

Advanced C++ programming

♦ Introduction to C++

- C++: from C and beyond
- Classes, objects and lifetime (vs. JAVA)
- Oriented-Object Programming (inheritance, polymorphism)

♦ Memory management & object manipulation



- References, « copy » / « move » object construction
- Overloading operators

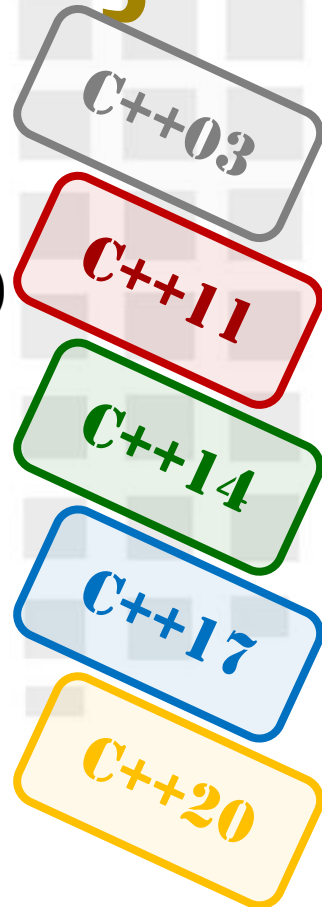
♦ Template vs OO programming

- Template functions and classes

♦ The Standard Template Library

- Containers, iterators and algorithms
- Using sequence & associative containers ...

♦ Smart pointers (STL & Boost)



References (1/6)

- Reference vs. pointer?
 - Link to an **existing** object (\simeq alias).
 \Rightarrow a reference always references an object (valid or not)

```
// i is an integer variable set to 10
int i = 10 ;

// j is another integer variable
int j = i ;

// r references i (i is an alias for r)
int& r = i ;

// Access to variable i through reference r
r = 20 ;
// now, i value is 20
```

```
// r is a not-initialized reference (error)
int& r ;

// r is a reference on i
int& r = i ;

// p1 is a pointer to variable i
int* p1 = &i ;

// p2 is a pointer to variable i (too)
int* p2 = &r ;
```



- Referenced **object lifetime** must be longer than a reference on it: e.g., a function that returns a reference on function local variable produces a non-valid reference!



References (2/6)

- **Basic need: editable function parameters?**
 - C++: parameter function passed by value
 - One idea: pass a reference as a parameter so that the referenced variable may be editable...

```
void increment (int& v) {  
    // Add one to the variable referenced by v  
    v++ ;  
}
```

```
// i is an integer variable set to 10  
int i = 10 ;  
  
// 'increment' expects a reference to an integer:  
// when 'i' is given when calling 'increment',  
// 'v' becomes a reference to 'i'  
increment (i) ;  
  
// now the value of 'i' is 11
```



References (3/6)

- Useful for non-editable parameters (1/2)?
 - How to pass a « big object » as a parameter ?
 - the full object is copied onto the stack when calling ► may be **costly**
 - One idea: pass a reference to such an object, but also specifying the compiler, **this reference shouldn't be used to modify this object ► `const` keyword**
 - Reduced cost : only an address is copied (like a pointer)
 - Though the reference, access to the object is read-only

```
void increment (const int& v) {  
    // Error when compiling : the reference v does not allow  
    // to modify the underlying  
    v++ ;  
}
```



References (4/6)

- Useful for non-editable parameters (2/2)
 - Convenient for reference to read-only object!
 - Though the reference, object attribute modifications are **forbidden**
 - Though the reference, only object « **const** » methods are **allowed**



compilation

```
class Point {
    int x ;
    // ft may modify the attributes
    // of one type Point object
    void ft () ;
}
```

```
void increment (const Point& p) {
    // Compilation error: the object referenced through p
    // is not editable using reference p
    p.x = 30 ;
    // Compilation error if « ft » method isn't
    // a « const » method i.e. declared as a method
    // not modifying the object on which it is called
    p.ft() ;
}
```



```
class Point {
    int x ;
    // ft cannot modify
    // Point attributes
    void ft () const ;
}
```

```
void increment (const Point& p) {
    // No compilation error
    p.ft() ;
}
```



