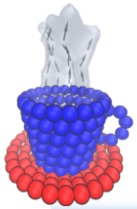
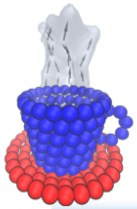


Object-oriented Programming in C++

Axel Arnold Olaf Lenz

Institut für Computerphysik
Universität Stuttgart

March 17-21, 2014



Outline

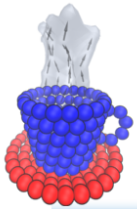
- Definition of classes
- Lifetime of objects
- Constructor, destructor
- Default and copy constructor
- Encapsulation: private, public
- Friends
- Function and operator overloading
- Inheritance
- Protected encapsulation
- Polymorphism and virtual functions
- Abstract base classes and pure virtual functions
- Multiple inheritance

Definition of classes

Player
playerNum : int name : string
setPlayerNum(playerNum : num) getPlayerNum() : int doNextMove(piece : Piece, event : Event, out message : string)

becomes

```
class Player {
public:
  int playerNum;
  string name;
  // setter and getter for the player number
  void setPlayerNum(int playerNum);
  int getPlayerNum();
  // perform the next move
  void doNextMove(Piece &piece, Event event, string &msg);
};
```

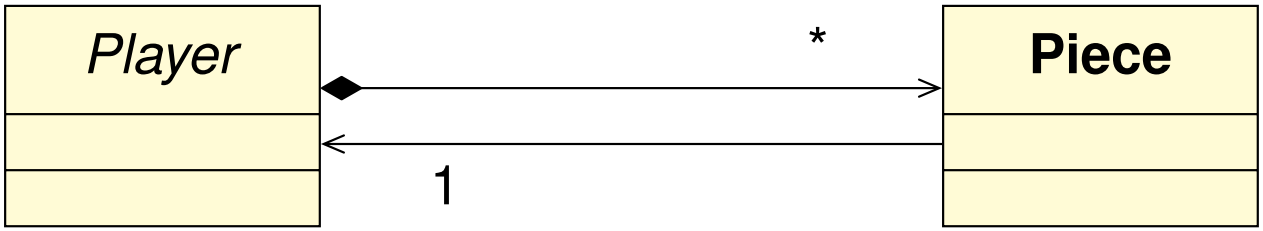


Definition of classes

```
class Player {
public:
  int playerNum;
  string name;
  // setter and getter for the player number
  void setPlayerNum(int playerNum);
  int getPlayerNum();
  // perform the next move
  void doNextMove(Piece &piece, Event event, string &msg);
};
```

- **remember semicolon at the end of the definition!**
- **`public`** will be explained later
- output parameters translate to references (`message`)
- no separator between member functions and variables required
- good style to separate them

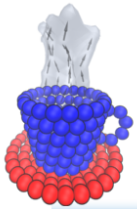
(Forward) declaration



becomes

```
class Player; // <-
class Piece {
  Player &player;
};
class Player {
  vector< Piece* > pieces;
};
```

- classes can be (forward) **declared** (here: `Player`)
- necessary if `Piece` uses `Player` and vice versa
- ok as long as only addresses are needed (pointers or references)
- usually, forward declare higher level classes

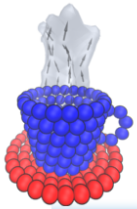


Declaration of member functions

Player.hpp

```
#ifndef PLAYER_HPP
#define PLAYER_HPP
class Player {
  void doNextMove(Piece &piece, Event ev, string &msg);
  int getPlayerNum() { return playerNum; }
};
#endif
```

- member functions are declared inside the class definition
- definition of *short* functions in class (**`inline`**)
- class definitions are usually placed in header file '*class*.hpp'
- preprocessor guards to avoid double definition of class



Definition of member functions

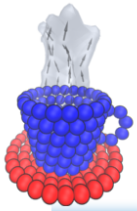
Player.hpp

```
class Player {
  void doNextMove(Piece &piece, Event ev, string &msg);
  int getPlayerNum() { return playerNum; }
};
```

Player.cpp

```
void Player::doNextMove(Piece &piece, Event ev, string &msg) {
  piece.tryStep(ev, msg);
}
```

