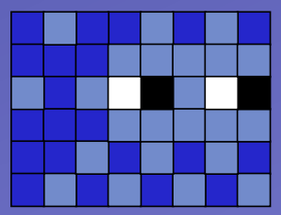
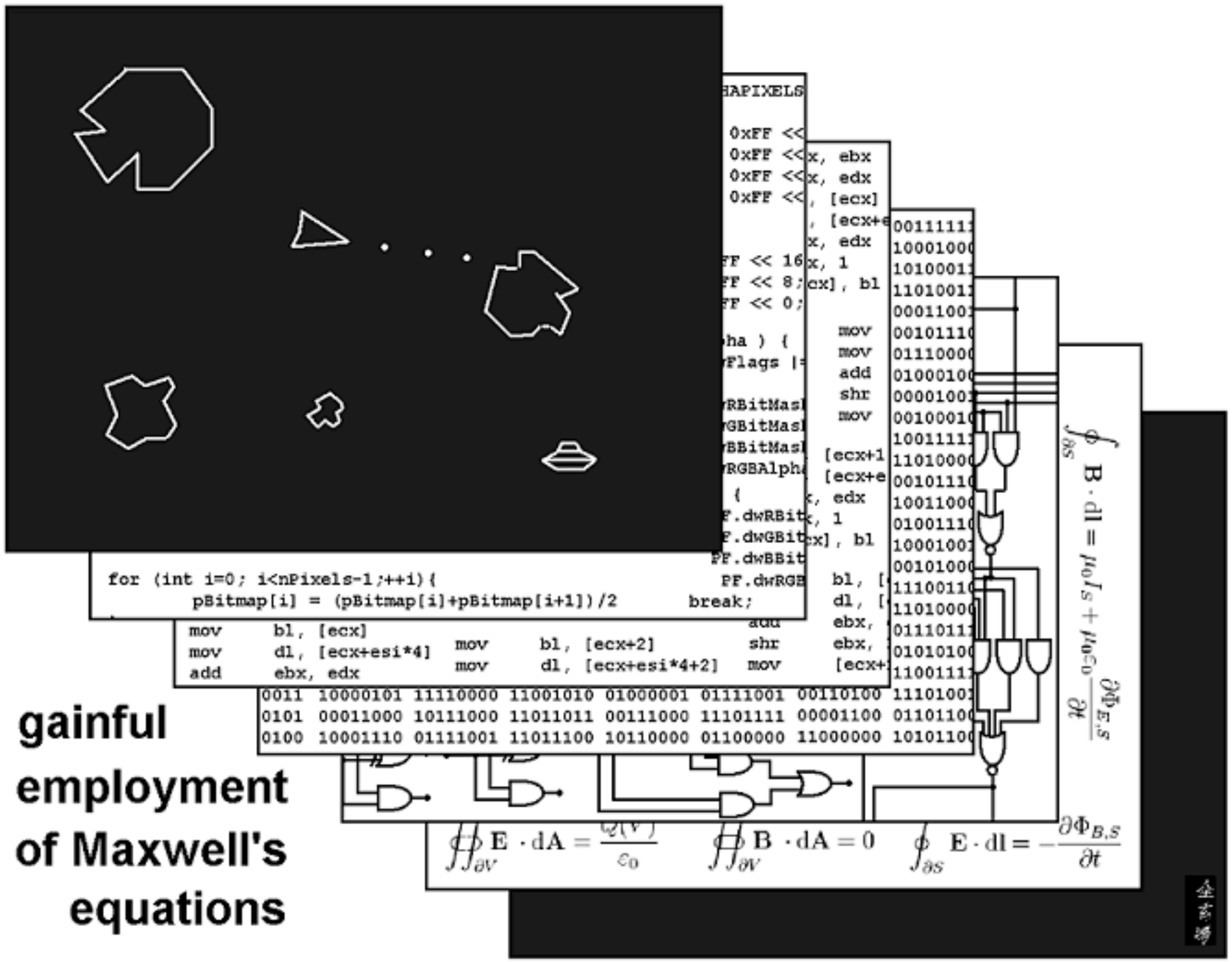
An intro to C++

Francesco Safai Tehrani
francesco.safaitehrani@roma1.infn.it



The root of all evil



http://abstrusegoose.com/307

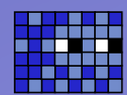
Software, as seen by a cat fancier

Courtesy of
Özgür Çobanoğlu

Non-hierarchical software !..