References (5/6)

- Function returned value may be a reference (1/2)
 - C++: result is returned **by value** (copied onto the stack)
 - A reference may be returned (for cost optimization)
 - ▶ Be extremely careful to underlaying object lifetime

```
Point& Milieu (const Point& A , const Point& B) {
    Point I ((A.x+B.x)/2 , (A.y+B.y)/2) ;

    // A reference to local object I is returned. I being local, is destroyed
    // when the function ends: the returned reference does not reference
    // an existing valid object anymore.
    return I ;
}
```



runtime

```
Point A(10,10) , B(20,20) ;
Point& I = Milieu (A , B) ;
```

```
Point Milieu (const Point& A , const Point& B) {
    Point I ((A.x+B.x)/2 , (A.y+B.y)/2) ;

    // A copy of object I is put onto the stack. Object I, still local, is
    // destroyed when the function ends. The copied object can then be caught
    // as a result, by the caller, being a valid object.
    return I ;
}
```



```
Point A(10,10) , B(20,20) ;
Point I = Milieu (A , B) ;
```



References (6/6)

- Function returned value may be a reference (2/2)
 - A returned reference may be used as a L-value (editable)
 - If you don't want this behavior, use **const**!

```
int tab[20] ;

// Two functions returning a reference to a cell of 'tab'.
// 'tab' being global, its lifetime exceeds these functions
// ranges: returning a reference to 'tab' is not an issue.
int& elt1 (int i) { return tab[i] ; }
const int& elt2 (int i) { return tab[i] ; }

// Returning a non-const reference
int k = elt1(10) ; // k value is set to the tab[10] value
elt1(10) = 50 ; // tab[10] value is set to 50 (k is unchanged)

// Returning a const reference
int m = elt2(10) ; // m value is set to 50 (tab[10] value)
elt2(10) = 80 ; // Compiler error as elt2 returns a const reference:
                // the referenced object (11th cell of 'tab')
                // cannot be modified through this reference.
```



compilation



What is RVO?

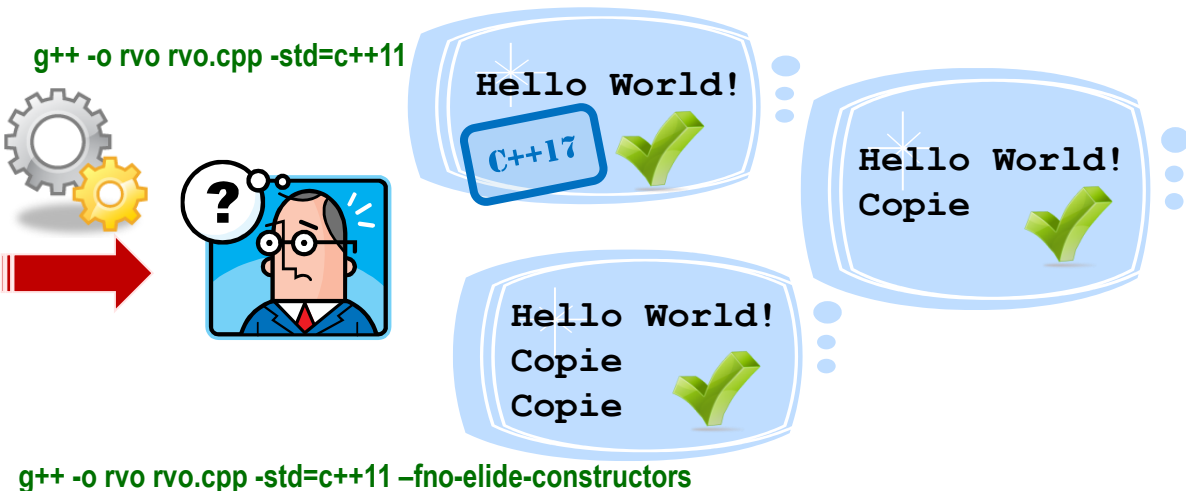
• RVO: « Return Value Optimization »

- Goal: prevent useless copies when object returned by value
- The compiler decides whether to perform **RVO**. If YES:
 - the “collecting” object is directly initialized from inside the function (when returning the value)
 - guaranteed with C++17 ► you can return results by value at “no cost”!

```
struct C {
    C () {}
    C (const C&) { cout << "Copie\n" ; }
};

C f() {
    return C();
}
```

```
int main() {
    cout << "Hello World!\n" ;
    C obj = f();
}
```



Operators

• C++ concept

- An operator performs specific computations on values.
- Built-in operators are provided but they can be **overloaded**.

