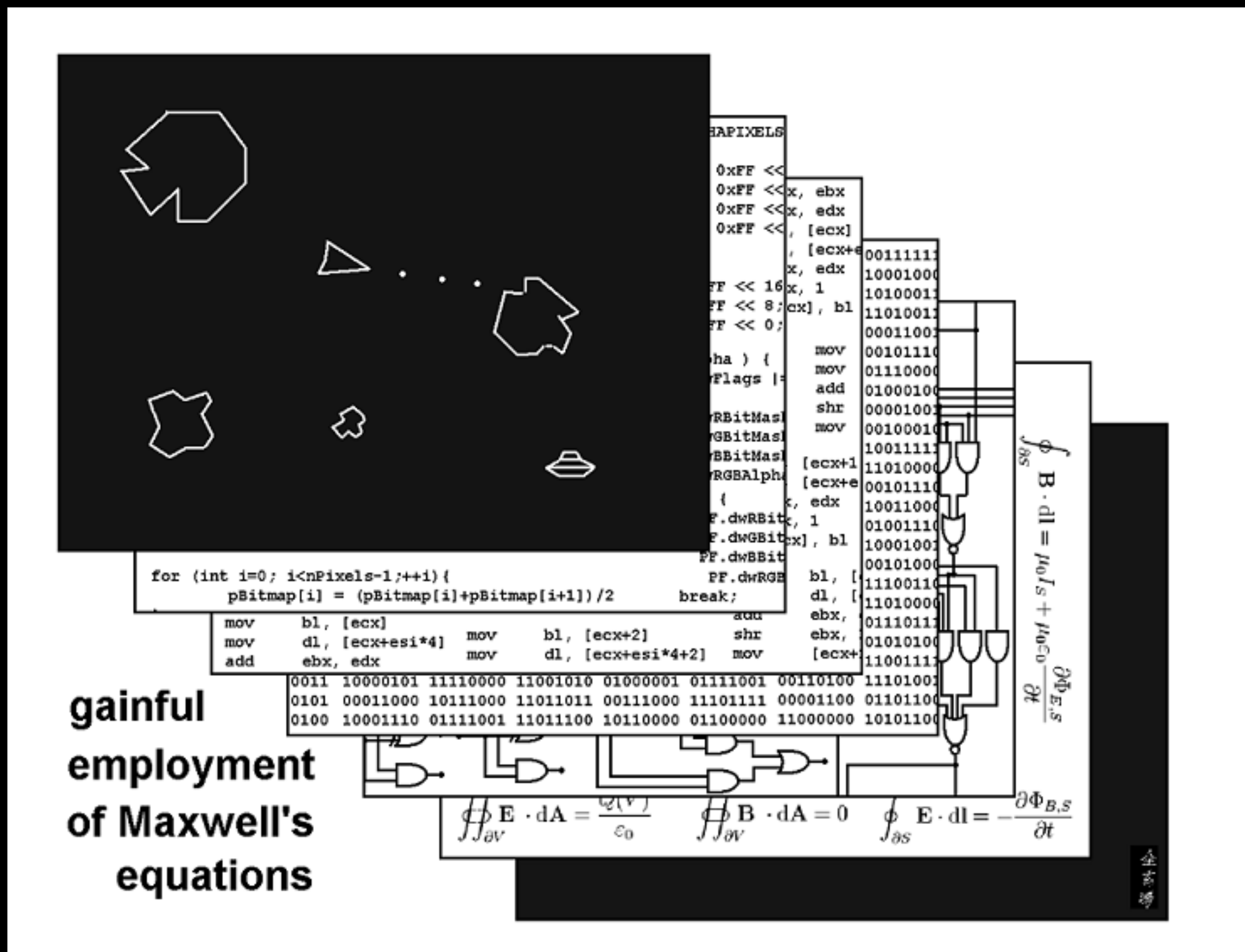
An Introduction to C++

David Dobrigkeit Chinellato
daviddc@ifi.unicamp.br

(Many thanks to Francesco Safai Tehrani!)
(these slides are based on his)