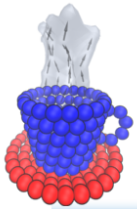
- full member identifier: `class::member`
- only one definition per member function (including inline def's)
- member function definitions in separate cpp-file '*class*.cpp'



Scope of member functions

```
class Player {
  Piece *currentPiece;
  void doNextMove(Piece &piece, Event ev, string &msg) {
    currentPiece = &piece;
    if (piece.tryStep(ev, msg))
      currentPiece->doStep(ev, msg);
  }
};
```

- call member functions by `object.function()`
- also access to member variables, if public
- compare C struct syntax
- pointers can be dereferenced using '->' like in C
- only almost true due to operator overloading (later)



Referring to yourself: this

```
void Piece::register() {
  board.addPiece(this);
}
void Piece::escape() {
  board.removePiece(this);
}
```

How can an object refer to “itself”?

`Piece` registers itself with the board

Answer

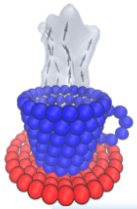
In a member function, **`this`** always points to the current object

- **`this`** is a pointer, although guaranteed to be valid
- there is no reference to the current object, use **`*this`** if necessary

Creation and life time of objects

```
int test() {
  for (int i = 0; i < 10; ++i) {
    Piece piece;
    piece.register();
    // here, piece is destroyed automatically
  }
}
```

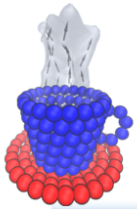
- objects are created like variables in C by giving type and name
- ...and die at the end of the scope (code block)
- exception: static variables like in C
- therefore, after the loop above, not a single piece exists



Creation and life time of objects

```
int test() {
  for (int i = 0; i < 10; ++i) {
    Piece *piecePtr = new Piece;
    piecePtr->register();
    delete piecePtr;
    // without the older pieces remain, inaccessible
  }
}
```

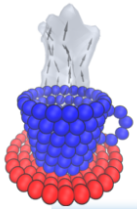
- objects can also be created by **`new`**
- these live till they are destroyed by explicitely calling **`delete`** *once*
- there should be *one* responsible object for destroying, the **owner**
- more than one owner leads to segmentation faults
- no owner (and therefore no **`delete`**) to memory leaks
- in OOP, ownership frequently changes, making things difficult



Constructor

```
class Game {
  Board* board;
  vector< Player* > players;
public:
  Game(int numPlayers = 2): board(0) {
    for (int i = 0; i < numPlayers; ++i)
      players.push_back(new Player);
  }
};
```

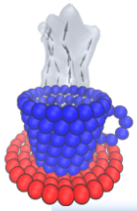
- constructor is a function initializing an object
- has the same name as the classs (“Class::Class”)
- can take arguments that specify what to create
- there can be several constructors (see overloading)
- initialize member variables after colon (here: `board`)
- multiple variables separated by comma



Destructor

```
class Game {
  Board* board;
  vector< Player* > players;
public:
  ~Game() {
    for (auto player: players) delete player;
    delete board;
  }
};
```

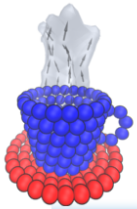
- destructor is called right *before* unallocating the memory
- should clean up
- in particular destroy all owned objects
- inform other objects that have pointers to this object
- the destructor cannot take arguments



Default and copy constructor

```
class Game {
public:
  Game()       { /* call default constructors of all objects */ }
  Game(const Game &src)  { /* use copy constructors instead */ }
};
```

- two *predefined* constructors:
  - Default constructor `Class::Class()`
    - only defined if no user-defined constructor declared
    - initializes all objects using their default constructor
    - plain old data types (char, int, ...) are uninitialized
  - Copy constructor `Class::Class(const Class &src)`
    - copies all member variables from object `src`
    - for pointers often a bad idea - duplicates ownership
- to avoid automatic generation, declare constructors *private*



Static class members

```
class CountOccurance {
  static int cnt;
public:
  CountOccurance() { ++cnt; }
  ~CountOccurance() { --cnt; }
  static int getCount() { return cnt; }
};
int CountOccurance::cnt = 0;
int main() {
  new CountOccurance();
  cout << CountOccurance::getCount() << endl;
  return 0;
}
```