• Unary operators (arithmetic, logical, ...)

- `+` `-` `*` `&` `~` `!` `++` `--` `->` `->*`

• Binary operators (arithmetic, logical, assignment, ...)

- `+` `-` `*` `/` `%` `^` `&` `|` `<<` `>>` `>` `<` `==` `!=` `+=` `-=` `*=` `/=` `%=` `^=` `|=` `&&` `||`

- `()` `,` `=` `,` `[]` `,` `<<` `,` `new` `new[]` `delete` `delete[]`

• You may define operators for your classes

- **coded** semantics vs. **commonly expected** semantics



Overloading operators (1/2)

- Goal: provide user-defined classes with operators
 - ▶ Initial operators arity and signatures must be **respected**
- Example: define a **+** operator for class **Point**

- Method signature

```
class Point {  
    int x , y ;  
  
    Point operator+ (const Point& A) {  
        return Point (A.x+this->x , A.y+this->y) ;  
    }  
}
```

- Function signature

```
Point operator+ (const Point& A, const Point& B) {  
    return Point (A.x+B.x , A.y+B.y) ;  
}
```

- Easy use & excellent readability!

```
Point A , B ;  
Point C = A + B ;
```



Overloading operators (2/2)

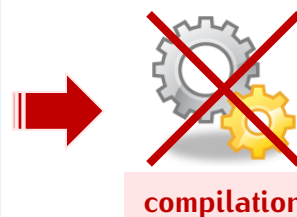
• Are the two definitions equivalent ?

```
struct Point {
    int x , y ;

    Point (int i) : x(i) , y(i) { }

    Point (int xp , int yp) : x(xp) , y(yp) { }

    Point operator+ (const Point& A) {
        return Point (A.x+this->x , A.y+this->y) ;
    }
}
```



```
Point A ;
int x(4) ;

Point C = A + x ;
Point C = x + A ;
```

no cast on method call

If you define these constructors,
the compiler can silently cast
one int into a Point

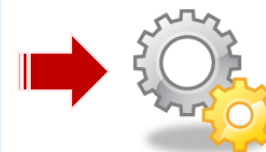
```
Point A = 4 ; // OK sauf si ctor
// explicit
```

```
struct Point {
    int x , y ;

    Point (int i) : x(i) , y(i) { }

    Point (int xp , int yp) : x(xp) , y(yp) { }

    Point operator+ (const Point& A , const Point& B) {
        return Point (A.x+B.x , A.y+B.y) ;
    }
}
```



```
Point A ;
int x(4) ;

Point C = A + x ;
Point C = x + A ;
```



Overloading << operator (1/2)

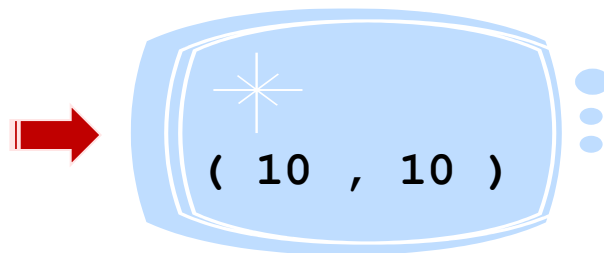
- How to “print” a user-defined object?
 - C++: insert an object in a I/O stream (output here)
 - ▶ Define the << operator for the user-defined class
 - Respect the << operator signature: it is a **function** only

```
#include <iostream>
```

```
class Point {  
public :  
    int x , y ;  
}
```

```
std::ostream& operator<< (std::ostream& flout , const Point& p)  
{  
    flout << "( " << P.x << " , " << P.y << " )" ;  
  
    return flout ;  
}
```

```
// Point B is being created  
Point B (10 , 10) ;  
  
// 'B' is inserted in the 'cout' stream  
// => displayed on screen  
std::cout << B << std::endl ;
```



