile("output.txt");
  if(!outFile) {
    std::cout << "Error: could not write to file output.txt" << std::endl;
    return 1; //A nonzero return value indicates failure
  }
  outFile << sum << std::endl;
  outFile.close();
  return 0;
}
```

Writing a C++ class

- A class is a container for data and functions. This simple class stores coordinates. It has two member variables; an x and a y coordinate.
- You can create many `coords`, e.g (2,5) and (1,1). They have the same type but different internal states. An instance of the class is called an *object*.
- The constructor creates new objects. The destructor cleans up when an object is destroyed. We'll talk more about these important functions later.

```
class coords { //Here I declare a class of type "coords"
  public:
  coords(int xCoord, int yCoord); //Constructor. Call to create an instance of the class.
  ~coords(); //Destructor. Gets called when an instance of the class is destroyed.

  int x; //The only two member variables are the x and y coordinates
  int y;

  private: //This class has no private members
};

coords::coords(int xCoord, int yCoord) { //Simply store the user supplied coordinates
  x = xCoord;
  y = yCoord;
}

coords::~coords() {
  //There are no special tasks to perform when destroying a set of coordinates
}
```

Writing a C++ class

- Let's improve the coords class. We want the ability to change an existing coordinate. We also want the ability to work in a polar coordinate system.
- At this point we should divide the code into a header file (.h) for the class declaration and a source file (.cpp) for the implementation.
- The constructor now accepts a third argument `isPolar`. If it is not provided explicitly, `false` is used. This is called a default argument.

```
#ifndef BETTERCOORDS_H //This macro ensures that the .h file is only read once
#define BETTERCOORDS_H //It's fine if you don't understand how this works in detail

class betterCoords {
  public:
  betterCoords(double firstCoord, double secondCoord, bool isPolar = false);
  ~betterCoords() {};
  void setCartesian(double xCoord, double yCoord); //Set coordinates in cartesian space
  void setPolar(double rCoord, double phiCoord); //Set coordinates in polar space
  double x; //Cartesian coords
  double y;
  double r; //Polar coords
  double phi;

  private:
  void transformToCartesian(); //Helper functions to transform between coordinate systems
  void transformToPolar();
};
#endif //This ends the ifndef macro
```

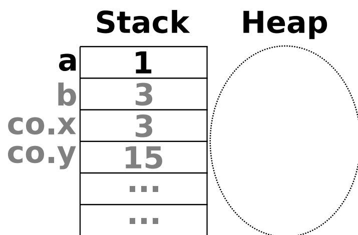
Writing a C++ class

```
#include "betterCoords.h" //Include the class declaration
#include <cmath> //For sqrt, sin, cos and atan2
using namespace std;

betterCoords::betterCoords(double firstCoord, double secondCoord, bool isPolar) {
  if(isPolar) setPolar(firstCoord, secondCoord); //The user supplied polar coordinates
  else setCartesian(firstCoord, secondCoord); //The user supplied cartesian coordinates
}

void betterCoords::setCartesian(double xCoord, double yCoord) {
  x = xCoord; //Set cartesian coordinates
  y = yCoord;
  transformToPolar(); //Then calculate the equivalent polar coordinates
}

void betterCoords::setPolar(double rCoord, double phiCoord) {
  r = rCoord; //Set polar coordinates
  phi = phiCoord;
  transformToCartesian(); //Then calculate the equivalent cartesian coordinates
}

void betterCoords::transformToCartesian() {
  x = r*cos(phi);
  y = r*sin(phi);
}

void betterCoords::transformToPolar() {
  r = sqrt(x*x + y*y);
  phi = atan2(x, y);
}
```

The stack and the heap

- The memory available for a program to use (at least as far as we're concerned) is made up of two areas - The stack and the heap.
- The stack is a small (megabytes), fixed size chunk of memory for local variables. All examples so far have used only the stack.
- When a variable on the stack falls out of scope, it is deallocated. You don't have to worry about memory management with the stack.
- The stack is small, so it overflows if you put too many things on it. But don't worry - This typically only happens due to bugs (e.g an infinite loop).

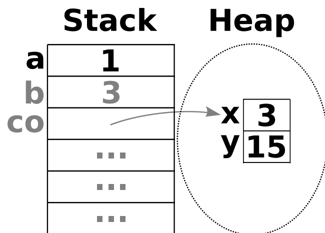
```
#include "coords.h"

