Øvingsforelesning 9

TDT 4102
Prosedyre- og objektorientert programmering

15. mars 2017

Antonín Klíma
antonink@idi.ntnu.no

Øving 9: Minesweeper

- rather short
- mainly combines knowledge

Initialization:

- Dynamic 2D array of objects
- Mines: random combination

Progression:

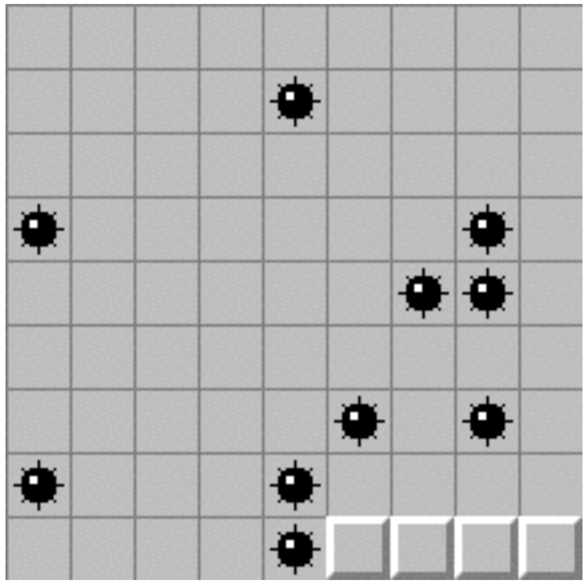
- Checking 8-boundaries
- Recursive opening