Overloading << operator (2/2)

- Implementation variations
 - Encapsulation

```
class Point {
private :
    int x , y ;
public :
    int getX () const { return x ; }
    int getY () const { return y ; }
}
```

```
std::ostream& operator<< (std::ostream& flout , const Point& p)
{
    flout << "( " << P.getX() << " , " << P.getY() << " )" ;

    return flout ;
}
```

- Introducing friend functions
 - Function (not method) declared in the class with keyword **friend**
 - This function may access all attributes and methods (even private)

```
class Point {
private :
    int x , y ;
...
    friend std::ostream& operator<< (std::ostream& flout , const Point& p) ;
}
```

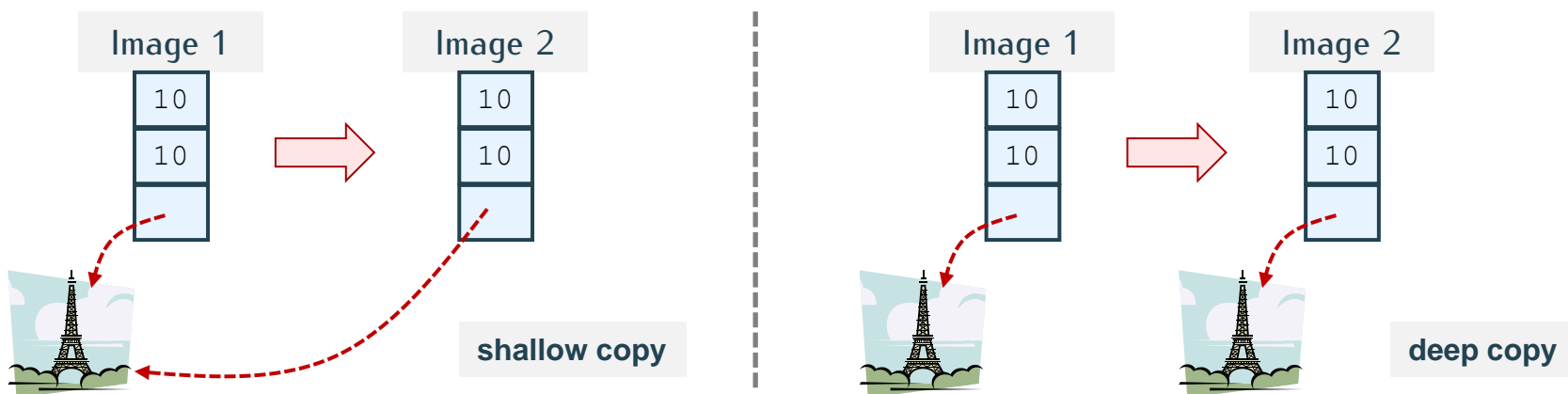
```
std::ostream& operator<< (std::ostream& flout , const Point& p){
    flout << "( " << P.x << " , " << P.y << " )" ;
    return flout ;
}
```



Copy constructor (1/2)

• Goal

- Build a new object, copying an existent one
- By default, for all classes, the compiler define such a constructor but it only performs a shallow copy



- You **must define your own copy constructor** if a shallow copy is not enough to copy objects of your class.

Copy constructor (2/2)

• How to define a copy constructor?

```
class Image {
    int    width , height ;
    byte* image ;

    Image (int w , int h) : width(w) , height(h) {
        image = new byte [w*h] ;
    }

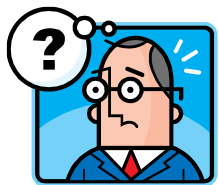
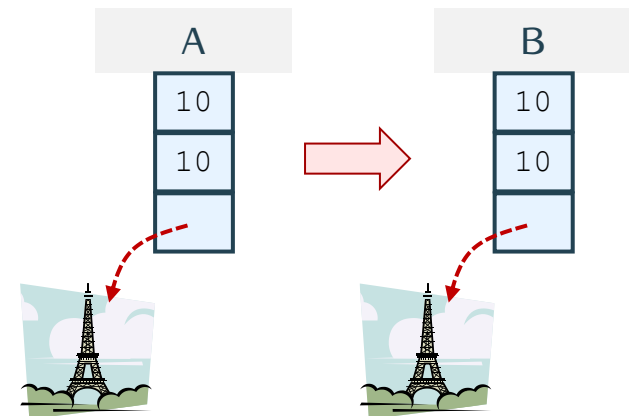
    Image (const Image& I) : width(I.width) , height(I.height) {
        image = new byte [I.width * I.height] ;
        memcpy (image , I.image , I.width*I.height) ;
    }

    ~Image () {
        delete [] image ;
    }
}
```