void makeCoordinates(int b) {
  coords co(b, b*5);
}

int main() {
  int a = 1;
  makeCoordinates(a + 2);
  //Grayed out variables have now been deallocated
  return 0;
}
```



The stack and the heap

- The heap is a large pool of memory that can grow dynamically.
- To put a variable on the heap, create it with the `new` operator. This operator returns a *pointer* through which the variable is accessed.
- A pointer is really just an integer. The number corresponds to a memory address. The pointer *points* to that memory.
- Variables on the heap are never deallocated automatically. The memory must be freed manually using the `delete` operator.
- The pointer itself is on the stack and is deallocated automatically.

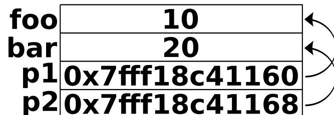
```
#include "coords.h"

void makeCoordinates(int b) {
  coords* co = new coords(b, b*5);
}

int main() {
  int a = 1;
  makeCoordinates(a + 2);
  //Grayed out variables have now been deallocated
  return 0;
}
```



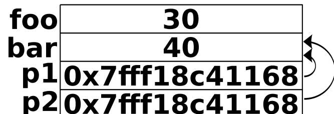
Pointers and references

- A pointer can point anywhere in memory, both the stack and the heap.
- To declare that a variable is a pointer, put an asterisk (∗) after its type.
- To get the memory address of a variable, use the reference operator (&).
- If you have a pointer and you want the value that the pointer points to, use the dereference operator (∗). That's right - The asterisk has *two* uses!

```
int foo = 10; //Two regular variables
int bar = 20;
int* p1; //Two pointers to int
int* p2;
p1 = &foo; //p1 points to foo
p2 = &bar; //p2 points to bar
```




```
*p2 = 30; //bar = 30
*p1 = *p2; //foo = bar
p1 = p2; //p1 now points to bar
*p1 = 40; //bar = 40
```



- To access members of a class via pointer, use the arrow (->) operator.

```
betterCoords a(1, 1); //Regular object
a.SetCartesian(2, 2); //Access with dot
betterCoords* b = new betterCoords(1, 1); //Pointer to object
b->SetCartesian(2, 2); //Access with arrow. This is the same as (*b).SetCartesian(2, 2)
```

Vectors

- A `vector` is a sequential container that can change size dynamically.
- It is a *template class*. The `vector` type must be defined at compile time.
- Vectors are fast at element access and insertion/removal at the end.

```
#include <iostream> //For cout and cin
#include <vector>
using namespace std;

int main() {
  vector<int> vec; //Create a vector with base type int
  int input;
  while(cin >> input) vec.push_back(input); //Store each input
  for(size_t i = 0; i < vec.size(); ++i) cout << vec.at(i) << endl; //Print them back
  return 0;
}
```

- Use `at` to access individual elements. It's also possible to use `[]`. **Never do this!** There is no bounds checking at run time. Your bugs will go unnoticed.

```
vector<int> vec; //Create an empty vector
cout << vec[3] << endl; //Index is out of bounds. Your program will happily print garbage
cout << vec.at(3) << endl; //Using at produces an error at run time, exposing your bug
```

Arrays

- An array is a fixed-size sequential container used extensively in C. Never use arrays in C++. Use vectors instead. Here's a teaser on why arrays are evil.

```
#include <iostream> //For cout and cin
using namespace std;

int main() {
  const int length = 10; //The length must be known at compile time
  int arr[length]; //This array is fixed-size
  int input;
  int pos = 0; //An array doesn't know its own size or how many elements it contains
  while(cin >> input) {
    arr[pos] = input;
    if(pos == length) break; //Remember that the array can't grow, so this is our limit
    ++pos; //We have to keep track of the position
  }
  for(int i = 0; i < pos; ++i) cout << arr[i] << endl; //No at(), easy to make a mistake
  return 0;
}
```

- Arrays allocated on the heap are deleted with the `delete[]` operator.

Strings

- A string is a sequence of characters, implemented by the `string` class.

```
#include <iostream> //For cout and cin
#include <string>
using namespace std;

int main() {
  string str("It's dangerous to go alone, take this!");
  size_t pos = str.find("take"); //Position in string where "take" is found

  cout << str.substr(0, 18) << str.substr(pos) << endl;
  return 0; //It's dangerous to take this!
}
```

- A C-string is a NULL terminated array of characters used extensively in C.
- C-strings are evil for the same reasons arrays are evil. Use strings instead.
- To get the C-string representation of a `string`, use the `c_str()` function.