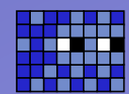
Object oriented programming

- From functions to objects
  - With functions you have code blocks that accept arguments and process them returning a response
    - code functions != mathematical functions
      - unless you're using Functional Programming...
  - With objects you have 'entities' that interact with other entities via well defined interfaces
    - Interfaces fully describe what an object can do
- Plenty of different OOPs
  - Two main ones: static and dynamic
    - Static: objects have a well defined nature which cannot change
    - Dynamic: the object's nature is detemined by its behavior



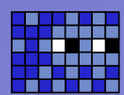
Object Oriented Programming

- OOP is a technology (actually, many technologies)
  - different languages implement (not so) slightly different OOPs
- Some of the concepts of OOP can be traced back to the ‘50s
  - MIT AED-0, Simscript, Sketchpad, ...
- The first language to explicitly use the concept of objects was **Simula**, a discrete event simulation language, in the early ‘60s
  - two versions: Simula I and Simula 67
  - Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in Oslo
- The term OOP was introduced with the language **Smalltalk**, developed at Xerox PARC
  - two main versions: Smalltalk 72 and Smalltalk 80
  - they also invented the mouse and the mouse-driven GUI, the desktop metaphor
    - and WYSIWYG, Ethernet, laser printers, bitmap graphics, ...
  - then they kind of gave it away (or so the story goes)...



From C to C++

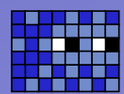
- C != C++
  - but this should be abundantly clear by now ;)
- C++ is one of the versions of the ‘C with objects’ paradigm
  - inspired by SIMULA 67
- another ‘C with objects’ takes inspiration from Smalltalk 80 instead...
  - Objective-C
    - if you use a Mac or an iSomething you are running Objective-C software
- both Smalltalk 80 and Simula 67 are still used...
  - but not very commonly
- Who uses what? Let’s see some (simplified) examples:
  - Windows*, Android: C++ (and C)
  - Mac OS X, iOS: Objective-C (and C++ and C)
  - Linux kernel, Android kernel, Darwin kernel: C (and some assembler...)
- So... which one is best? ;)



C++ by examples

- I am not going to try to teach you all of C++
  - or even very little C++...
- I'll give some examples...
  - hopefully you'll be able to glimpse the truth from these

- I am NOT going to talk about complex stuff
  - just basic C++
    - "C++ for the masses"
- Lots of talking, little writing
  - there's plenty of good books out there... no need to replicate them



Talking about OOP

- To talk about OOP and explain how it works we will need a number of terms (buzzwords) that identify the elements of OOP
- OOP has historically always been buzzword-heavy:
  - class, object, instance, message, member variable, member function, interface, overloading, constructor,
    destructor, delegation, inheritance, specialization,
    generalization, abstraction, ownership, template,
    implementation, private, public, protected, friend,
    .....
- Most of these term have a well defined meaning...
  - in a specific language...
  - there might be subtle and not-so-subtle differences between languages
  - we'll keep it simple and just talk about C++
  - but, caveat emptor, what we say is only valid for C++
    - kinda.
- Concepts will be introduced in context
  - when possible...

1D CA

I will use this example in the C++ lecture, hence the longer description.

Let's give some nomenclature.