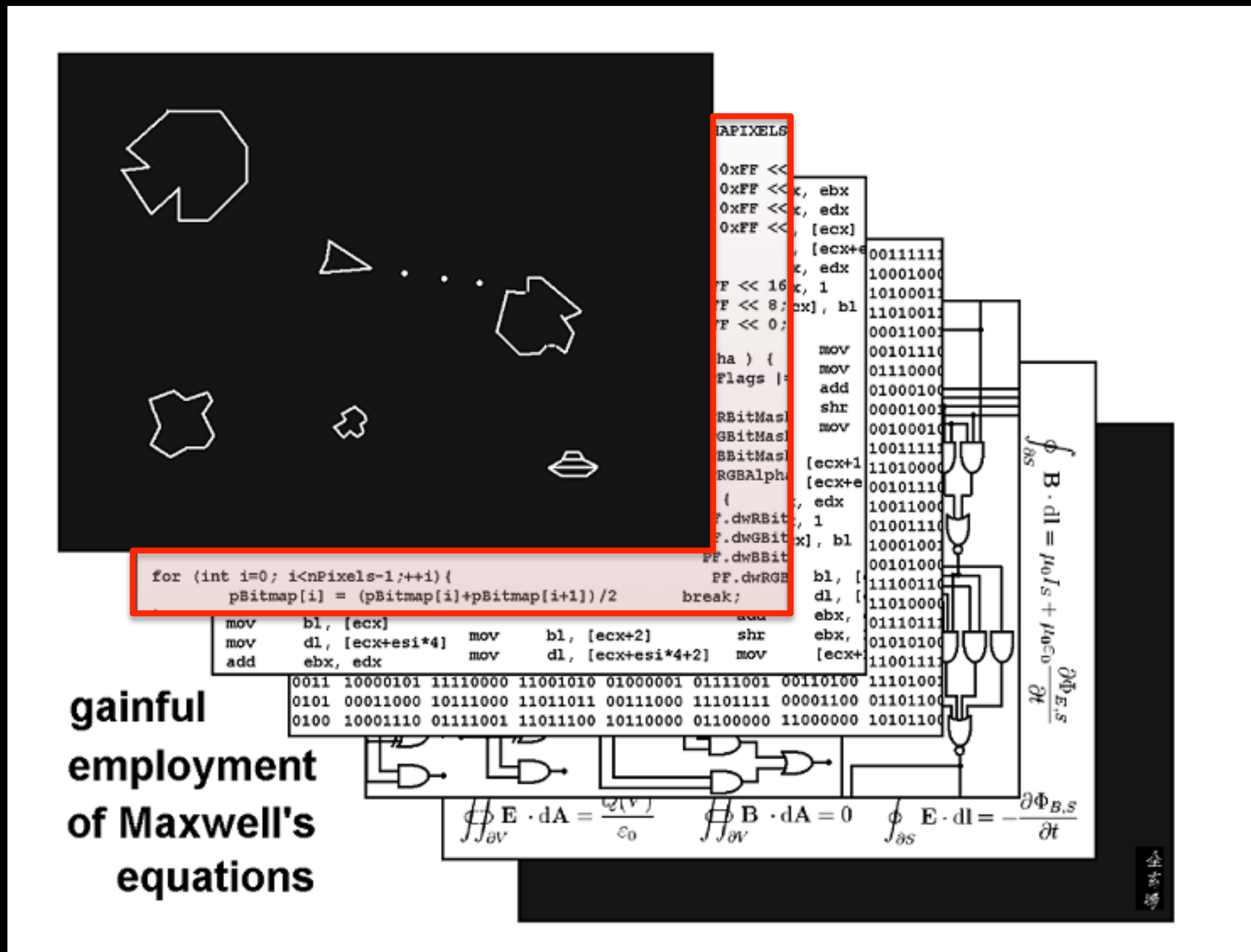
UNICAMP

Where we are…



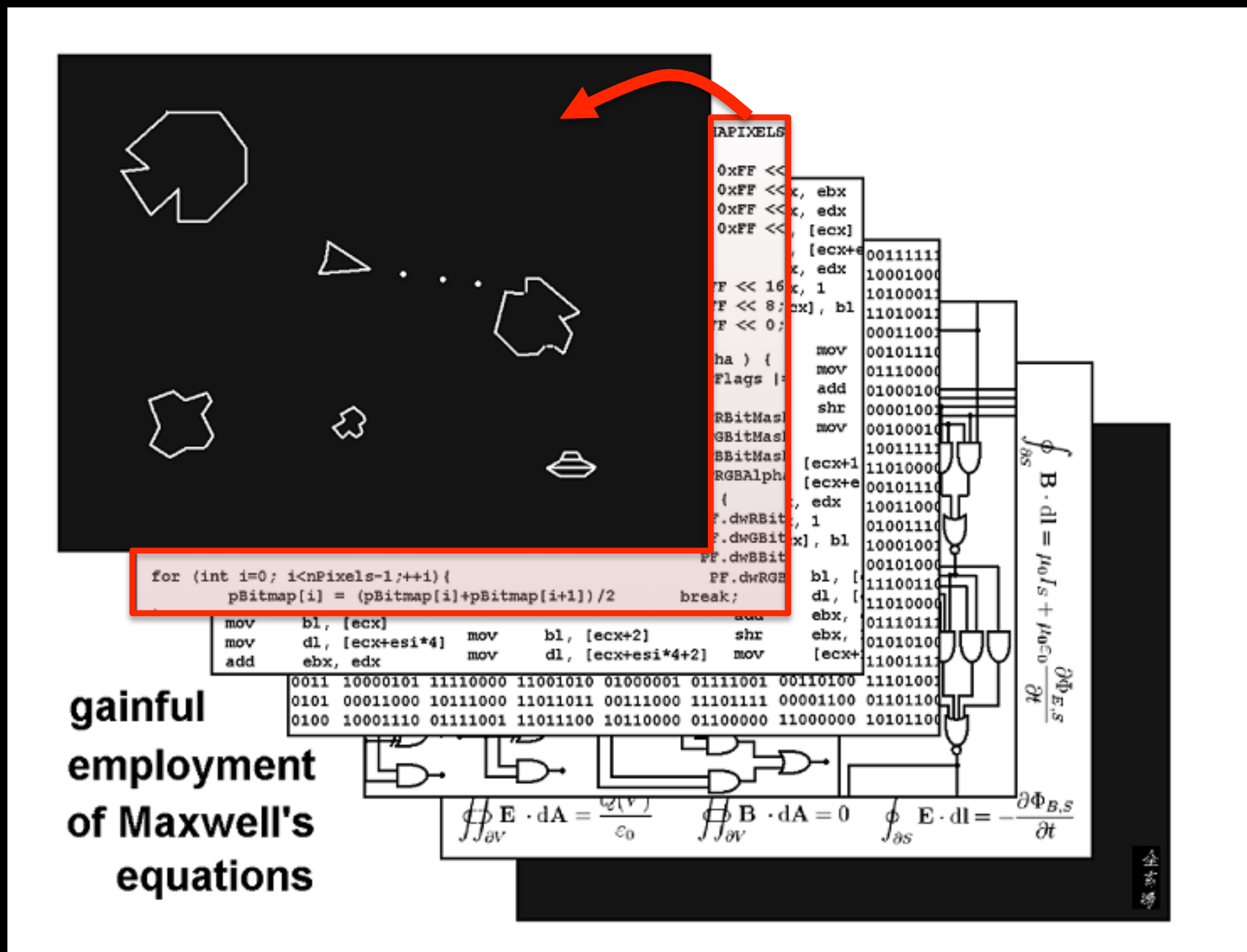
http://abstrusegoose.com/307

Where we are…



http://abstrusegoose.com/307

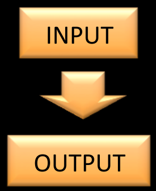
Where we are…



http://abstrusegoose.com/307

Object Oriented Programming

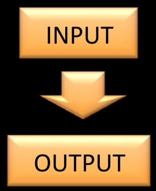
a.k.a. O.O.P.



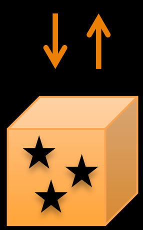
- **Functions:** code blocks that accept arguments and process those, returning some sort of response
  - Not (necessarily) a mathematical function!

Object Oriented Programming

a.k.a. O.O.P.



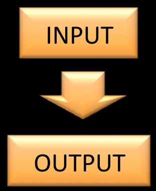
- **Functions:** code blocks that accept arguments and process those, returning some sort of response
  - Not (necessarily) a mathematical function!



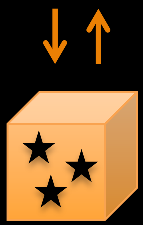
- **Objects:** 'Entities' that interact with other entities via well defined **interfaces**
  - The **interface** is the key: it fully describes what the object can do!
  - But the object is much more than the function: it may store data, etc...

Object Oriented Programming

a.k.a. O.O.P.



- **Functions:** code blocks that accept arguments and process those, returning some sort of response
  - Not (necessarily) a mathematical function!



- **Objects:** 'Entities' that interact with other entities via well defined **interfaces**
  - The **interface** is the key: it fully describes what the object can do!
  - But the object is much more than the function: it may store data, etc…

There are two main types of OOP:

- Static: objects have a fixed, well defined nature
- Dynamic: an object's nature is determined by its behavior

Object Oriented Programming (2)

a.k.a. O.O.P.

- OOP is essentially a technology
  - Different languages implement different OOPs
- The basic concepts can be traced back to the ‘50s
  - MIT AED-0, Simscript, Sketchpad, …
- The first explicit use of objects was in **Simula,** a discrete event simulation language, in the ’60s
  - Two versions: Simula I and Simula 67 (the latter ≈implemented ‘objects’)
  - Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in Oslo
- The term OOP was actually introduced in the language **Smalltalk**, developed at Xerox PARC
  - Versions Smalltalk 72 and 80 -> (more emphasis on dynamic than static)
  - Also invented: mouse (+GUI), ‘desktop’ metaphor, WYSIWYG, bitmaps, ….

http://en.wikipedia.org/wiki/Object-oriented_programming#History

From C to C++

- C is not C++ !
  - Well, so far so good… that should be known by now!
- C++ is a version of ‘C with objects’
  - Inspired by Simula 67
- Other ‘C with objects’ implementation exist, e.g. Objective-C
  - This one inspired by Smalltalk 80
  - If you are using an iSomething then you’re using software written in Objective C
- Both Smalltalk 80 and Simula 67 are still sometimes used (rare!)
- And where can we find what?
  - 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)

C++ by examples !

"……, Jim, I'm a [physicist], not a [programmer]!" - Dr. McCoy

- …or maybe not?

C++ by examples !

"……, Jim, I'm a [physicist], not a [programmer]!" - Dr. McCoy

- …or maybe not?