- class members (functions and variables) can be static
- variable needs to be defined once somewhere (not in header!)
- static member functions are called with *class* prefix

Function overloading

Tile
tryStep(piece : Piece, how : Event, out message : string) *tryStep(monster : Monster, how : Event, out message : string)*

becomes

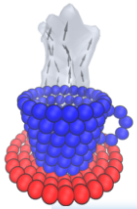
```
class Tile {
public:
  void tryStep(Piece &piece, Event how, string &message);
  void tryStep(Monster &monster, Event how, string &message);
};
```

- several methods with same name but different signature
- signature is formed by the types of all taken arguments
- return value is *not* part of the signature
- also global (C-style) functions can be overloaded

Default arguments

```
class QGameBoard
{
public:
  QGameBoard(int tileSize = 20, QWidget *parent = 0);
  void setTileSize(int sizeX, int sizeY = 0);
};
```

- another kind of function overloading are default arguments
- values get predefined values if not specified when calling
- always starts from the back:
  `QGameBoard(30)` is ok: `parent = 0`
  `QGameBoard(otherWidget)` not: `parent = 0`, the pointer `otherWidget` is cast into an integer for `tileSize`
- here, this also overloads the default constructor `QGameBoard()`



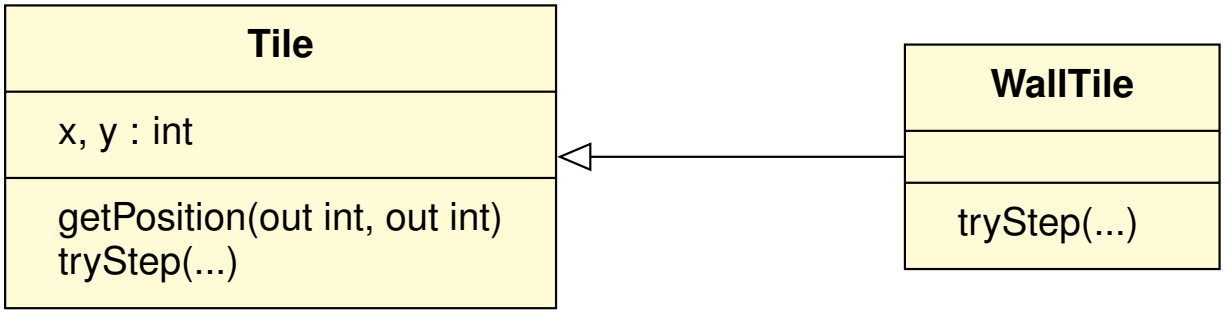
Operator overloading

- most operators can be overloaded
- abuse can lead to *big* confusion
- obviously relies heavily on signatures

Some examples:

```
// output using iostream
ostream &operator<<(ostream &stream, const Class &a);
// assignment of any type:
Class &Class::operator=(OtherClass b);
// sum (product, difference,... analogously):
RetClass Class::operator+(OtherClass b);
// ++object:
Class &Class::operator++();
// object++ (yes, they can differ):
Class Class::operator++(int); // int is a dummy argument!
// for objects representing functions:
RetClass Class::operator()(ParamClass a, ParamClass b, ...);
```

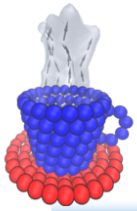
Inheritance



becomes

```
class Tile {
  int x, y;
  void getPosition(int x, int y);
  virtual void tryStep(Piece &piece, Event how, string &message);
};
class WallTile : public Tile {
  virtual void tryStep(Piece &piece, Event how, string &message);
};
```

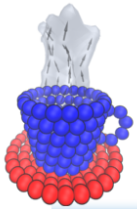
- derived classes declare super classes after class name and colon
- getPosition also exists in WallTile (**inherited** from `Tile`)



Constructor and base classes

```
class Tile {
public:
  Tile();
  ~Tile();
};
class WallTile: public Tile {
public:
  WallTile(): Tile() { }
  WallTile(int) { }
  ~WallTile();
};
```

- base classes are initialized in constructor following colon (like member variables)
- using default constructors can be omitted (`WallTile(int)`)
- arguments to constructors can be computed freely
- base classes are automatically destroyed