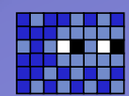
The cellular automaton lives on a playground of N spaces.
The playground is circular (its leftmost element and its rightmost element are next to each other).
Each 'space' in the playground is called a cell.
A cell can be dead (0) or alive (1).
Time flows in discrete steps. At each given step the cell state can change or remain the same.
The cell state changes depending on its own state and the state of its first neighbors (usually known as a neighborhood).

Each automaton is completely defined by its initial state and its evolution rules.

Since there are eight possible configurations for a neighborhood, each with a possible outcome of 0 or 1 in the next state, there is a total of 256 possible evolution rules.

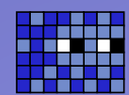
State	111	110	101	100	011	010	001	000
Outcome	0	0	1	1	0	1	0	0

As an example, this is rule 52 (2^2 + 2^4 + 2^5).



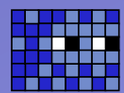
The implementation

- You should already be familiar with the problem and the related algorithms via the C exercise that we discussed in the previous lecture.
- 4 versions of the same program, with increasing OO complexity.
- The problem is EXTREMELY simple, and frankly C++ is a bit overkill for it...
  - ... but I hope it'll help you figure out a few things
- Once we lay down the infrastructure it will pretty much stay the same throughout the various versions...
- I have no plan, nor hope (nor desire) to try to cram all of C++ into a 1h lecture
  - also most people use a relatively small subset of C++
    - somebody called it C++--... ;)
- The code is written to be plain and understandable, which implies I didn't use many advanced features of C++
  - e.g. I use loops instead of iterators, no STLs, ...



Code structure

- I’ve broken down the problem into two separate entities:
  - the Playground class which abstracts the notion of the arena where the cells ‘live’
  - the Cell entity which abstracts ... well, the cell.
- In such a simple case, I expect most people would agree with me on my ‘design’
  - and still I expect plenty would disagree
    - the long debated issue: “should an event be an object?”
  - different people will design their programs differently...
    - experience, knowledge of the problem domain, technical proficiency, ...
  - There Is More Than One Way To Do It
    - and more than way that is ‘right’.



V. I

```
#ifndef _CELL_H_
#define _CELL_H_

using namespace std;

class Cell {
private:
  int state;
  int RuleSet[8];

public:
  Cell() { state = 0; }

  Cell(int aState): state(aState) {}

  Cell(int aState, int* aRuleSet) {
    state = aState;
    for(int i=0; i<8; i++) {
      RuleSet[i] = aRuleSet[i];
    }
  }

  virtual ~Cell() {}

  int evolve(Cell* neighbors[]);

  int getState() {
    return state;
  }

  int setState(int aState) {
    state = aState;
  }
};

#endif
```

Constructor: the method that is invoked when instantiating an object

Overloading: a kind of polymorphism where a method can have different signatures. The choice of the correct method to invoke is performed based on the invocation signature

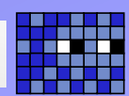
Overloaded constructors (ctors)

Destructor

Destructor: the method that is (implicitly) invoked when releasing an object

Virtual: a big can of worm that has to do with what method gets invoked when. (A bit) more on this later...

Destructors HAVE to be virtual, unless you really know what you're doing.



```cpp
#include "Cell.hh"

int Cell::evolve(Cell* neighbors[]) {
  int stateInfo = 4*(neighbors[0])->getState() + 2 * state + (neighbors[1])->getState();
  return RuleSet[stateInfo];
}
```

In C++:

.hh headers contain the interface of a class (also .H, .hxx, .hpp, ...)
.cc files contain the implementation of a class (also .C, .cxx, .cpp, ...)

Okay, what the heck is a class?
A class is a design entity representing a part that contributes to the solution of your problem. Different people have different classes (a.k.a. one man's constant is another man's variable).

A class is the blueprint of the actual entity (the project for a car versus the actual car) and as such has no state, nor behavior, that is to say it cannot store data nor answer to messages. (Although, in some languages classes are actually instances of entities calles metaclasses, making them objects... but not quite)

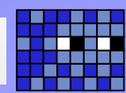
The operation of generating an object from a class is called instatiation.
Hence an object is an instance of a class, that is a real entity (i.e. one that has state and behavior) that conforms to the class interface.