This is the used-defined copy constructor for Image class

```
// Creating an Image A
Image A (10 , 10) ;

// Creating another Image B
// being a copy of A through
// the copy constructor
Image B (A) ;
Image B = A ;
```



► But ... how to copy an object into an existing object?

This is an **assignment** operation not a construction one.



Operators

- **C++ concept**

- An operator performs specific computations on values.
- Built-in operators are provided but they can be **overloaded**.

- **Unary operators (arithmetic, logical, ...)**

- `+` `-` `*` `&` `~` `!` `++` `--` `->` `->*`

- **Binary operators (arithmetic, logical, assignment, ...)**

- `+` `-` `*` `/` `%` `^` `&` `|` `<<` `>>` `>` `<` `==` `!=` `+=` `-=` `*=` `/=` `%=` `^=` `|=` `&&` `||`

- `()` `,` `=` `,` `[]` `,` `<<` `,` `new` `new[]` `delete` `delete[]`

- **You may define operators for your classes**

- **Coded** semantics vs. **commonly expected** semantics



Overloading operator = (1/2)

- **Operator = takes only care of assignment**
 - Copy constructor is called when creating **new** objects.
 - Assignment « A = B » means **A and B already exist**

```
class Point {
    int x , y ;

    Point (int xp , int yp) : x(xp) , y(yp) { }

    Point (Point& P) : x(P.x) , y(P.y) { }

    Point& operator= (const Point& A) {
        if (this != &A) {
            x = A.x ;
            y = A.y ;
        }
        return *this ;
    }
}
```

Prevent self-assignment!

// Default ctor (error as not defined)
Point B ;

// Only existing constructor
Point B (10 , 10) ;

// Calling copy constructor
Point A (B) ;

// Calling copy constructor
Point C = B ;

// Calling assignment operator
B = A ;

- In this example, this = operator does a shallow copy assignment (not necessary as the compiler default assignment does that)



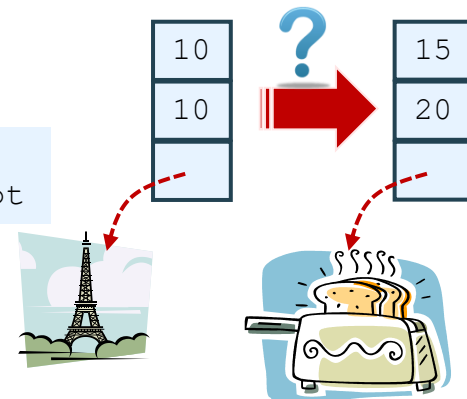
Overloading operator = (2/2)