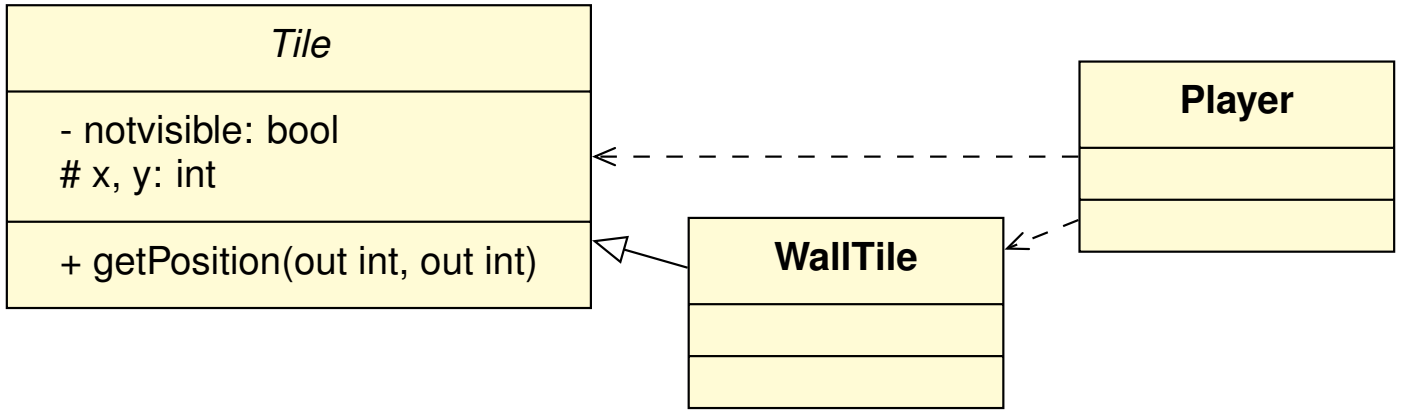


Constructor and base classes

```
class Tile {
public:
  Tile();
  ~Tile();
};
class WallTile: public Tile {
public:
  WallTile(): Tile() { }
  WallTile(int) { }
  ~WallTile();
};
```

- order follows matryoshka principle: from general to special
- base class constructors come *first*
- base class destructors come *last*

Encapsulation: private, public, protected



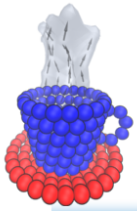
becomes

```
class Tile {
private:
  bool notvisible;
protected:
  int x, y;
public:
  void getPosition(int x, int y);
};
class WallTile : public Tile { };
```

Encapsulation: private, public, protected

```
class Tile {
private:
  bool notvisible;
protected:
  int x, y;
public:
  void getPosition(int x, int y);
};
class WallTile : public Tile { };
```

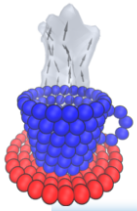
- **private** members are only accessible to members of the class
- **protected** is also accessible to members of derived classes
- **public** members are visible to all
- no difference between function and variable members
- a non-public destructor hinders destroying objects — rarely a good idea



Encapsulation: private, public, protected

```
class Tile {
private:
  bool notvisible;
protected:
  int x, y;
public:
  void getPosition(int x, int y);
};
class WallTile : public Tile { };
```

- `notvisible` is only accessible to `Tile` instances (**private**)
- `x` and !y! are accessible to instances of classes `Tile` and `WallTile` (**protected**)
- `getPosition` is visible to all, including e.g. `Player` instances (**public**)



Encapsulation: private, public, protected

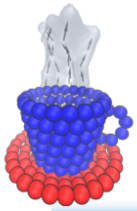
```
class Tile {
private:
  bool notvisible;
protected:
  int x, y;
public:
  void getPosition(int x, int y);
};
class WallTile : public Tile { };
```

- base classes also have a visibility (private, protected or public)
- most stringent visibility applies
- private members never become protected or public
- protected members never become public

Encapsulation: private, public, protected

```
class Tile {
private:
  bool notvisible;
protected:
  int x, y;
public:
  void getPosition(int x, int y);
};
class WallTile : public Tile { };
```