- Let's not try to go through all of C++
  - …or even 'the better part of it!'
- Let's just go through a few examples…
  - Hopefully, you'll grasp (at least some!) useful techniques in these as we go

Talking about OOP

- We need to know basic nomenclature (buzzwords!) to identify and get along with OOP
- There are tons of words there:
  - Class, object, instance, message, member variable, member function, interface,
  - overloading, constructor, destructor, delegation, inheritance, specialization,
  - generalization, abstraction, ownership, template,
  - implementation, private, public, protected, friend

Talking about OOP

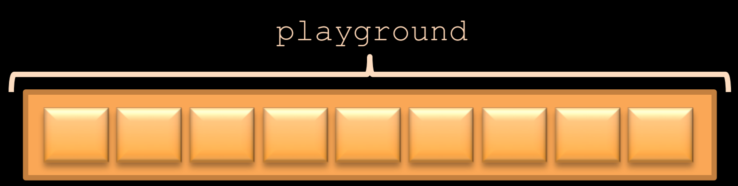
- We need to know basic nomenclature (buzzwords!) to identify and get along with OOP
- There are tons of words there:
  - Class, object, instance, message, member variable, member function, interface,
  - overloading, constructor, destructor, delegation, inheritance, specialization,
  - generalization, abstraction, ownership, template,
  - implementation, private, public, protected, friend
- Most of these are well defined…
  - …well, in a specific language. Subtle (or not so much) differences may crop up
  - But let's keep it simple! We'll stick to C++…
- Let's introduce the relevant words as we go!

A case study

1-Dimensional Cellular Automata

Let's focus on a *known* exercise!

- Let's think of a **playground of N spaces.**

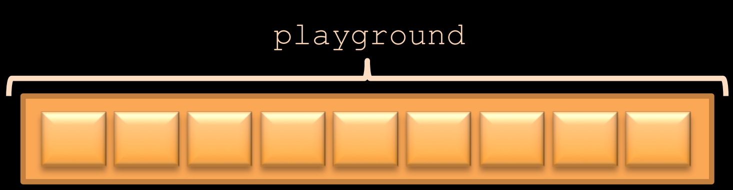


A case study

1-Dimensional Cellular Automata

Let's focus on a *known* exercise!

- Let's think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent

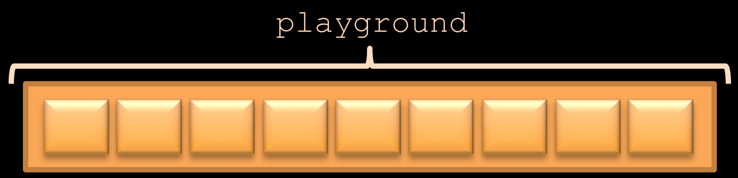


A case study

1-Dimensional Cellular Automata

Let’s focus on a *known* exercise!

- Let’s think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent
- **Each space** in the playground **is** called **a cell**

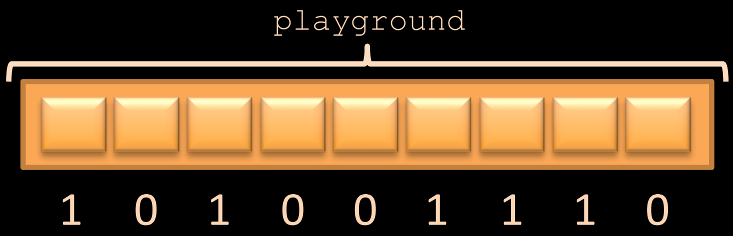


A case study

1-Dimensional Cellular Automata

Let's focus on a *known* exercise!

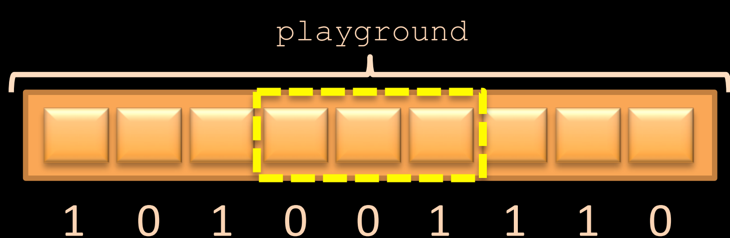
- Let's think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent
- **Each space** in the playground **is** called **a cell**
- Each cell is either dead (**state '0'**) or alive (**state '1'**)



1-Dimensional Cellular Automata

Let’s focus on a *known* exercise!

- Let’s think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent
- **Each space** in the playground **is** called **a cell**
- Each cell is either dead (**state ‘0’**) or alive (**state ‘1’**)
- **Time flows in discrete steps;** at each given step the cell state can be updated, either swapping state or staying the same, depending on its neighbor cells (‘neighborhood’)

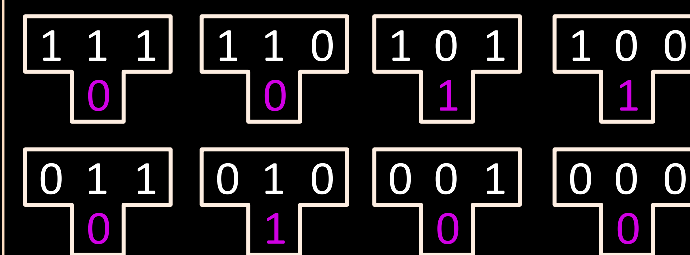


A case study

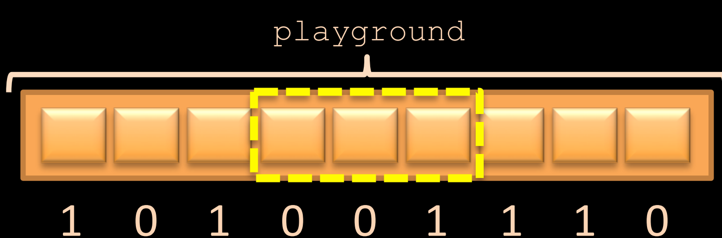
1-Dimensional Cellular Automata

Let's focus on a *known* exercise!