A class, being a blueprint, does not have a state.
An interface fully defines what an object can do, or (using the messaging model) what messages it can answer.

The object is the core entity of C++ programming. A C++ programming can be seen as a 'network' of 'objects' which interact with each other exchanging 'messages'.

```
#ifndef _PLAYGROUND_H_
#define _PLAYGROUND_H_

#include "Cell.hh"

using namespace std;

class Playground {
private:
  Cell** currentArena;
  Cell** nextArena;
  Cell** tmpArena;
  int RuleSet[8];
  int rule;
  int size;

public:
  Playground(int aSize, int aRule) {
    size = aSize;
    rule = aRule;

    currentArena = new Cell*[size];
    nextArena    = new Cell*[size];

    createRuleSet();

    for(int idx=0; idx<size; idx++) {
      currentArena[idx] = new Cell(0, RuleSet);
      nextArena[idx]    = new Cell(0, RuleSet);
    }
    initCurrentArena();
  }

  virtual ~Playground() {
    delete[] currentArena;
    delete[] nextArena;
  }

  void createRuleSet();
  void initCurrentArena();
  void nextGeneration();
  void printArena();
};

#endif
```

```cpp
#include <iostream>
#include "Playground.hh"

using namespace std;

void Playground::createRuleSet() {
  for(int idx=0; idx<8; idx++) {
    RuleSet[idx] = (rule >> idx) & 1;
  }
}

void Playground::nextGeneration() {
  Cell* neighborhood[2];

  for(int idx=0; idx<size; idx++) {
    int pidx = (idx-1)<0?(size-1):(idx-1);
    neighborhood[0] = currentArena[pidx];
    neighborhood[1] = currentArena[(idx+1)%size];
    (nextArena[idx])->setState((currentArena[idx])->evolve(neighborhood));
  }
  tmpArena     = currentArena;
  currentArena = nextArena;
  nextArena    = tmpArena;
}

void Playground::initCurrentArena() {
  // impulse
  (currentArena[size/2])->setState(1);
}

void Playground::printArena() {
   for(int idx=0; idx<size; idx++) {
      if((currentArena[idx])->getState()) {
        cout << "o";
      } else {
        cout << " ";
      }
   }
   cout << endl;
}
```

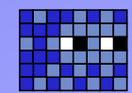



```cpp
#include<iostream>
#include "Cell.hh"
#include "Playground.hh"

#define PLAYGROUND_SIZE 80
#define GENERATIONS     100
#define RULE            30

int main (int argc, const char * argv[]) {
  Playground* myPlayground = new Playground(PLAYGROUND_SIZE, RULE);

  for(int idx=0; idx<GENERATIONS; idx++) {
    myPlayground->nextGeneration();
    myPlayground->printArena();
  }
  return 0;
}
```

Compile like this:

```
g++ -c Cell.cc
g++ -c Playground.cc
g++ -o ca1d ca1d.cc Cell.o Playground.o
```

V.2



```
#ifndef _ABSCELL_H_
#define _ABSCELL_H_

class AbsCell {

public:
  virtual ~AbsCell() {};

  virtual int evolve(AbsCell* neighbors[])=0;
  virtual int getState()=0;
  virtual int setState(int)=0;
};

#endif
```

purely virtual functions

This is an abstract class: a class that cannot be instantiated, since it's just an interface that contains no behavior. Real classes can be implemented from an abstract class via inheritance (also known as generalization/specialization).

Inheritance is an OOP mechanism that establishes an interface relationship between two classes. When class B 'inherits' from class A, their interfaces are the same, that is B can respond to all the messages A can respond to (but NOT vice versa...).

The way B behaves can (and in general will) be different from the way A behaves, that is B can specialize A's behaviors. B can also extend its interface with respect to A's one.