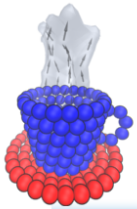
- objects are always also base class objects
  (a `WallTile` is a `Tile`, a `Mammal` an `Animal`)
  $\Longrightarrow$ **base class (interface) should be public**
- ... if not strong reasons against
- Qt: $\approx$ 2300 public inheritances, $\approx$ 30 private and $\approx$ 10 protected



Friends

```
class Hidden {
private:
  Hidden();
  Hidden(const Hidden &);
  ~Hidden();
  friend ostream &operator<<(ostream &, const Hidden &);
  friend class HiddenFactory;
};
```

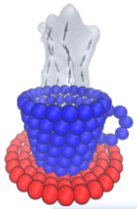
- **`friend`** overrides any visibility restrictions
- friends are declared in the class definition
  $\Longrightarrow$ you cannot inject friends
- friends can be classes or (member) functions
- common use is allowing the <<-operator to access inner parts
- here, it enforces to use `HiddenFactory` to create objects



Polymorphism: virtual functions

```
class Tile {
  virtual string whatAmI() { return "tile"; }
};
class FloorTile: public Tile {
  virtual string whatAmI() { return "floor tile"; }
};
void informMe(Tile &tile) {
  cout << "this is a " << tile.whatAmI() << endl;
}
```

- without virtual, `informMe` always says “this is a tile”
- **`virtual`** declares a function to be **polymorph**
- acts differently depending on the *actual* type of the object
- `Tile::whatAmI` is different for a `Tile` and a `WallTile`
- forgetting to define a virtual function leads to undefined references to the ‘vtable for class’

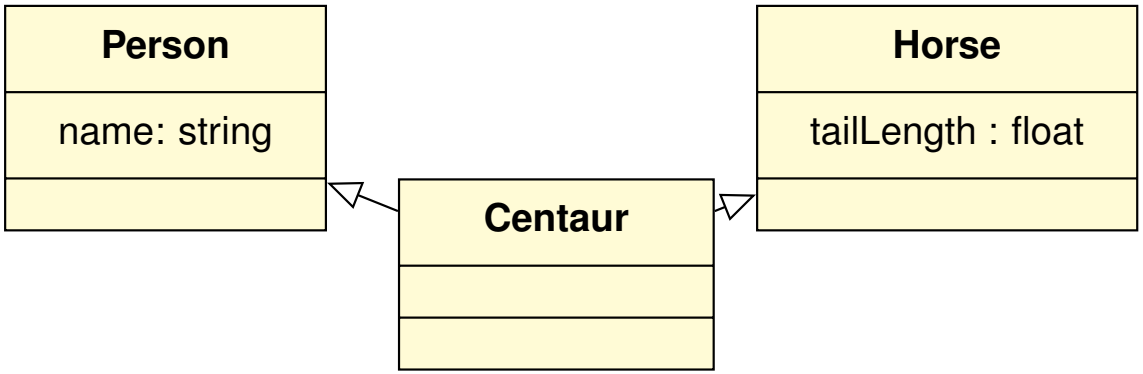


Pure virtual functions and abstract base classes

```
class Tile {
  virtual string whatAmI() = 0;
};
class FloorTile: public Tile {
  virtual string whatAmI() { return "floor tile"; }
};
int main() {
  // Error! Abstract class, cannot be instantiated
  Tile tile;
}
```

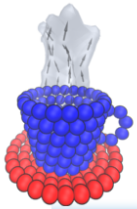
- purely virtual (abstract) functions are only defined for derived classes
- classes with purely virtual functions are **abstract** classes
- you cannot create objects of abstract classes

Multiple inheritance



becomes

```
class Person {
  string name;
  Person(const string &n): name(n) {}
};
class Horse {
  float tailLength;
  Horse(float t): tailLength(t) {}
};
class Centaur: public Person, public Horse {
  Centaur(const string &n, float t) : Person(n), Horse(t) {}
};
```