★board boundaries

Minesweeper

Initialization

- Dynamic 2D array of objects
- Mines: random combination

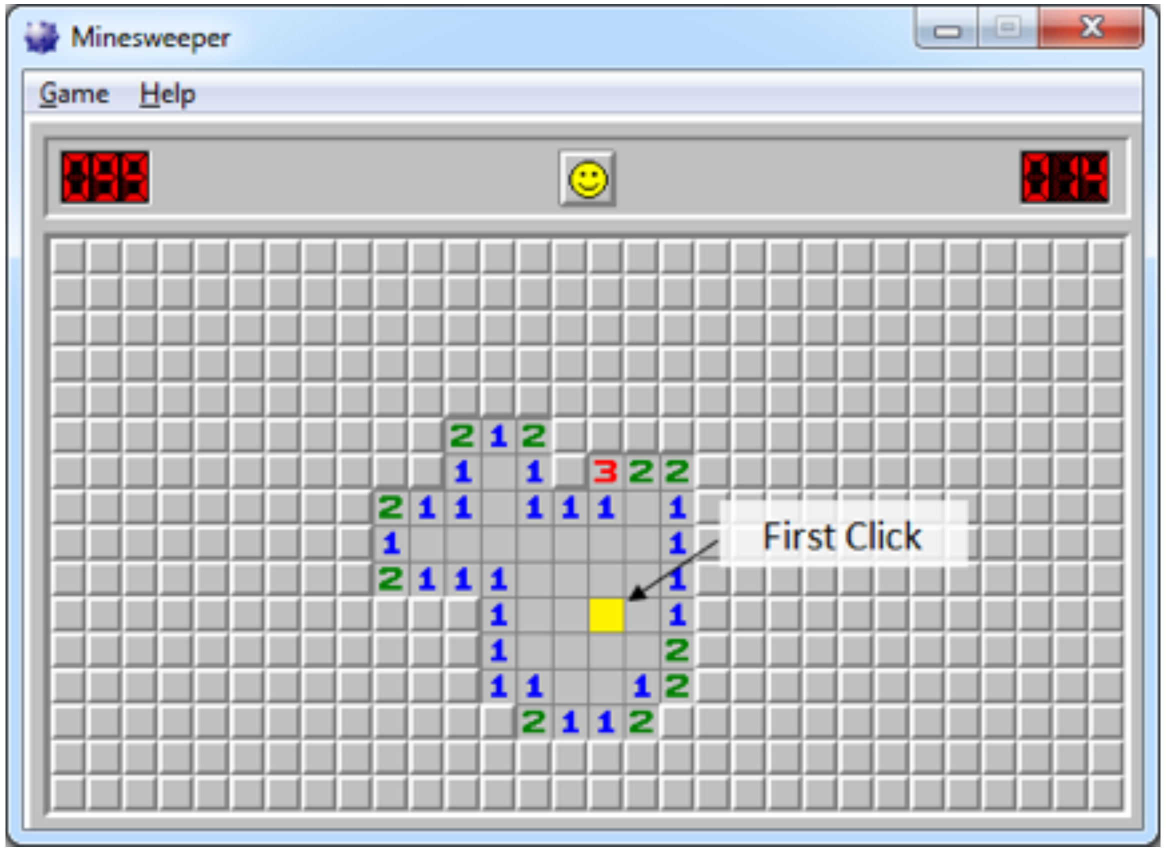


Minesweeper
Progression

- Recursive opening
- Checking 8-boundaries
  - grid boundaries

Minesweeper
Progression

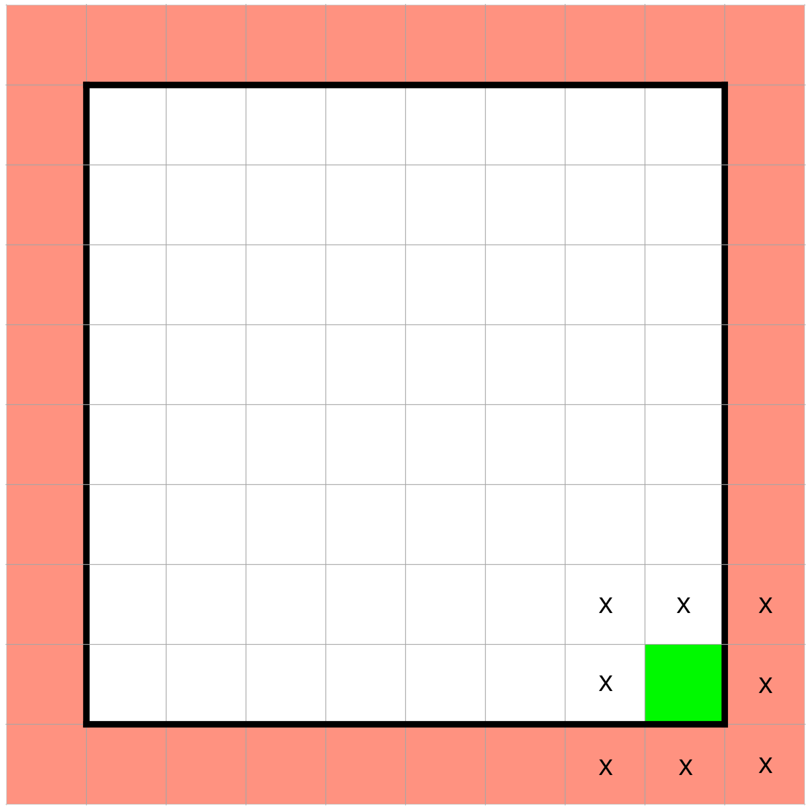
- Recursive opening

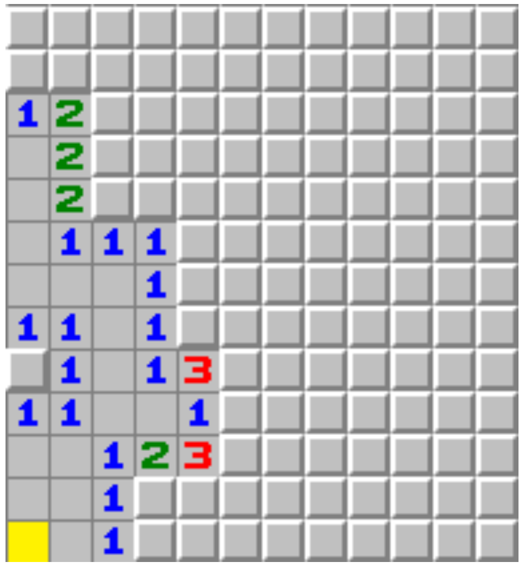


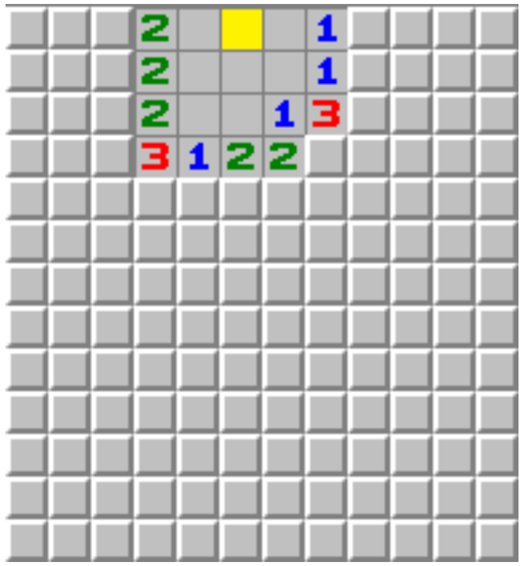
[datagenetics.com]

Minesweeper
Progression

- Checking 8-boundaries







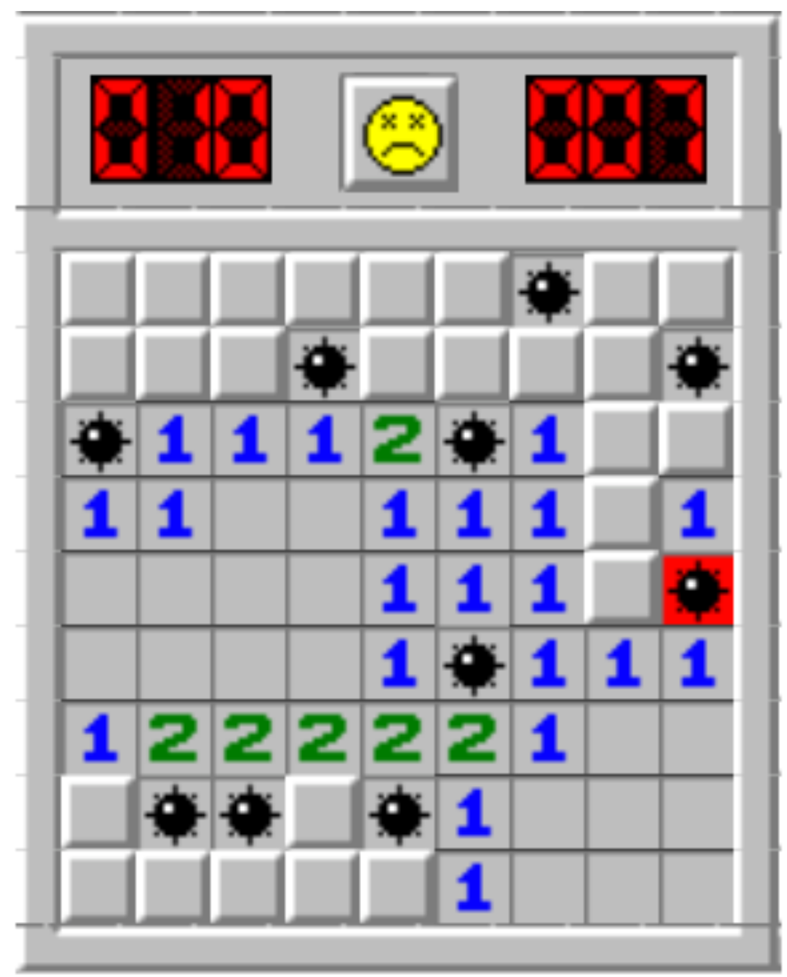
[datagenetics.com]

Minesweeper

win & loss

win

loss



But first..

A prelude on **const** type qualifier

- most important uses
- common caveats
- will come handy later

Const keyword

- Automatic error checking
- Potential compiler optimization
- Promises not to change variable

Rule of thumb? 

If you can, use const.

Const keyword

Most notable uses

- Passing by const reference
  - avoids unnecessary copying
  - internally passed as pointer

```
class T;
class Type{
    Type( T const& obj );
};
```

Caveat

Call-by-reference

Reference (T&) .. lvalue

```
void goo(int)   { cout << "goo\n"; }
void goo(char&) { cout << "hoo\n"; }