- Let's think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent
- **Each space** in the playground **is** called **a cell**
- Each cell is either dead (**state '0'**) or alive (**state '1'**)
- **Time flows in discrete steps;** at each given step the cell state can be updated, either swapping state or staying the same, depending on its neighbor cells ('neighborhood')
- There are **only eight possible configurations** for a neighborhood, and each with a possible outcome of 0 or 1 in the next state. Thus, there are only 256 possible evolution rules! →



As an example:
This is rule 52 (= $2^2 + 2^4 + 2^5$)



The implementation

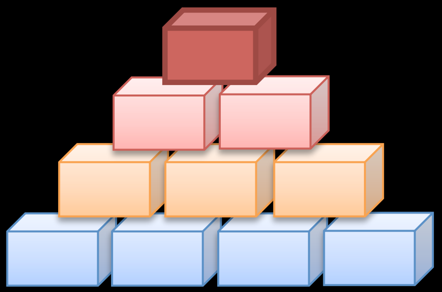
- This problem is hopefully familiar: we've covered this as a C exercise that we discussed previously…

The implementation

- This problem is hopefully familiar: we've covered this as a C exercise that we discussed previously…
- We will cover 4 versions of the 1D CA with increasing OOP !
  - This is still a fairly simple exercise…
  - … and this is complete overkill, but it's meant to exemplify OOP ideas!

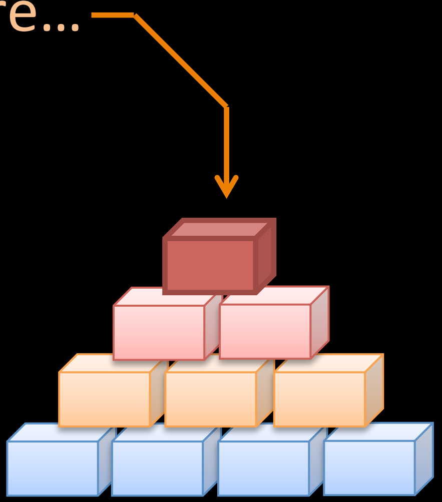
The implementation

- This problem is hopefully familiar: we've covered this as a C exercise that we discussed previously…
- We will cover 4 versions of the 1D CA with increasing OOP !
  - This is still a fairly simple exercise…
  - … and this is complete overkill, but it's meant to exemplify OOP ideas!
- Let's lay down some simple common infrastructure…
  - And sophisticate, step by step!



The implementation

- This problem is hopefully familiar: we've covered this as a C exercise that we discussed previously…
- We will cover 4 versions of the 1D CA with increasing OOP !
  - This is still a fairly simple exercise…
  - … and this is complete overkill, but it's meant to exemplify OOP ideas!
- Let's lay down some simple common infrastructure…
  - And sophisticate, step by step!
- This is not all!
  - This code does not use all features possible to 'simplify'
  - Instead, it is meant to be friendly!
    - We use loops instead of iterators, etc.…



Code Structure

Decisions, Decisions

- We need to break down the problem into entities!
  - **Playground Class:** abstracts the notion of the arena where cells 'live'
  - **Cell Class:** abstracts the cell

Code Structure

Decisions, Decisions

- We need to break down the problem into entities!
  - **Playground Class:** abstracts the notion of the arena where cells 'live'
  - **Cell Class:** abstracts the cell

- In this simple exercise, this is a straightforward decision
  - …but still, the 'breaking down' is an important design choice!
  - Different people will design their programs differently!
  - Experience, knowledge of problem, technical proficiency…
  - …but some of it is actually also taste and style !

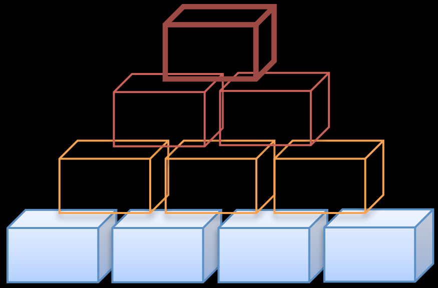
Code Structure

Decisions, Decisions

- We need to break down the problem into entities!
  - **Playground Class:** abstracts the notion of the arena where cells 'live'
  - **Cell Class:** abstracts the cell
- In this simple exercise, this is a straightforward decision
  - …but still, the 'breaking down' is an important design choice!
  - Different people will design their programs differently!
  - Experience, knowledge of problem, technical proficiency…
  - …but some of it is actually also taste and style !
- There is much more than one way!
  - …and none of them is 'right' !
    (well, as long as it works, of course!)