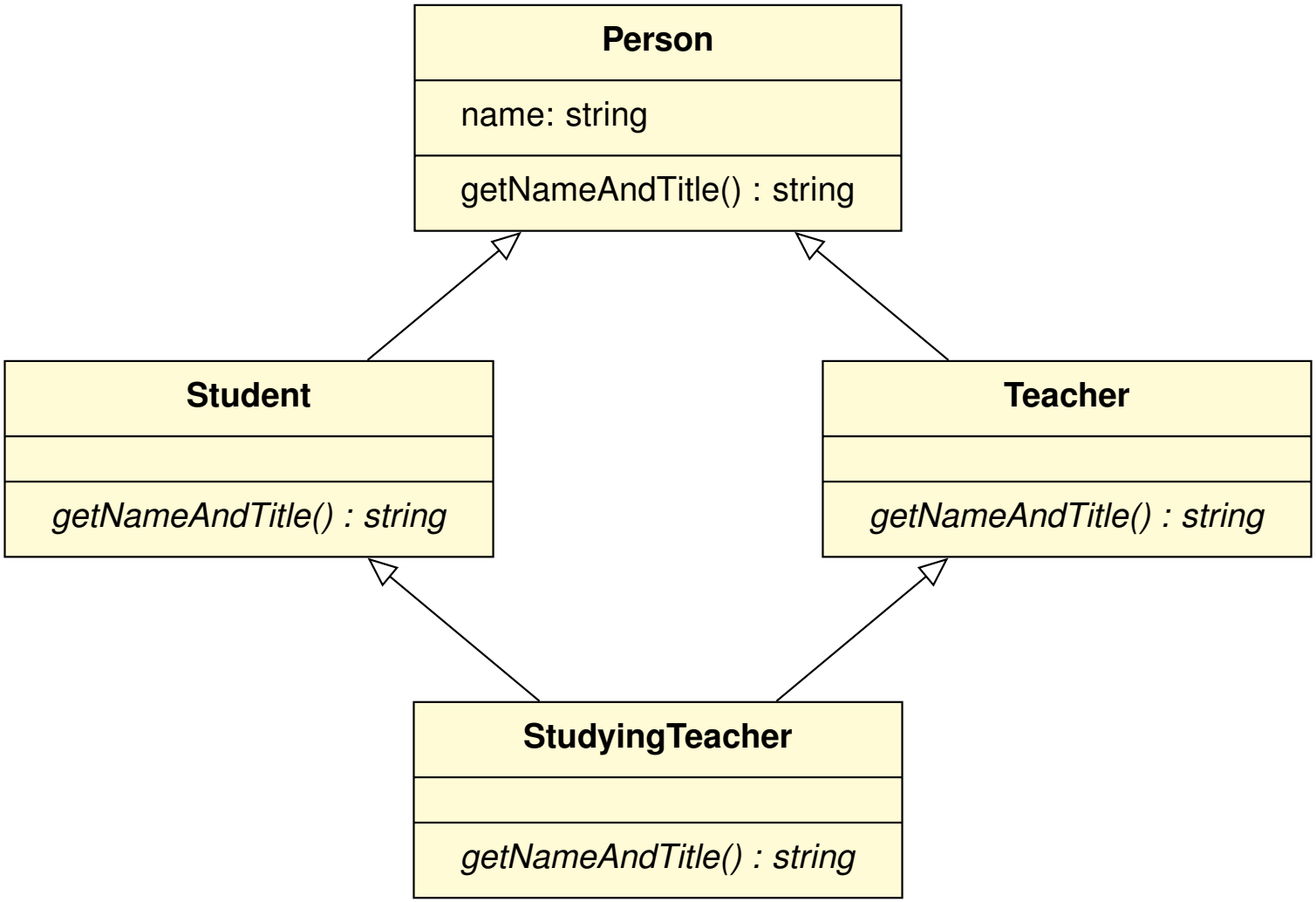


Multiple inheritance

```
class Person {
  string name;
  Person(const string &n): name(n) {}
};
class Horse {
  float tailLength;
  Horse(float t): tailLength(t) {}
};
class Centaur: public Person, public Horse {
  Centaur(const string &n, float t) : Person(n), Horse(t) {}
};
```

- multiple base classes are separated by commas
- each carries a visibility (public, protected or private)
- class has member variables and functions of both base classes