```
char cString[10] = "Test"; //Contains 5 characters including the terminating \0
const char* cStringPtr = "Test"; //Pointer to string literal, can't be modified
string cppString("Test"); //Using cppString.c_str() returns const char*
```

Lists, Pairs

- A `list` is a container with fast element insertion and removal.
- Unlike vectors, elements in a `list` have no absolute position. Use an `iterator` to loop through them. Iterators act similarly to pointers.

```
#include <iostream> //For cout and cin
#include <list>
using namespace std;

int main() {
  list<int> lst; //List with base type int
  lst.push_back(10); //Insert some elements, then iterate over the list and print them
  lst.push_back(15);
  for(list<int>::iterator it = lst.begin(); it != lst.end(); ++it) cout << *it << endl;
  return 0;
}
```

- A `pair` is a simple container that stores two values.

```
#include <iostream> //For cout and cin
#include <utility> //For pair
using namespace std;

int main() {
  pair<int, double> p(5, 3.14); //A pair of int and double
  cout << "The pair is " << p.first ", " << p.second << endl;
  return 0;
}
```

Sets

- A set is a container that stores unique objects. If a set already contains a certain element, adding that element again does nothing. Sets are ordered.
- Adding/removing elements takes logarithmic time, which is relatively slow.
- Searching also takes logarithmic time - This is as fast as a search can get!

```
#include <iostream> //For cout and cin
#include <set>
using namespace std;

int main() {
  set<int> s; //Set with base type int
  s.insert(7); //Add some elements. The order in which they are added doesn't matter.
  s.insert(1);
  s.insert(5);
  for(set<int>::iterator it = s.begin(); it != s.end(); ++it) { //Traverse with iterator
    cout << *it << endl; //Prints 1, 5, 7
  }
  if(s.count(8)) cout << "The set contains the number 8" << endl; //Search in the set
  return 0;
}
```

Maps

- A map is an associative container that stores key/value pairs. A key can not be inserted twice, but the value of an existing key can be changed.

```
#include <iostream> //For cout and cin
#include <string>
#include <map>
#include <utility> //For make_pair
using namespace std;

int main() {
  map<string, int> pBook; //Map associating strings to ints. It's a phone book!
  pBook.insert(make_pair("Reginald", 123)); //Pairs can be inserted in various ways
  pBook.insert(pair<string, int>("Marmaduke", 456));
  pBook["Bobby Floyd"] = 789;

  map<string, int>::iterator it = pBook.find("Bruce Lee"); //How to search a map
  if(it != pBook.end()) cout << it->first << "has number " << it->second << endl;
  pBook["Reginald"] = pBook["Jim Bob"]; //Beware of using [] - Jim Bob is now in the book

  for(map<string, int>::iterator it2 = pBook.begin(); it2 != pBook.end(); ++it2) {
    cout << it2->first << " - " << it2->second << endl; //Print everyone in the book
  }
  return 0;
}
```

```
Bobby Floyd - 789
Jim Bob - 0
Marmaduke - 456
Reginald - 0
```

Pass by value, reference or pointer

- When calling a function, you are really passing *copies* of all the arguments.
- If you want to change the passed values, you must use references or pointers.

```
int x = 1;
int y = 2;

void swapByValue(x, y); //This will NOT swap the values!
void swapByReference(x, y); //This will work. Using references is recommended.
void swapByPointer(&x, &y); //This will work, but don't use pointers unless necessary.
```

```
void swapByValue(int a, int b) { //a and b are copies of x and y
  int temp = a; //Whatever we do here has no effect on the original x and y
  a = b;
  b = temp;
}
```

```
void swapByReference(int& a, int& b) { //a and b are references to x and y
  int temp = a; //For all intents and purposes, they ARE x and y
  a = b;
  b = temp;
}
```

```
void swapByPointer(int* a, int* b) { //a and b are pointers to x and y
  int temp = *a; //Not safe - What if they are NULL pointers? Use references instead.
  *a = *b;
  *b = temp;
}
```

Constructors and Destructors

- Constructors are called to create new instances of a class. They should initialize all member variables of the class. In general they accept arguments.