int main() {
    char chr = 'c';
    goo(chr);
    goo('c');
}
```

BUT

Const reference (const T&) .. can be rvalue

Const keyword

Call by reference caveat - advanced

rvalue references (&&)

```
void goo(int)    { cout << "goo\n"; }
void goo(char&) { cout << "hoo\n"; }
void goo(char&&) { cout << "hoho\n"; }

int main() {
    char chr = 'c';
    goo(chr);
    goo('c');
```

Example of use

```
class T;
class Type{
    Type( T const& obj );   // expensive
    Type( T && obj );       // cheap
};
```

Const keyword

Most notable uses

Constant methods

```
int get() const {return this->i;}
```

- promise not to change the object
- vital for working with constant objects

```
class Type{
    int i;
public:
    Type(int i): i(i) {}
    int get() {return i;}
};

int main() {

    const Type tmp(0);
    tmp.get();
```

Member function 'get' not viable: 'this' argument has type 'const Type', but function is not marked const

Const keyword

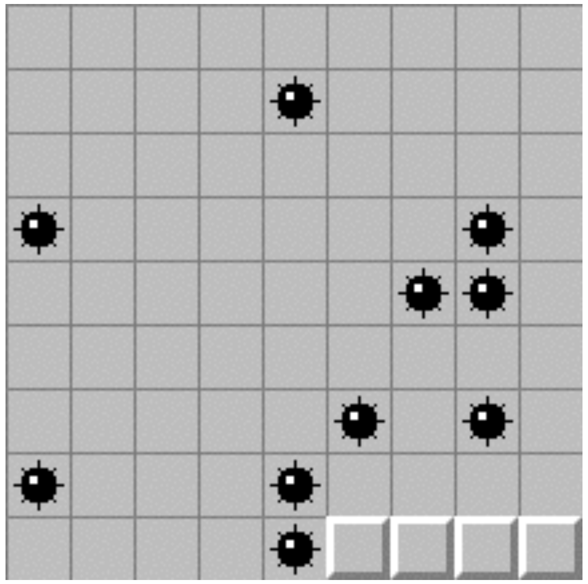
pointers and const

```
const int *p1 = nullptr;
//  p1 = new int;
// *p1 = 666;

int * const p2 = nullptr;
//  p2 = new int;
// *p2 = 666;
```

Which of the statements are allowed?