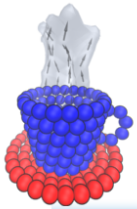
The diamond problem



The diamond problem — solution in C++

```
class Person {
protected: string name;
public:    Person(const string &_name): name(_name) { }
           virtual string getNameAndTitle();
};
class Teacher: virtual public Person {
public:    Teacher(const string &_name): Person(_name) { }
           virtual string getNameAndTitle();
};
class Student: virtual public Person {
public:    Student(const string &_name): Person(_name) { }
           virtual string getNameAndTitle();
};
class StudyingTeacher
  : virtual public Teacher, virtual public Student {
public:    StudyingTeacher(const string &n)
             : Person(n), Teacher(n), Student(n) {}
           virtual string getNameAndTitle();
};
```

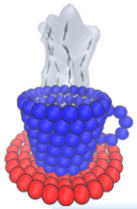
- declare base classes virtual
- requires also to initialize the common ancestor in the constructor
- initializiation of ancestor in intermediate classes is skipped



constant objects — const

```
const int theNumber = 42;
const char *nonConstantPtrToConstChar = "Hase";
nonConstantPtrToConstChar = "Igel";
const char * const constantPtrToConstChar = "Hase";
const QPointF constantPoint(2, 3);
```

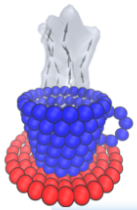
- **`const`** objects cannot be changed
- can be of any type (plain old data type, pointer, struct or class)
- the compiler generates code based on this assumption!
- be careful with pointers
  - **`const`** *on the left* binds to the data type pointed to — you cannot change the data, but point to different data
  - **`const`** *on the right* makes the address pointed to unchangeable
- C-string constants are of type **`const char`**, i.e. cannot be changed after compilation



constant objects — const

```
class Class {
public:
  const int myInitialValue;
  Class(int n): myInitialValue(n) { }
};
```

- classes can define constant member variables
- even if public, they cannot be changed by any object
- they have to be initialized in the initializer list of the constructor(s) (after the colon)



constant objects — const

```
class Class {
  int myNumber;
public:
  Class(int n) { myNumber = n; }
  void setMyNumber(int i) { myNumber = i; }
  int getMyNumber()  const { return myNumber; }
};
```

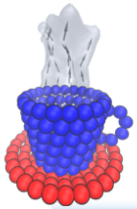
- classes need to be **const-aware** to make use of constant objects
- on constant objects, only member functions declared **`const`** can be called
  here, only `getMyNumber`*, but not* `setMyNumber`
- **`const`** functions can also called on non-const objects
- **`const`** member functions cannot change any member variables

constant objects — const

```
class Class {
public:
  void test()       { cout << "nonconst" << endl; }
  void test() const { cout << "const" << endl; }
};
Class changeable(1);
const Class unchangeable(2);
```

- **const** is part of the *signature*
- **const** and non-**const** versions of the same function can exist
- here, `changeable.test()` will output “nonconst”, `unchangeable.test()` will output “const”
- return values can differ (used frequently in STL)

```
class vector {
  iterator       begin();
  const_iterator begin() const;
```



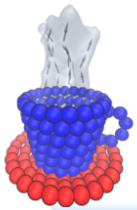
constant objects — const

Pros

- const-aware programs are less error-prone
- functions that should not change anything cannot do so
- allows the compiler to optimize more aggressively
- if writing libraries, const-awareness is necessary for others to be able to program const-aware

Cons

- hard to interface non-const-aware code (e.g. plain C)
- leads often to code duplication
- vector e.g. needs const and non-const iterators, which essentially do the same
- templates reduce programming overhead, but not code size



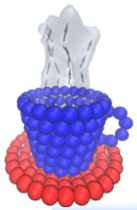
`dynamic_cast` and `static_cast`

```
class Tile {

};
class WallTile: public Tile { };

int main() {
  Tile *tile = new Tile;
  WallTile *wall = new WallTile;
  Tile *tilePtr = static_cast<Tile *>(wall);
  WallTile *wallPtr = static_cast<WallTile *>(tile);
}
```

- **`static_cast`** on class pointers only checks compatibility (target class should be super or derived class)
- ends badly if a base object gets cast to a derived object
- here, `wallPtr` should not be used!
- still better than C-like casting, which **you shall not use!**