The One-dimensional Cellular Automaton

`Version 1.0`



v1. Basic Class Definition

1D CA v1: Object Definition: Cell

```
#ifndef _CELL_H_
#define _CELL_H_

using namespace std;

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

public:    Constructors
    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** (i.e. creating) an object of a certain type

1D CA v1: Object Definition: Cell

```
#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
```

Constructors

Overloading

Constructor: the method that is invoked when **instantiating** (i.e. creating) an object of a certain type

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

```
#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
```

Constructors

Overloading

Constructor: the method that is invoked when **instantiating** (i.e. creating) an object of a certain type

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

Destructor: the method that gets called when releasing / deleting an object.

Virtual: a tricky attribute which we'll explain a bit about later…

N.B. destructors have to be virtual, or else great care has to be taken!

1D CA v1: Object Definition: Cell

```
#include "Cell.hh"

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

```
#include "Cell.hh"

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

Remember: 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, …)

```cpp
#include "Cell.hh"

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

Remember: 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, …)

Alright, then what is a class?

```
#include "Cell.hh"

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

Remember: 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, …)

Alright, then what is a class?

- A class is a design entity representing a part of the solution to your problem.

```
#include "Cell.hh"

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

Remember: 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, …)

Alright, then what is a class?

- A class is a design entity representing a part of the solution to your problem.
- Different people will create different solutions (and classes!)

```
#include "Cell.hh"

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

Remember: 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, …)

Alright, then what is a class?

- A class is a design entity representing a part of the solution to your problem.
- Different people will create different solutions (and classes!)
- To be more precise, the class is a 'blueprint' of the actual entity. It's the project for a car, but is not a car yet, and has no state, behavior, etc: it cannot store data or answer to messages.

```cpp
#include "Cell.hh"

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

Remember: 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, …)

Alright, then what is a class?

- A class is a design entity representing a part of the solution to your problem.
- Different people will create different solutions (and classes!)
- To be more precise, the class is a 'blueprint' of the actual entity. It's the project for a car, but is not a car yet, and has no state, behavior, etc: it cannot store data or answer to messages.
- When you 'create a car' from this blueprint, i.e. generate an object from a class, this is called **instantiation.** This object is a real entity: it has state and behavior!
- An **interface** fully defines what an object can do (or what messages it can answer)

```cpp
#include "Cell.hh"

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

Remember: 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, …)

Alright, then what is a class?

- A class is a design entity representing a part of the solution to your problem.
- Different people will create different solutions (and classes!)
- To be more precise, the class is a 'blueprint' of the actual entity. It's the project for a car, but is not a car yet, and has no state, behavior, etc: it cannot store data or answer to messages.
- When you 'create a car' from this blueprint, i.e. generate an object from a class, this is called **instantiation.** This object is a real entity: it has state and behavior!
- An **interface** fully defines what an object can do (or what messages it can answer)
- The **object** is the core entity of C++ programming. A C++ program can be seen as a 'network of objects' which interact with each other.

1D CA v1: Object Definition: Cell

```cpp
#include "Cell.hh"

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

Remember: 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, …)

Alright, then what is a class?

- A class is a design entity representing a part of the solution to your problem.
- Different people will create different solutions (and classes!)
- To be more precise, the class is a 'blueprint' of the actual entity. It's the project for a car, but is not a car yet, and has no state, behavior, etc: it cannot store data or answer to messages.
- When you 'create a car' from this blueprint, i.e. generate an object from a class, this is called **instantiation.** This object is a real entity: it has state and behavior!
- An **interface** fully defines what an object can do (or what messages it can answer)
- The **object** is the core entity of C++ programming. A C++ program can be seen as a 'network of objects' which interact with each other.
- Or, alternatively: each object can ask another to do something and return a result (which is known as **delegation**)

1D CA v1: Object Definition: Playground

```
#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
```

When defining the playground class, we can now make ample use of the 'cell' class!

1D CA v1: Object Definition: Playground

```
#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
```

When defining the playground class, we can now make ample use of the 'cell' class!

The playground contains the (discrete!) time evolution method (which will have to call Cell::Evolve!)

```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;
}
```

Among other simple setup functions… There is the most important ingredient: the evolution!

```
#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;
}
```

Among other simple setup functions… There is the most important ingredient: the evolution!

Here, we call Cell:Evolve (where the evolution rule is!) to determine 'nextarena', the array with the next states!

```
#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;
}
```

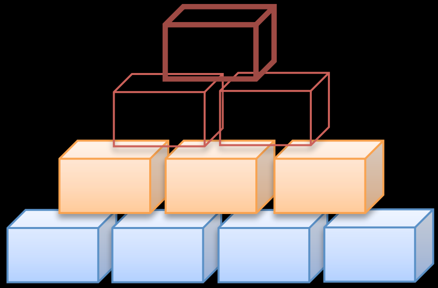
Code simplicity:

- When using classes, your code will be *modular*, and the main program will be exceedingly simple – such as the one above!

…and this was the 1D CA v1. Let's improve!

The One-dimensional Cellular Automaton

Version 2.0



v2. Abstract Classes and Inheritance

v1. Basic Class Definition

```
#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 methods!
This essentially tells the compiler: 'wait for it…'

- This is an **abstract class:** a class that cannot be instantiated. It does define an *interface,* but does not define the *behavior* associated to said interface.