Minesweeper
Initialization

- **Dynamic 2D array of objects**
  - principles
  - pros & cons
  - a trick



- Mines: random combination

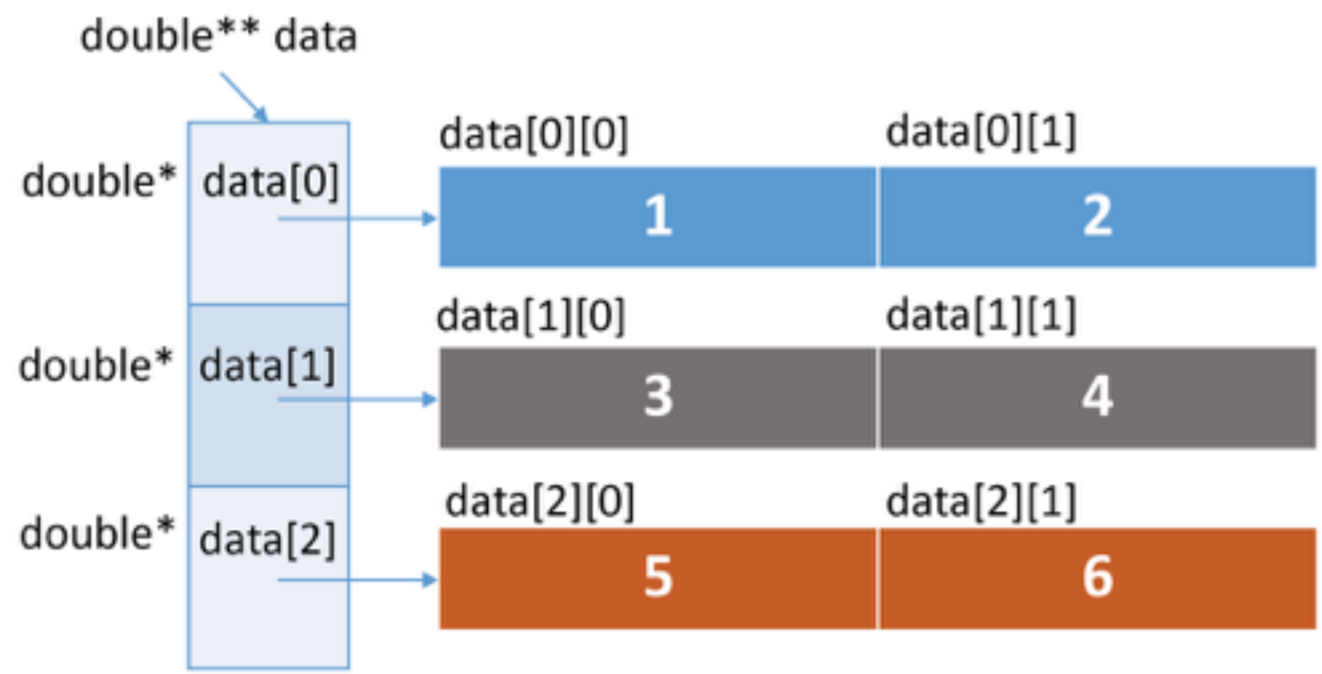
Repetition

Saving 2D arrays

Pointers to pointers

Pros

- Intuitive
- Indexing [i][j] ready



Cons

- more complicated (de)allocation
- less efficient (time & space)

1 3 5

2 4 6

Saving 2D arrays

Single array

$$\begin{matrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{matrix}$$

Pros

- Efficient (time & space)
  - Better caching
  - Less dereferencing
- Easy (de)allocation

Cons

- More code to write
- Δperformance: (usually) negligible

Saving 2D arrays

Static 2D array?

```
int array1[3][2] = {{0, 1}, {2, 3}, {4, 5}};
```

How is it stored in memory?

In memory looks like this:

```
0 1 2 3 4 5
```

exactly the same as:

```
int array2[6] = { 0, 1, 2, 3, 4, 5 };
```

[stackoverflow]

caveat

```
int q[2][3];
cout<<q[0][4]<<endl; // legal, same as q[1][1] !
```

Saving 2D arrays

Single array

$$\begin{matrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{matrix}$$

Can we get back intuitive indexing?

Saving 2D arrays

Yes! The following code does what we expect:

```
class Grid{
    unsigned height, width;
    int* values;
public:
    int* operator[] (int i)
    {
        return values + i*width;
    }

    Grid(unsigned height, unsigned width): width(width), height(height),
        values(nullptr)
    {
        values = new int[width*height];
        for (int i=0;i<height*width;i++) values[i]=i;
    }
};

int main() {

    Grid test(3,4);

    cout<<test[1][2];
    test[1][2] = 0;
    cout<<test[1][2];

    return 0;
}
```

Or does it?

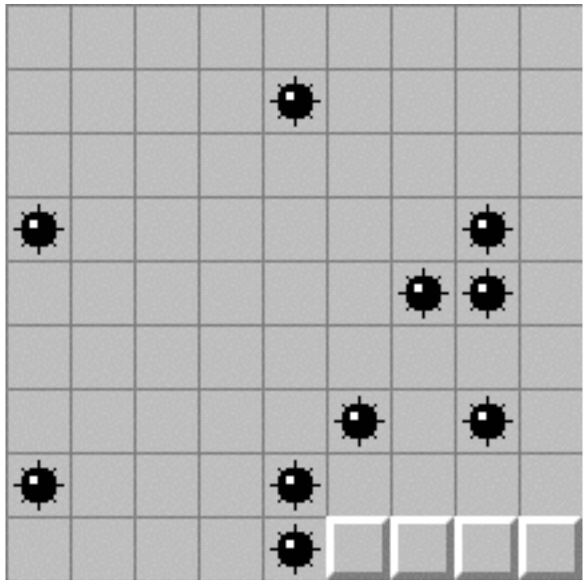
Saving 2D arrays

Now it's better. :)

```
class Grid{
    unsigned height, width;
    int* values;
public:
    const int* operator[] (int i)
    {
        return values + i*width;
    }

    Grid(unsigned height, unsigned width): width(width), height(height),
        values(nullptr)
    {
        values = new int[width*height];
        for (int i=0;i<height*width;i++) values[i]=i;
    }
};

int main() {

    Grid test(3,4);

    cout<<test[1][2];
    test[1][2] = 0;                    Read-only variable is not assignable
    cout<<test[1][2];

    return 0;
}
```