```
coords::coords(int xCoord, int yCoord) {
  x = xCoord; //Use the supplied values
  y = yCoord;
}
coords myCoords(2, 5); //How to invoke the constructor
```

- A constructor that takes no arguments is called a *default* constructor.
- Always write one. It is often invoked automatically, e.g vector<coords>(5).

```
coords::coords() {
  x = 0; //Choose a reasonable default value
  y = 0;
}
coords myCoords(); //Will be (0,0)
```

- To create copies of other objects, write a *copy* constructor.

```
coords::coords(coords& toCopy) {
  x = toCopy.x; //Copy values from the other object
  y = toCopy.y;
}
coords myCoords(existingCoords); //Initialize as copy of existingCoords
coords myCoords = existingCoords; //These two lines are equivalent
```

Constructors and Destructors

- A class must have a non-copy constructor. If you do not write one, the compiler automatically generates a default constructor with an empty body.
- A class must have a copy constructor. If you do not write one, the compiler generates one that performs a member-wise copy (aka a *shallow* copy).
- Consider this `line` class. Note that it stores *pointers* to coordinates.

```
class line {
  public:
  line(double x1, double y1, double x2, double y2); //Constructor
  ~line(); //Destructor
  betterCoords* start; //For some reason, we have chosen to store pointers to the coords
  betterCoords* stop;
};
```

- A shallow copy copies the pointers rather than what they point to.

```
line::line(line& toCopy) { //This is a shallow copy
  start = toCopy.start;
  stop = toCopy.stop; //Changing toCopy will affect line. We don't want this!
}

line::line(line& toCopy) { //After a deep copy, line and toCopy are independent
  start = new betterCoords(toCopy.start->x, toCopy.start->y);
  stop = new betterCoords(toCopy.stop->x, toCopy.stop->y);
}
```

Constructors and Destructors

- The destructor is automatically called when an object is destroyed.
- It should delete objects on the heap and perform any other cleanup tasks.
- The compiler generates an empty destructor if you don't write one yourself.

```
line::~line() {
  if(start) { //Safeguard against deleting NULL pointers
    delete start; //This destroys the object pointed to by start
    start = 0; //Not necessary here but good practice in more complicated cases
  }
  if(stop) {
    delete stop;
    stop = 0;
  }
}
```

- Once all pointers to a heap allocated variable are lost, it can't be deleted.
- This is called a *memory leak*. Here's a good rule of thumb:
- Every call to `new` should be matched by exactly one call to `delete`.

```
for(int i = 0; i < 10; ++i) { //A simple memory leak
  betterCoords* coordsPtr = new betterCoords(0, 0);
}
```

Initialization lists

- Consider this line class. How should the constructor be implemented?

```
class line {
  public:
  line(double x1, double y1, double x2, double y2);
  ~line();
  betterCoords start; //These aren't pointers this time
  betterCoords stop;
};
```

- Here `start` and `stop` are initialized with their default constructors. They are then assigned new values. But that default initialization is a waste of CPU!

```
line::line(double x1, double y1, double x2, double y2) {
  start = betterCoords(x1, y1); //Members are initialized using the default constructor
  stop = betterCoords(x2, y2); //The (compiler provided) assignment operator makes a shallow copy
}
```

- Now `start` and `stop` are initialized with their usual constructors. This is faster, and actually *required* when initializing variables that are `const`.

```
line::line(double x1, double y1, double x2, double y2) : start(x1, y1), stop(x2, y2) {
  //Note that the list order doesn't matter. Start will always be initialized first.
}
```

Command line parameters

```
#include <iostream> //For cout

int main(int argc, char* argv[]) {
  std::cout << "Received " << argc << " parameters:" << std::endl;
  for(int i = 0; i < argc; ++i) {
    std::cout << argv[i] << std::endl;
  }
  return 0;
}
```

- You can pass parameters to a program via command line. They arrive as C-strings contained within an array (can you tell that it's copied from C?)
- The first parameter is always the name of the program. Let's say, for the sake of example, that it's called 'commandLineParams'.
- Here is how it would look if built and run from a terminal.

```
$ g++ -o commandLineParams commandLineParams.cpp
$ ./commandLineParams abc 123 -bla --bla
Received 5 parameters
./commandLineParams
abc
123
-bla
--bla
```

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!
```

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.

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. As usual, they are evil and should not be used.

```
//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.

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);
}
```

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+
```