```
#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 methods!
This essentially tells the compiler: 'wait for it…'

- This is an **abstract class:** a class that cannot be instantiated. It does define an *interface,* but does not define the *behavior* associated to said interface.
- Real classes can be implemented from an abstract class via **inheritance** and these classes will inherit the interface and will have to respond to these calls.

1D CA v2: Object Definition: Abstract Cell

```
#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 methods!

This essentially tells the compiler: 'wait for it…'

- This is an **abstract class:** a class that cannot be instantiated. It does define an *interface,* but does not define the *behavior* associated to said interface.
- Real classes can be implemented from an abstract class via **inheritance** and these classes will inherit the interface and will have to respond to these calls.
- If B inherits from A, it will have to know how to answer to any calls that A is able to respond to (but not vice versa!)

```
#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 methods!
This essentially tells
the compiler: 'wait for it…'

- This is an **abstract class:** a class that cannot be instantiated. It does define an *interface,* but does not define the *behavior* associated to said interface.
- Real classes can be implemented from an abstract class via **inheritance** and these classes will inherit the interface and will have to respond to these calls.
- If B inherits from A, it will have to know how to answer to any calls that A is able to respond to (but not vice versa!)
- A classic example: the 'Rectangle' class can inherit from a 'Shape' class, which has an abstract 'Draw' method. The 'Rectangle' can have a **specialized** 'Draw' to draw a rectangle.

```
#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 methods!
This essentially tells the compiler: 'wait for it…'

- This is an **abstract class:** a class that cannot be instantiated. It does define an *interface,* but does not define the *behavior* associated to said interface.
- Real classes can be implemented from an abstract class via **inheritance** and these classes will inherit the interface and will have to respond to these calls.
- If B inherits from A, it will have to know how to answer to any calls that A is able to respond to (but not vice versa!)
- A classic example: the 'Rectangle' class can inherit from a 'Shape' class, which has an abstract 'Draw' method. The 'Rectangle' can have a **specialized** 'Draw' to draw a rectangle.
- In C++, this **forces the developer** to correctly implement the *full interface defined in the object from which it inherits,* or else ….

```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
```

Here it is: Inheritance!
The class Cell inherits the interface (and behavior, if defined) from AbsCell.

One can also say:
Cell **conforms** to, or **implements**, the AbsCell interface

1D CA v2: Object Definition: Cell

```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
```

Here it is: Inheritance!
The class Cell inherits the interface (and behavior, if defined) from AbsCell.

One can also say:
Cell **conforms** to, or **implements**, the AbsCell interface

And here, we **specialize** the evolve method defined in AbsCell …

```
#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();
};
#endif
```

Now we can replace most of the calls to AbsCell in the playground as calls to AbsCell, to take care of any general cases…

That's alright: all cells here should have the **needed** interface.

(But more on that in v3!)

Wait, now which ::Evolve is called?

There is one in AbsCell and one in Cell!

```
#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;
}
```

```
#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;
}
```

Wait, now which ::Evolve is called?

There is one in AbsCell and one in Cell!

C++ will take the definition from the non-abstract class: the one which inherited the interface (e.g. 'Cell'). This is what 'virtual' does!

```
#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;
}
```

Wait, now which ::Evolve is called?

There is one in AbsCell and one in Cell!

C++ will take the definition from the non-abstract class: the one which inherited the interface (e.g. 'Cell'). This is what 'virtual' does!

This is very powerful! You can make function calls using the known interface without the function itself even being defined in a particular context… The binding will take place later! (but it still has to take place!)

```
#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;
}
```

Wait, now which ::Evolve is called?

There is one in AbsCell and one in Cell!

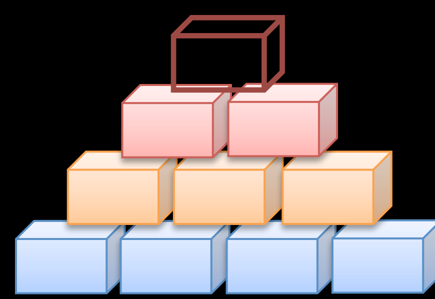
C++ will take the definition from the non-abstract class: the one which inherited the interface (e.g. 'Cell'). This is what 'virtual' does!

This is very powerful! You can make function calls using the known interface without the function itself even being defined in a particular context… The binding will take place later! (but it still has to take place!)

…and this was the 1D CA v2. Let's improve!

The One-dimensional Cellular Automaton

Version 3.0



v3. Template Classes

v2. Abstract Classes and Inheritance

v1. Basic Class Definition

What if we have other Cell types?...

- So far, so good! But… what if somebody asks us to define a different type of Cell (which conforms to AbsCell)?
- Would we have to rewrite the whole playground?

What if we have other Cell types?...

- So far, so good! But… what if somebody asks us to define a different type of Cell (which conforms to AbsCell)?
- Would we have to rewrite the whole playground?
- Naturally not!
  - There are nearly infinite possibilities, so….
  - What if we could tell playground to accept *multiple class types*, somehow?

UNICAMP

What if we have other Cell types?...

- So far, so good! But... what if somebody asks us to define a different type of Cell (which conforms to AbsCell)?
- Would we have to rewrite the whole playground?
- Naturally not!
  - There are nearly infinite possibilities, so....
  - What if we could tell playground to accept *multiple class types*, somehow?
- For that, we can use **templates!**
  - These are structures that have parametric arguments which can be of various classes (or, generally, **types**)
  - They are extremely powerful! They form the basis of 'generic programming', see:
  - http://en.wikipedia.org/wiki/Generic_programming
- Here, we will merely cover a super simple example!

UNICAMP

1D CA v3: Object Definition: Playground