Minesweeper
Initialization

- Dynamic 2D array of objects



- Mines: random combination
  - Naive vs effective solution
  - <algorithm> std::random_shuffle

Random combinations

Naive

Idea

- Choose random index
- Repeat if already chosen

Algorithm 1 Naive random combinations

```
function RandomCombination(k, n)
    combination = {}
    for i in {1, 2, .. k} do
        element = randomFromTo(1, n)
        while element ∈ combination do
            element = randomFromTo(1, n)
        end while
        combination = combination ∪ {element}
    end forreturn combination
end function
```

Complexity?

$$n \log n - n \log(n-k) \approx n \log n$$
$$\text{for } k \in O(n)$$

→ not optimal

Random combinations

Naive - derivation

#attempts = #step$_0$+ #step$_1$ + …

E[#attempts] = E[#step$_0$]+ E[#step$_1$] + …

step$_i$: $\mathrm{P}(\mathrm{valid}_i) = \frac{n-i}{n}$

$\mathbf{E}(\mathrm{step}_i) = \frac{n}{n-i}$

Expected running time

$$\mathbf{E}(\#\mathrm{attempts}) = \sum_{i=0}^{k-1} \frac{n}{n-i} \sim \int_{i=0}^{k-1} \frac{n}{n-i}$$

Random combinations

Fisher-Yates shuffle

Idea

- Choose random index from those not already chosen

Pros

- Unbiased
- Linear - O(k)

Random combinations

Fisher-Yates shuffle

Pseudocode - shuffle full array in O(n)

```
-- To shuffle an array a of n elements (indices 0..n-1):
for i from n−1 downto 1 do
     j ← random integer such that 0 ≤ j ≤ i
     exchange a[j] and a[i]
```

[wiki]

<algorithm> std::random_shuffle

How do we get the combination?

Conclusion:

Don't reinvent the wheel.

Minesweeper
Progression

- **Recursive opening**
  - principles
  - pros & cons
  - Fibonacci (øv 7)
- Checking 8-neighbourhood
  - grid boundaries

Repetition

Recursion

Algorithm 2 Generate n-th fibonacci number

function FIBONACCI(n)
    if $n \leq 2$ **then**
        return 1   **base case**
    end if
    return FIBONACCI($n-1$)+FIBONACCI($n-2$)
end function   **recursive call**

Algorithm 3 Compute factorial recursively

function FACT(n)
    if $n == 0$ **then**
        return 1   **base case**
    end if
    return n*FACT(n-1)   **recursive call**
end function

- function calls itself
- **requires base case**

- new model of thinking
- alternative to iteration
  - replaces it in *functional programming :)*

Repetition

Recursion

Algorithm 2 Generate n-th fibonacci number

function FIBONACCI(n)
  if $n \leq 2$ **then**
    return 1 base case
  end if
  return FIBONACCI($n-1$)+FIBONACCI($n-2$)
end function recursive call

Algorithm 3 Compute factorial recursively

function FACT(n)
  if $n == 0$ **then**
    return 1 base case
  end if
  return n*FACT(n-1) recursive call
end function

Pros

- conceptually elegant
- often reasonably efficient
  - when not, often leads to dynamic programming algorithm
- *divide & conquer*
- easier to prove correctness

Recursion

Algorithm 2 Generate n-th fibonacci number

function FIBONACCI(n)
    if $n \leq 2$ **then**
        return 1 base case
    end if
    return FIBONACCI($n-1$)+FIBONACCI($n-2$)
recursive call
end function

Algorithm 3 Compute factorial recursively

function FACT(n)
    if $n == 0$ **then**
        return 1 base case
    end if
    return n*FACT(n-1)   recursive call
end function

Cons

- generally greater overhead (space, time)
- function calls accumulate on *stack*
  - rather limited 1-2 MB
  - scalability issues - stack overflow :)
- possible redundant computations
- trickier to determine complexity

Recursion

Algorithm 2 Generate n-th fibonacci number

```
function FIBONACCI(n)
    if n ≤ 2 then
        return 1                              base case
    end if
     return FIBONACCI(n − 1)+FIBONACCI(n − 2)
end function                                  recursive call
```

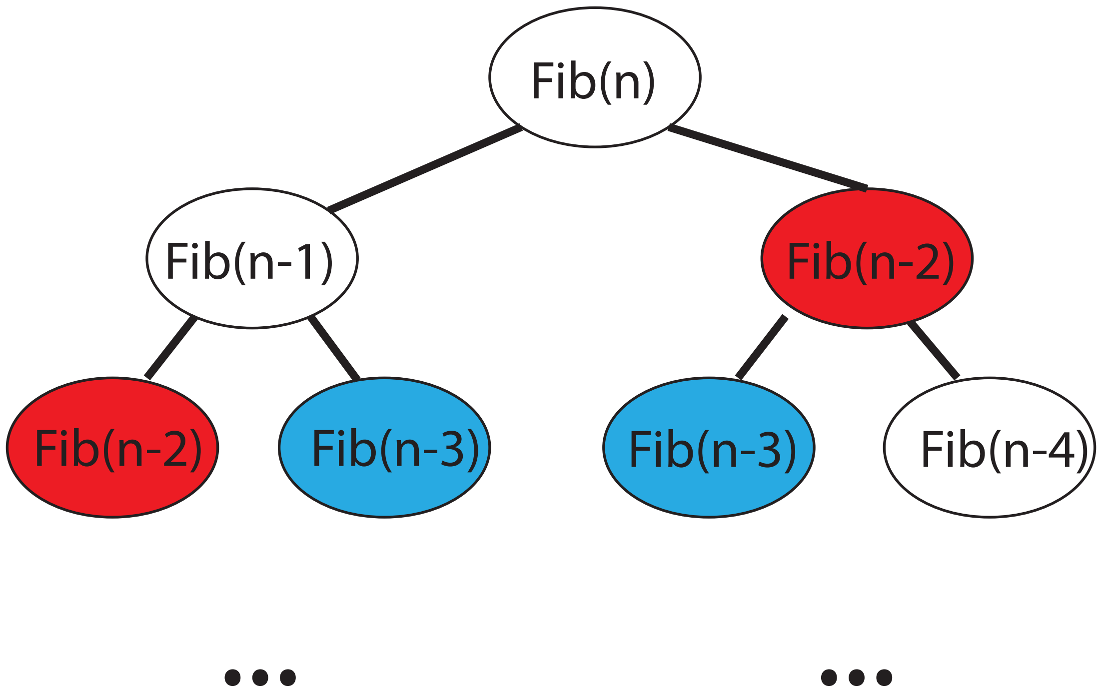
Algorithm 3 Compute factorial recursively

```
function FACT(n)
    if n == 0 then
        return 1          base case
    end if
     return n*FACT(n-1)   recursive call
end function
```

Control question

Which of the algorithms above is inefficient? Why? How inefficient?