A classic example: the Rectangle class inherits from the Shape class which implements the abstract draw method. The Rectangle class can 'specialize' the draw method to draw a rectangle.

From the C++ standpoint, using purely abstract methods forces the developer to correctly implement the full interface, otherwise the compiler will produce some seriously nasty errors.

Pure abstract classes are sometimes called interfaces (e.g. in Java). Sometimes different terms are used: you inherit from a real class, but you implement an interface/abstract class.

```cpp
#ifndef _CELL_H_
#define _CELL_H_

#include <iostream>
#include "AbsCell.hh"

using namespace std;

class Cell: public AbsCell {
private:
  int state;
  int RuleSet[8];

public:
  Cell() { state = 0; }

  Cell(int aState): state(aState) {}

  Cell(int aState, int* aRuleSet) {
    state = aState;
    for(int i=0; i<8; i++) {
      RuleSet[i] = aRuleSet[i];
    }
  }

  virtual ~Cell() {}

  virtual int evolve(AbsCell* neighbors[]);

  virtual int getState() {
    return state;
  }

  virtual int setState(int aState) {
    state = aState;
  }
};

#endif
```

Inheritance: the class Cell inherits the interface (and the behavior if any) from AbsCell.
You can also say that Cell conforms to the AbsCell interface.

and here I specialize the evolve method defined in the AbsCell interface...

```
#ifndef _PLAYGROUND_H_
#define _PLAYGROUND_H_
#include "Cell.hh"

using namespace std;

class Playground {
private:
  AbsCell** currentArena;
  AbsCell** nextArena;
  AbsCell** tmpArena;
  int RuleSet[8];
  int rule, size;

public:
  Playground(int aSize, int aRule) {
    size = aSize;
    rule = aRule;

    currentArena = new AbsCell*[size];
    nextArena    = new AbsCell*[size];

    createRuleSet();

    for(int idx=0; idx<size; idx++) {
      currentArena[idx] = new Cell(0, RuleSet);
      nextArena[idx]    = new Cell(0, RuleSet);
    }
    initCurrentArena();
  }

  virtual ~Playground() {
    delete[] currentArena;
    delete[] nextArena;
  }

  void createRuleSet();
  void initCurrentArena();
  void nextGeneration();
  void printArena();
};
```

in this class I need to substitute most of the Cell variables with AbsCell variables

The reason will become clear later...

Liskov substition principle:
Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.

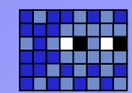
```cpp
#include <iostream>
#include "Playground.hh"

using namespace std;

void Playground::createRuleSet() {
  for(int idx=0; idx<8; idx++) {
    RuleSet[idx] = (rule >> idx) & 1;
  }
}

void Playground::nextGeneration() {
  AbsCell* neighborhood[2];

  for(int idx=0; idx<size; idx++) {
    int pidx = (idx-1)<0?(size-1):(idx-1);
    neighborhood[0] = currentArena[pidx];
    neighborhood[1] = currentArena[(idx+1)%size];
    (nextArena[idx])->setState((currentArena[idx])->evolve(neighborhood));
  }
  tmpArena     = currentArena;
  currentArena = nextArena;
  nextArena    = tmpArena;
}

void Playground::initCurrentArena() {
  // impulse
  (currentArena[size/2])->setState(1);
}

void Playground::printArena() {
   for(int idx=0; idx<size; idx++) {
      if((currentArena[idx])->getState()) {
        cout << "o";
      } else {
        cout << " ";
      }
   }
   cout << endl;
}
```

Okay, now which evolve method is being called?

The one from AbsCell or the one from Cell? Why? How does C++ do the right thing?

Virtualization and late binding!

V.3



Okay, great so far... but there are other CAs out there... what if I want to define a different kind of cell conforming to the AbsCell interface? Do I need to rewrite the Playground every time I have a different kind of cell?