```
#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();
};
```

This is a 'playground' template!

Here, we tell the compiler that this is a template definition and the class T is an (as of yet) unidentified argument to playground

1D CA v3: Object Definition: Playground

```
#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();
};
```

This is a 'playground' template!

Here, we tell the compiler that this is a template definition and the class T is an (as of yet) unidentified argument to playground

The compiler knows **nothing** about T until I start using the playground. Then, when I instantiate a playground, I will have to define what the class T is!

```
#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;
}
```

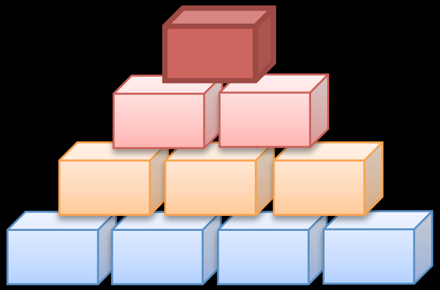
And this is the instantiation!

Essentially, we passed the 'Cell' class **as an argument** to the template of playground. When we did this, the compiler used the template to provide a "customized" playground class in which 'T' was 'Cell', and carried on!

...and this was the 1D CA v3. Let's improve!

The One-dimensional Cellular Automaton

Version 4.0



v4. Delegation and more
v3. Template Classes
v2. Abstract Classes and Inheritance
v1. Basic Class Definition

```
#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
```

Shouldn't the cell always know how to display itself?

- Responsibility: Who should do what?
- Shouldn't the AbsCell decide how to display itself?
- Somebody should, and everybody should be able to delegate the task of drawing to the cell (any cell!) of AbsCell type!

1D CA v4: Object Definition: Cell / Cell3

```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
```

- Two Cell Types: Cell and Cell3, both conform to AbsCell but otherwise identical!

1D CA v4: Object Definition: Cell / Cell3

```
#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
```

```
#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
```

Let's play with this….

- Two Cell Types: Cell and Cell3, both conform to AbsCell but otherwise identical!

1D CA v4: Object Definition: Cell / Cell3

```
#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 << " ";
    }
}
```

"o"

```
#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 << " ";
    }
}
```

"."

- They get drawn slightly differently, but the interface to the call is already defined in AbsCell! (i.e. even before the compiler knows of any specific ::print())

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

It looks like a small enough change: let’s make sure to invoke AbsCell::print() when the playground gets asked to printArena()

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

It looks like a small enough change: let's make sure to invoke AbsCell::print() when the playground gets asked to printArena()

…but actually, this is a good example of **delegation**: the Playground class delegates the print task to the Cell, instead of drawing on screen by itself, as it was doing before. Now the Cell will print out, not Playground!

```
#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;
}
```

This is short and concise, but we've used a lot of techniques here!

The first playground is filled with "○" while the second is "."

...and this was the 1D CA v4

And that was it for now…

- If you didn't know C++ (or some of it)
  - …you might not have learned it today… Sorry.
  - But that was not quite the idea! Hopefully, you got something from this…

And that was it for now…

- If you didn't know C++ (or some of it)
  - …you might not have learned it today… Sorry.
  - But that was not quite the idea! Hopefully, you got something from this…
- It takes time…
  - Yes, some of us are physicists and not programmers, but that's no excuse!

And that was it for now…

- If you didn't know C++ (or some of it)
  - …you might not have learned it today… Sorry.
  - But that was not quite the idea! Hopefully, you got something from this…
- It takes time…
  - Yes, some of us are physicists and not programmers, but that's no excuse!
- In the end, like it or not, it's a tool
  - And one that is as powerful as we can make it

And that was it for now…

- If you didn't know C++ (or some of it)
  - …you might not have learned it today… Sorry.
  - But that was not quite the idea! Hopefully, you got something from this…
- It takes time…
  - Yes, some of us are physicists and not programmers, but that's no excuse!
- In the end, like it or not, it's a tool
  - And one that is as powerful as we can make it
- Object oriented programming: a different way of thinking
  - But it's not exclusive to C++!

And that was it for now…

- If you didn't know C++ (or some of it)
  - …you might not have learned it today… Sorry.
  - But that was not quite the idea! Hopefully, you got something from this…
- It takes time…
  - Yes, some of us are physicists and not programmers, but that's no excuse!
- In the end, like it or not, it's a tool
  - And one that is as powerful as we can make it
- Object oriented programming: a different way of thinking
  - But it's not exclusive to C++!
- The more you know about what's out there…
  - …the less you'll be surprised. And nobody likes to be surprised…

Thank you!