Fibonacci

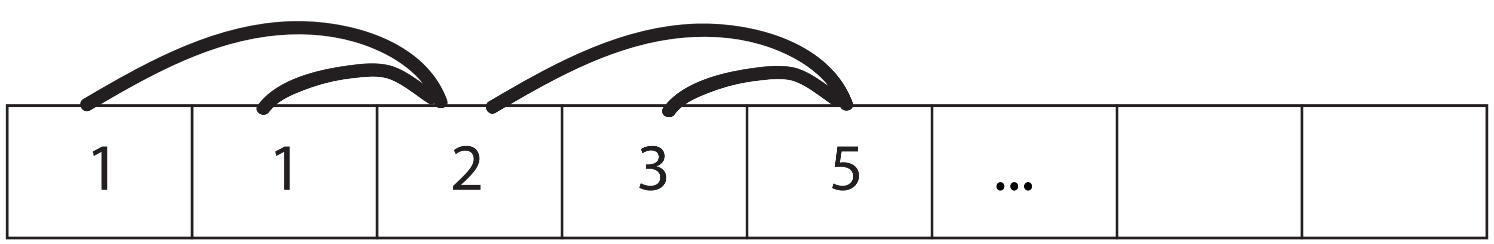


Recursion

$O(2^n)$
repeated subproblems
$O(1)$ per leaf

Iteration

dynamic programming



$O(n)$
$O(1)$ per value

Master theorem

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b > 1$$

In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:

- *n* is the size of the problem.
- *a* is the number of subproblems in the recursion.
- *n*/*b* is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- *f* (*n*) is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

- determines time complexity
- can be applied to most feasible recursive algorithms

a	increases width
b	decreases height
f(n)	determines work done at nodes

Repetition

Recursion

Summary

Algorithm 2 Generate n-th fibonacci number

```
function FIBONACCI(n)
    if n ≤ 2 then
        return 1          base case
    end if
     return FIBONACCI(n − 1)+FIBONACCI(n − 2)
                                          recursive call
end function
```

Algorithm 3 Compute factorial recursively

```
function FACT(n)
    if n == 0 then
        return 1          base case
    end if
     return n*FACT(n-1)   recursive call
end function
```

- function calls itself
- requires base case
- elegant
  - but beware

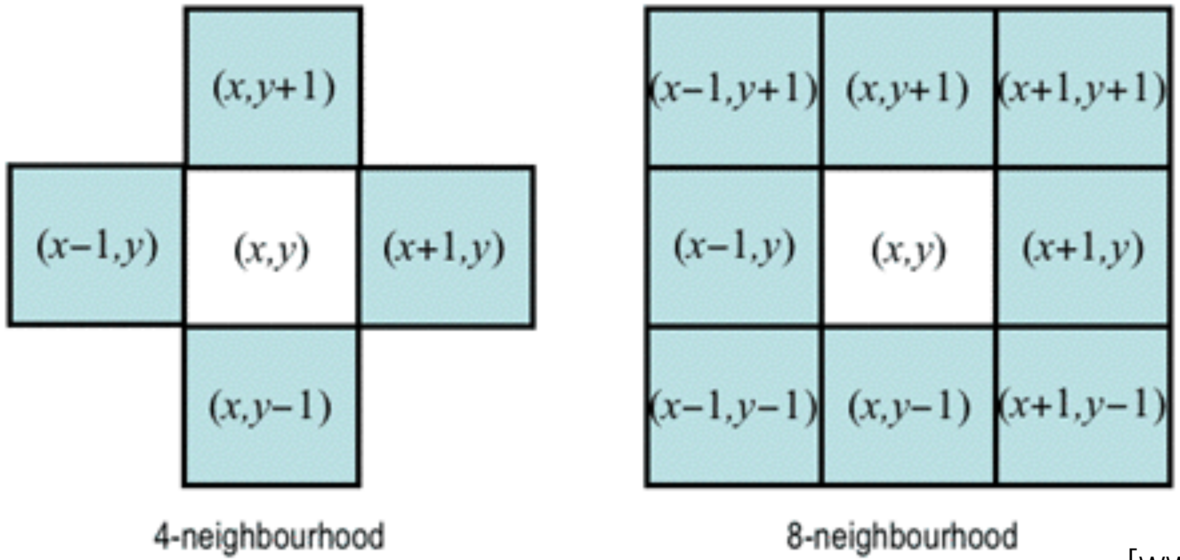
Minesweeper
Progression

- Recursive opening
- Checking 8-neighbourhood
  - naive vs offsets-based

Neighbourhoods in grids

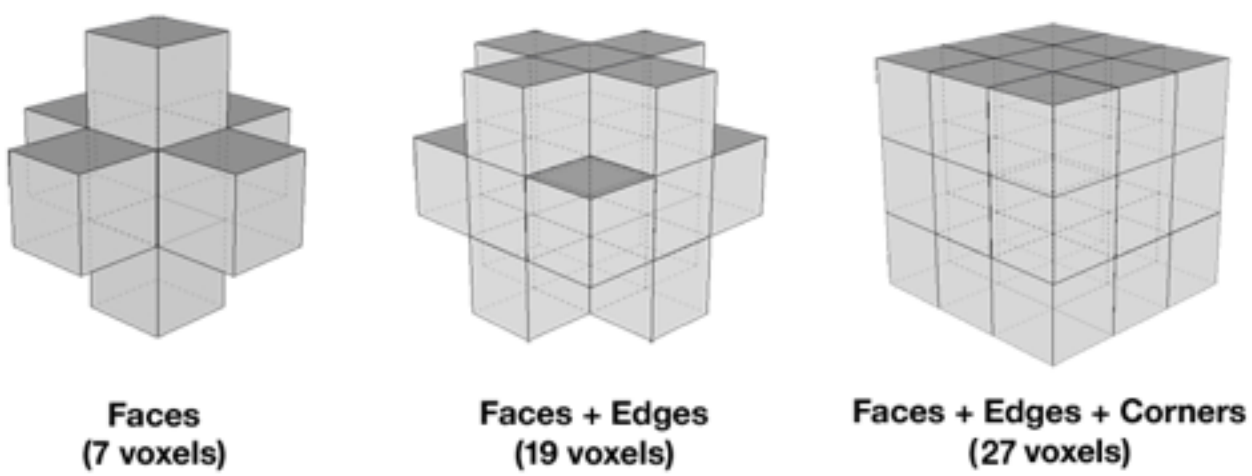
- esp. important in *image processing*

2D



[www.cs.auckland.ac.nz]

3D



[CPAC documentation]

8-neighbourhood in grids

How to go through it?

Naive

- 8 different statements
- error-prone
- bloated code
- scaling

☹

Alternative?

```
int Minesweeper::numAdjacentMines(int row, int col) const {
    int adjacentMines = 0;

    if (row != 0){
        adjacentMines += isTileMine(row-1,col);
    }
    if (col != 0){
        adjacentMines += isTileMine(row,col-1);
    }
    if (col != width-1){
        adjacentMines += isTileMine(row,col+1);
    }
    if (row != height-1){
        adjacentMines += isTileMine(row+1,col);
    }
    if ((row!=0) && (col!=0)){
        adjacentMines += isTileMine(row-1,col-1);
    }
    if ((row!=0) && (col!=width-1)){
        adjacentMines += isTileMine(row-1,col+1);
    }
    if ((row!=height-1) && (col!=0)){
        adjacentMines += isTileMine(row+1,col-1);
    }
    if ((row!=height-1) && (col!=width-1)){
        adjacentMines += isTileMine(row+1,col+1);
    }

    return adjacentMines;
}
```

[Piazza] (sorry!)