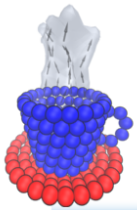
dynamic_cast and static_cast

```
class Tile {

};
class WallTile: public Tile { };

int main() {
  Tile *tile = new Tile;
  WallTile *wall = new WallTile;
  Tile *tilePtr = dynamic_cast<Tile *>(wall);
  WallTile *wallPtr = dynamic_cast<WallTile *>(tile);
}
```

- **dynamic_cast** checks full compatibility
- here, the cast to `WallTile` for `wallPtr` causes a compiler error
- without polymorphism only casting to base classes possible

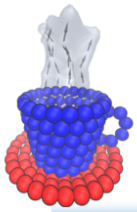


dynamic_cast and static_cast

```
class Tile {
  virtual void makeMePolymorph() {};
};
class WallTile: public Tile { };

int main() {
  Tile *tile = new Tile;
  WallTile *wall = new WallTile;
  Tile *tilePtr = dynamic_cast<Tile *>(wall);
  WallTile *wallPtr = dynamic_cast<WallTile *>(tile);
  if (wallPtr) use(wallPtr);
}
```

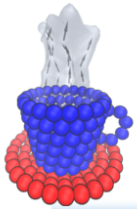
- on polymorphic classes (with at least one virtual function) **dynamic_cast** returns 0 if the cast is impossible
- here, the cast to `WallTile` for `wallPtr` is syntactically ok
- but `wallPtr` will be 0



Example of using `dynamic_cast`

```
for (auto it : content->items()) {
  QGraphicsTextItem *item =
    dynamic_cast<QGraphicsTextItem *>(it);
  if (item)
    item->setFont(QFont("Helvetica"));
}
```

- working with lists of different items (e.g. `QGraphicsScene`)
- all elements are pointers to a common base class (here: `QGraphicsItem`)
- for each item, you can check whether it is e.g. a `QGraphicsTextItem` and then change the font

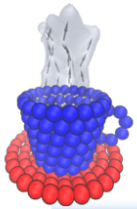


const – non-const conversion: `const_cast`

```
const char *unchangeable;
char *changeable =
          const_cast<char *>(unchangeable);

char *changeable2;
const char *unchangeable2 =
          const_cast<const char *>(changeable2);
```

- **`const_cast`** converts between const and non-const variables
- if `unchangeable` points to a true C-string, writing to `changeable` will still cause a segmentation fault
- only use: interfacing with C-libraries that are not const-aware
- you need to know that the function doesn't change anything



Automatic type conversion

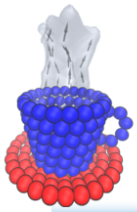
```
class String {
  string value;
public:
  String(const string &val): value(val) {}
};
int main()
{
  String s("Hallo");
}
```

- this code works due to **automatic type conversion**
- compiler sees that String's constructor takes a `string` instance
- it tries to convert the **`const char *`** to a string using any string's constructors

Automatic type conversion

```
class MyString {
public:
  MyString(const char *val);
};
class User {
public:
  User(MyString bla);
  User(string bla);
};
int main() {
  User user("Hallo");
}
```

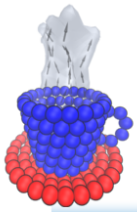
- creating `user` fails since the conversion is **ambigous**
- the intermediate class could equally be `MyString` or `string`
- both are accepted by the User class
- both can be constructed from **`const char *`**



Automatic type conversion

```
class MyString {
public:
  explicit MyString(const char *val);
};
class User {
public:
  User(MyString bla);
  User(string bla);
};
int main() {
  User user("Hallo");
  User user2(MyString("Welt"));
}
```

- **explicit** excludes constructors from consideration for implicit conversion
- you can use them by explicitely (!) calling the constructor
- solves the ambiguity



The auto data type

```
void print(const vector<int> &v) {
  for (auto value: v)
    cout << value << endl;
}
const auto theNumber = 5;
```

- C++11 adds the data type **auto**
- determines the data type from the right hand sides datatype
- works if the right hand side has a unique type
  - constants and other variables
  - return values of functions (return value cannot be overloaded!)
- in particular useful with range-based for (see STL, Olaf)