Naturally not, and this question has an (almost) infinite amount of answers in terms of what you can do.

For this particular tasks I've chosen to use a template: a structure that has one or more parametric arguments which can be classes or types.

Templates are extremely powerful tools and constitute a kind of language within the language. I'm not even going to scrape its surface.

Just FYI, templates are the base elements of generic programming in C++... (http://en.wikipedia.org/wiki/Generic_programming)

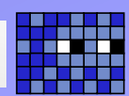
```
#ifndef _PLAYGROUND_H_
#define _PLAYGROUND_H_

#include "AbsCell.hh"

using namespace std;

template <class T> class Playground {
private:
  AbsCell** currentArena;
  AbsCell** nextArena;
  AbsCell** tmpArena;
  int RuleSet[8];
  int rule, size;

public:

  Playground(int aSize, int aRule) {
    size = aSize;
    rule = aRule;

    currentArena = new AbsCell*[size];
    nextArena    = new AbsCell*[size];

    this->createRuleSet();

    for(int idx=0; idx<size; idx++) {
      currentArena[idx] = new T(0, RuleSet);
      nextArena[idx]    = new T(0, RuleSet);
    }
    this->initCurrentArena();
  }

  virtual ~Playground() {
    delete[] currentArena;
    delete[] nextArena;
  }

  void createRuleSet();
  void initCurrentArena();
  void nextGeneration();
  void printArena();
};
```

Template definition: I'm informing the compiler that T is going to be a class.

The compiler knows NOTHING about T until I actually 'instantiate' the template with a real class

```
template <class T> void Playground<T>::createRuleSet() {
  for(int idx=0; idx<8; idx++) {
    RuleSet[idx] = (rule >> idx) & 1;
  }
}

template <class T> void Playground<T>::nextGeneration() {
  AbsCell* neighborhood[2];

  for(int idx=0; idx<size; idx++) {
    int pidx = (idx-1)<0?(size-1):(idx-1);
    neighborhood[0] = currentArena[pidx];
    neighborhood[1] = currentArena[(idx+1)%size];
    (nextArena[idx])->setState((currentArena[idx])->evolve(neighborhood));
  }
  tmpArena     = currentArena;
  currentArena = nextArena;
  nextArena    = tmpArena;
}

template <class T> void Playground<T>::initCurrentArena() {
  // impulse
  (currentArena[size/2])->setState(1);
}

template <class T> void Playground<T>::printArena() {
   for(int idx=0; idx<size; idx++) {
      if((currentArena[idx])->getState()) {
        cout << "o";
      } else {
        cout << " ";
      }
   }
   cout << endl;
}
#endif
```

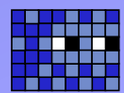

Playground.hh [template version, v3][2/2]



```cpp
#include<iostream>
#include "Playground.hh"
#include "Cell.hh"

#define PLAYGROUND_SIZE 80
#define GENERATIONS     100
#define RULE            30


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

  Playground<Cell>* myPlayground = new Playground<Cell>(PLAYGROUND_SIZE, RULE);

  for(int idx=0; idx<GENERATIONS; idx++) {
    myPlayground->nextGeneration();
    myPlayground->printArena();
  }

  return 0;
}
```

...and here I instantiate the template with T == 'Cell'...

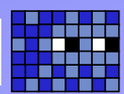
V.4



```cpp
#ifndef _ABSCELL_H_
#define _ABSCELL_H_

#include <iostream>

class AbsCell {

public:
  virtual ~AbsCell() {};

  virtual int evolve(AbsCell* neighbors[])=0;
  virtual int getState()=0;
  virtual int setState(int)=0;
  virtual void print()=0;
};

#endif
```

Responsibility: who should do what?
Shouldn't the AbsCell decide how to display itself?