8-neighbourhood in grids

Offsets

- readability
- scalability
- clean code

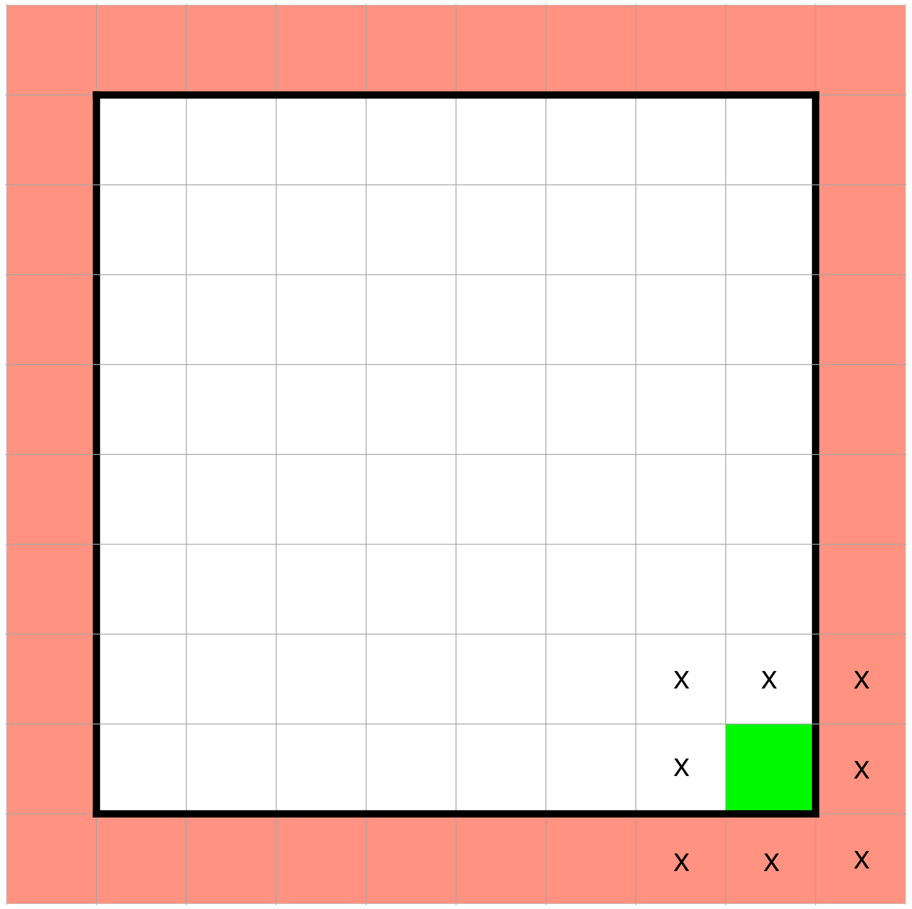
☺

```
// Naboene til en rute
const int row_offsets[8] = { -1, -1, -1,  0,  0,  1, 1,  1 };
const int col_offsets[8] = { -1,  0,  1, -1,  1, -1, 0,  1 };

...

for(int i = 0; i < 8; ++i) {
    const int neighbour_row = row + row_offsets[i];
    const int neighbour_col = col + col_offsets[i];
    ...
```

Grid boundaries

Tips

- minimize lines of code
  - often most robust
- consider a dedicated function
  - private/public?



Deleting default copying and =

An extra tip

shallow copy inappropriate + no need for these
⇒ delete the defaults

- avoids hard-to-see bugs

```
class Array {
    ...
    Array(const Array& other) = delete;
    Array& operator= (Array rhs) = delete;
    ...
};
```

Sometimes deeper meaning:

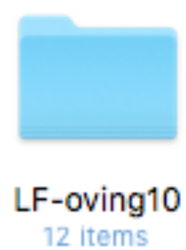
- Every student unique

Øving 9?

every game is unique?
“save game”?

What’s more?

A quick peek to øving 10..



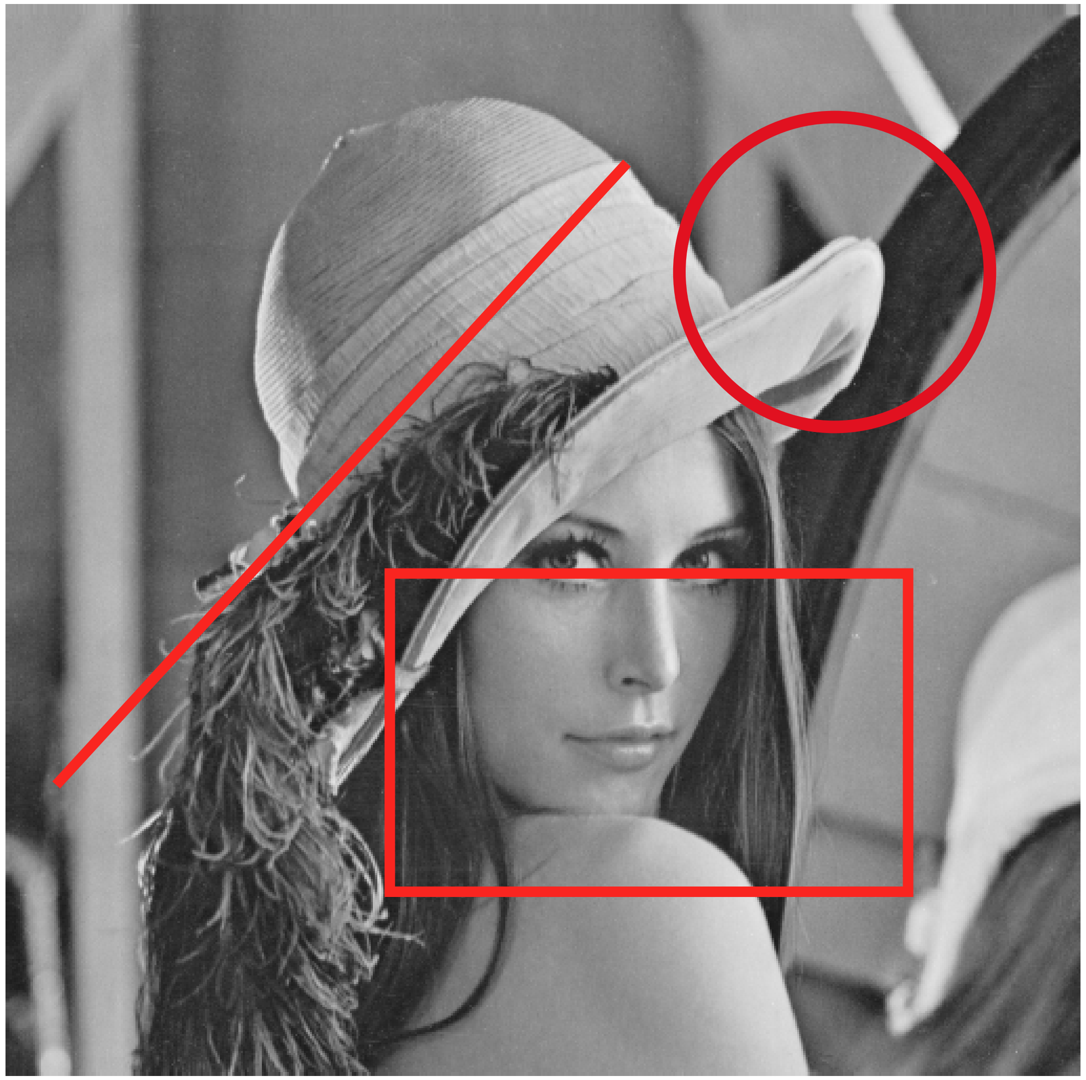
No; not so easy. :)

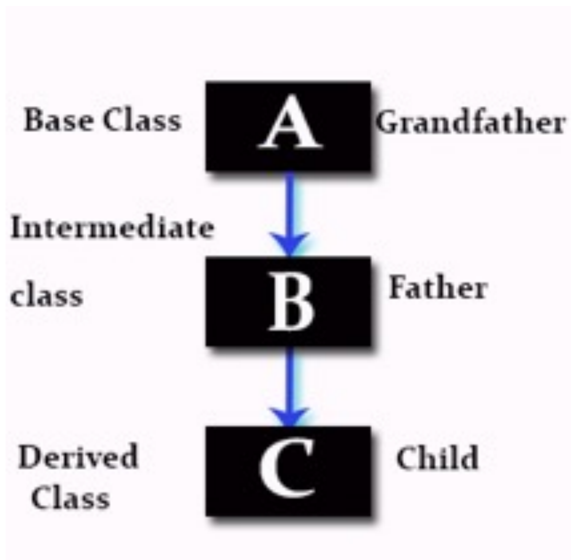
Øving 10: Drawing shapes

Sneak peek

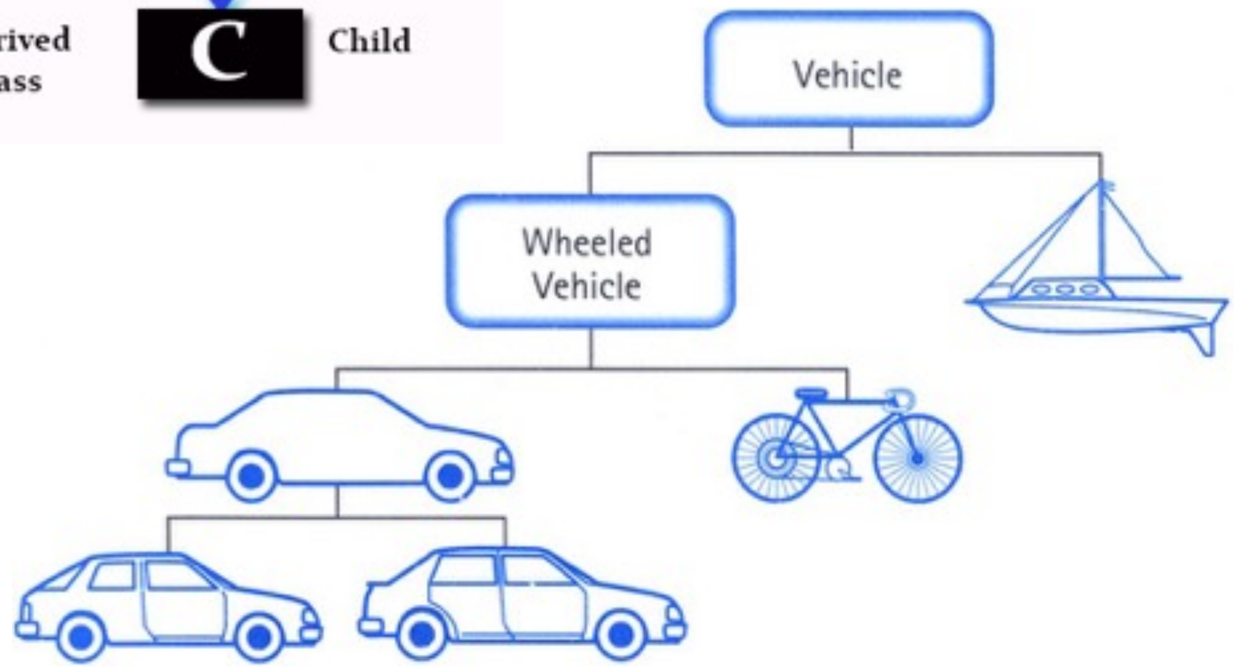
Topics

- Inheritance
  - Polymorphism
  - Virtual functions

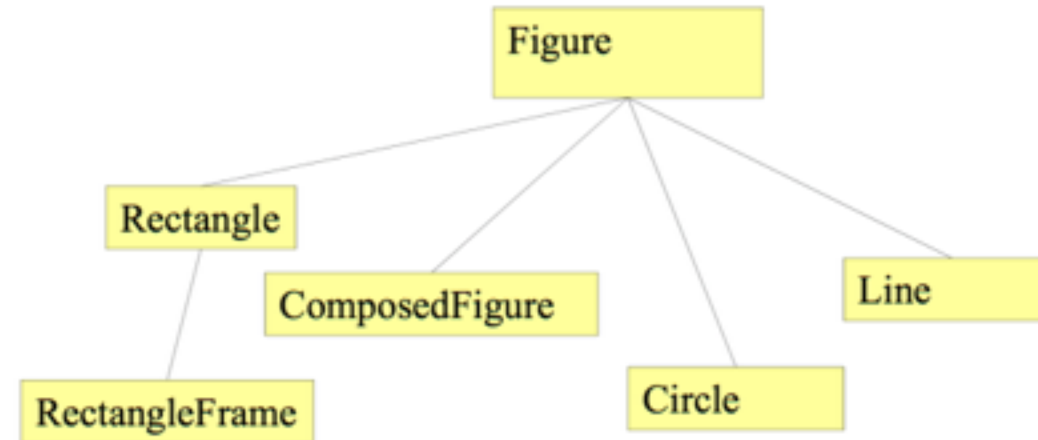




Inheritance




```
class Circle : public Shape {
...
};
```



Derived class

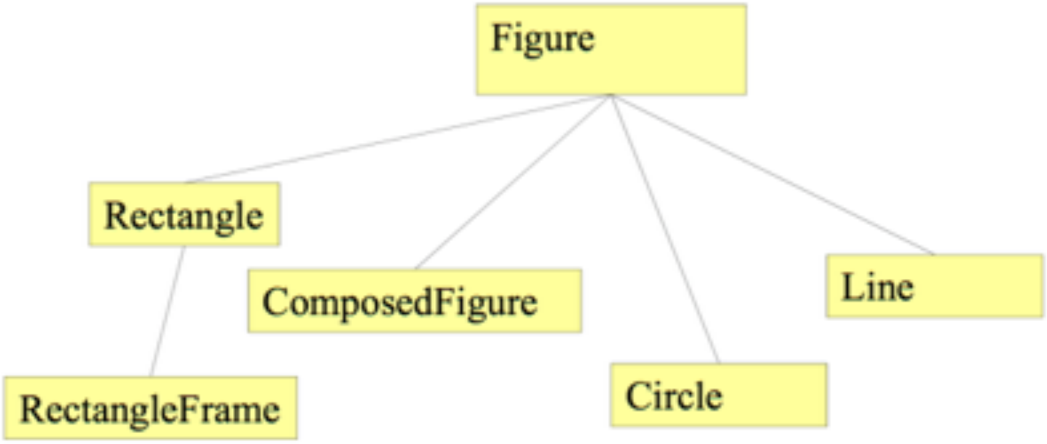
- "is a" baseclass 
- extra functionality
- different functionality

Purposes

- reusing existing code
- using *by* existing code
  - **shared interface**
- often elegant
  - often tricky / abused
  - better know what you're doing

Inheritance

virtual



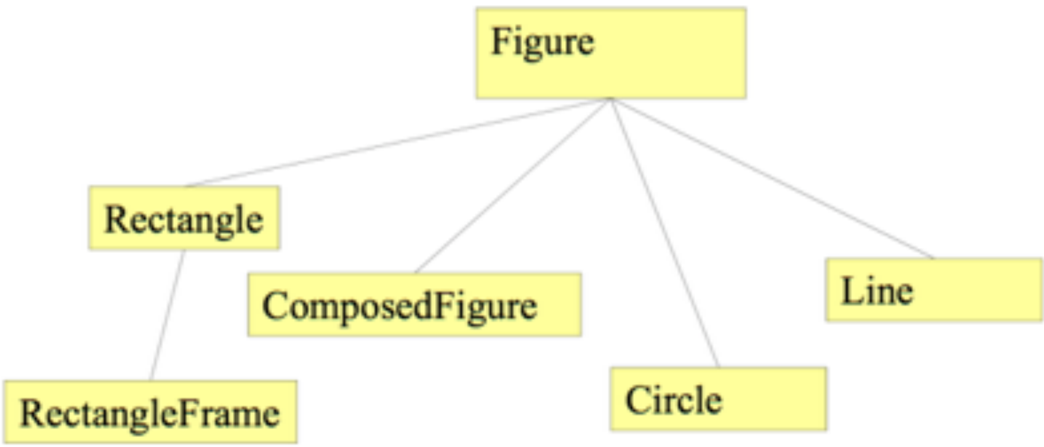

```
std::vector<const Figure*> figures;
figures.push_back(new Rectangle() )
figures.push_back(new Circle() )

for ( int i = 0; i < figures.size(); i++ ) {
 figures[i]->draw( img );
    }
```

- key concept
  - main argument for inheritance
- allows late binding
  - Code above draws implementations of **draw** by **Circle** and **Rectangle**, rather than that by **Figure**.

Why not all virtual?

Inheritance

virtual




```
std::vector<const Figure*> figures;
figures.push_back(new Rectangle() )
figures.push_back(new Circle() )

for ( int i = 0; i < figures.size(); i++ ) {
 figures[i]->draw( img );
    }
```

- Requires pointer or reference!!! → *Why?*
- The following code won't work as intended:

```
std::vector<const Figure> figures;
figures.push_back(Rectangle() )
figures.push_back(Circle() )

for ( int i = 0; i < figures.size(); i++ ) {
 figures[i].draw( img );
    }
```

Inheritance

pure **virtual**

```
class Base {
    // ...
    virtual void f() = 0;
    // ...
};
```

virtual method with no body and special syntax

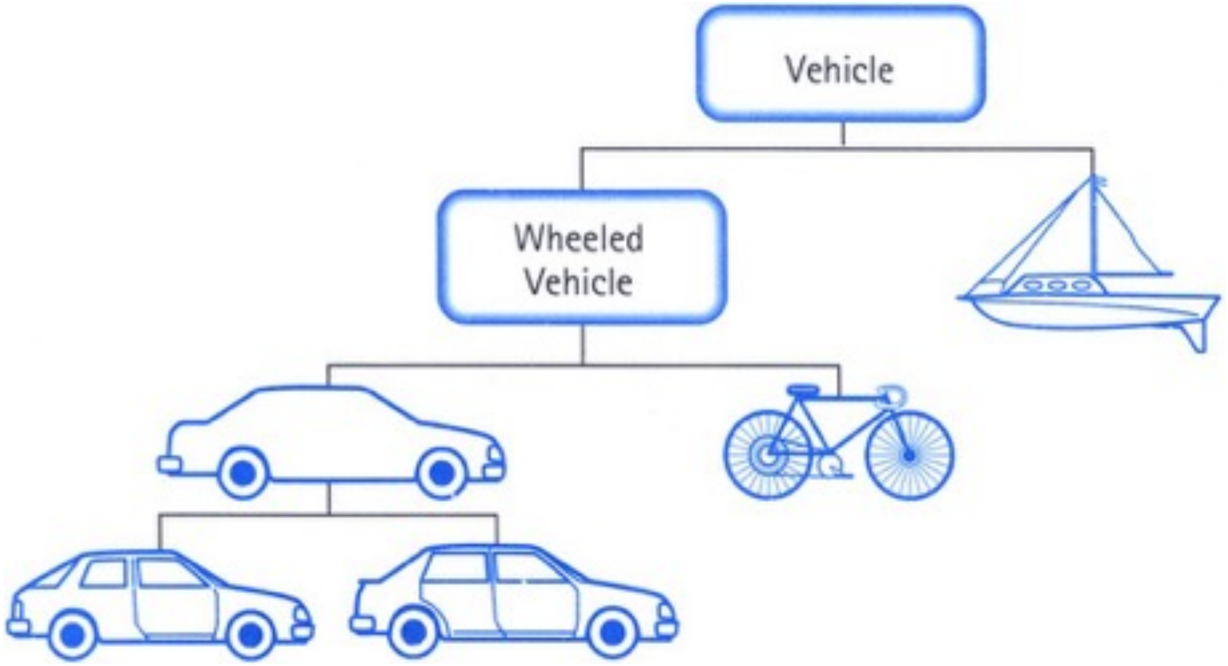
- class becomes *abstract*
  → no object can be made

In essence, defines **interface** that derived classes **need to provide**, so as not to be abstract.

Good to know

Inheritance

last note



Not inherited:
- constructors
- copy and operator=
- *private functions*

Summary

Øving 9

- short
- focus on good coding practices

Øving 10

- focus on inheritance & polymorphism
  - often vital for large projects