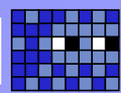
Somebody should, and everybody should be able to delegate to someone else, as long as somebody knows how to to do it (this is knowns as the perfect hierarchical principle).



```cpp
#ifndef _CELL_H_
#define _CELL_H_

#include <iostream>
#include "AbsCell.hh"

using namespace std;

class Cell: public AbsCell {
private:
  int state;
  int RuleSet[8];

public:
  Cell() { state = 0; }

  Cell(int aState): state(aState) {}

  Cell(int aState, int* aRuleSet) {
    state = aState;
    for(int i=0; i<8; i++) {
      RuleSet[i] = aRuleSet[i];
    }
  }

  virtual ~Cell() {}

  virtual int evolve(AbsCell* neighbors[]);

  virtual int getState() {
    return state;
  }

  virtual int setState(int aState) {
    state = aState;
  }

  virtual void print();
};

#endif
```

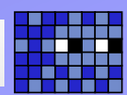
```cpp
#ifndef _CELL3_H_
#define _CELL3_H_

#include <iostream>
#include "AbsCell.hh"

using namespace std;

class Cell3: public AbsCell {
private:
  int state;
  int RuleSet[8];

public:
  Cell3() { state = 0; }

  Cell3(int aState): state(aState) {}

  Cell3(int aState, int* aRuleSet) {
    state = aState;
    for(int i=0; i<8; i++) {
      RuleSet[i] = aRuleSet[i];
    }
  }

  virtual ~Cell3() {}

  virtual int evolve(AbsCell* neighbors[]);

  virtual int getState() {
    return state;
  }

  virtual int setState(int aState) {
    state = aState;
  }

  virtual void print();
};

#endif
```


Cell.hh / Cell3.hh[v4]



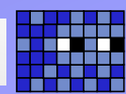
```cpp
#include "Cell.hh"

int Cell::evolve(AbsCell* neighbors[]) {
  int stateInfo = 4*(neighbors[0])->getState() + 2 * state + (neighbors[1])->getState();
  return RuleSet[stateInfo];
}

void Cell::print() {
  if(state) {
    cout << "o";
  } else {
    cout << " ";
  }
}
```

```cpp
#include "Cell3.hh"

int Cell3::evolve(AbsCell* neighbors[]) {
  int stateInfo = 4*(neighbors[0])->getState() + 2 * state + (neighbors[1])->getState();
  return RuleSet[stateInfo];
}

void Cell3::print() {
  if(state) {
    cout << ".";
  } else {
    cout << " ";
  }
}
```

```
[... same code as before, this is the only method that was
affected by the interface change due to delegation...]

template <class T> void Playground<T>::printArena() {
   for(int idx=0; idx<size; idx++) {
      (currentArena[idx])->print();
   }
   cout << endl;
}
```

Playground.hh [v4]



```cpp
#include<iostream>
#include "Playground.hh"
#include "Cell.hh"
#include "Cell3.hh"

#define PLAYGROUND_SIZE 80
#define GENERATIONS     100
#define RULE            30


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

  Playground<Cell>* myPlayground = new Playground<Cell>(PLAYGROUND_SIZE, RULE);

  for(int idx=0; idx<GENERATIONS; idx++) {
    myPlayground->nextGeneration();
    myPlayground->printArena();
  }

  cout << endl << "And now for something completely different... " << endl;

  Playground<Cell3>* myPlayground2 = new Playground<Cell3>(PLAYGROUND_SIZE, RULE);
  for(int idx=0; idx<GENERATIONS; idx++) {
    myPlayground2->nextGeneration();
    myPlayground2->printArena();
  }

  return 0;
}
```

In the end (this time really for real)

- If you don’t know C++, you didn’t learn it today
  - unless you knew it already
- It will take time to learn it
  - a lot of time
    - deal with it ;)
- There’s a lot more out there, and you will learn it over time.
- Like it or not, it’s a tool
- Object oriented programming is NOT C++
  - and vice versa!
  - sometimes C++ is defined a multi-paradigm language...
- The more you know the less you’ll be surprised...
  - and you don’t want to be surprised. Right? ;)