- Two signatures for assignment operator

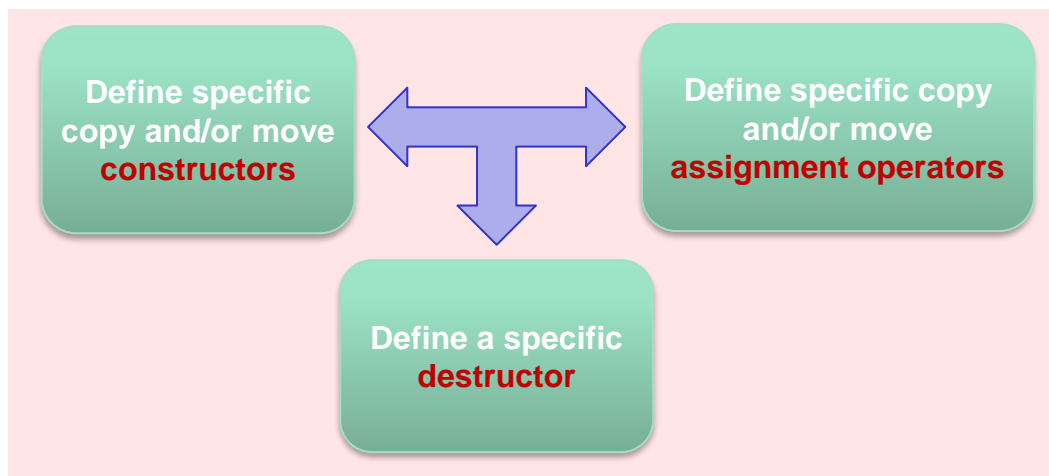
- « copy » or « move »

```
// Copy assignment
T& operator= (const T& other)
```

```
// Move assignment
T& operator= (T&& other) noexcept
```



- Respect the “rule of three”



► If a class needs to define one of them, the others two must also be defined



Some construction tips (1/2)



- Why delegating construction?
 - Prevent initialization code duplication
- Calling another constructor of the same class from one constructor



```
class Point {  
    int x , y ;  
  
public :  
    Point (int a , int b) : x(a) , y(b) {}  
  
    // Using another constructor from the same class  
    // to initialize an object:  
    // forbidden prior to C++11  
    Point () : Point (0 , 0) {}  
};
```

Delegation
the construction

```
// Creating a point A  
Point A (10 , 10) ;
```

```
// Creating a point B  
Point B () ;
```

- C++11 also allows:

```
class Point {  
    int x = 0 ;  
    int y = 0 ; ...
```



Some construction tips (2/2)



- Spell out what you really want ...
 - Make your code easier to understand
 - Let the compiler do the checking for you
 - ▶ “default”: actual default ctor / dtor / assignment?
 - ▶ “delete”: prohibit specific behaviors (copy, move, ...)

```
struct Point {
    int x , y ;

public :
    Point() = default ;
    Point (int a , int b) : x(a) , y(b) {}
} ;
```

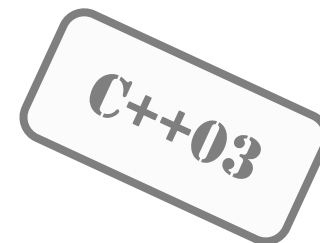
```
struct Point {
    int x , y ;

public :
    Point() = default ;
    Point (const Point &) = delete ;
    Point& operator= (const Point &) = delete ;
    Point (Point &&) = delete ;
    Point& operator= (Point &&) = delete ;
    ~Point() = delete ;
} ;
```

You prohibit explicitly, the creation of one `Point` from another one: **NonCopyable** class behavior



About initialization... (1/3)



• What it is possible to do:

- Built-in types: '=' or '()'

```
int n = 0 ;
void *p = 0 ;
char c = 'A' ;
```

```
int n (0) ;
void *p (0) ;
char c ('A') ;
```

- Classes & attributes : '()'

```
struct S {
    explicit S (int n , int m) : x(n), y(m) {}
private :
    int x , y ;
} ;
```

≠ POD
(Plain Old Data)

```
// Object initialization
S s1(0,1) ;
```

```
// Compilation error
S s2 = {0,1} ;
```

- Aggregates types: '{ }'

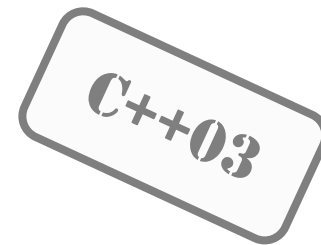
```
// POD arrays and structs are aggregates
int c1[2] = {0,2} ;
char c2[] = "message" ;
char c3[] = {'m','e','s','s','a','g','e','\0'} ;

struct S {
    int a , b ;
} ;
S s = {0,1} ;
```

variable = {...} ;

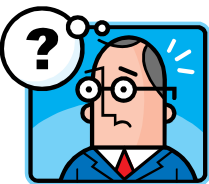


About initialization... (2/3)



- What it is not possible to do:

- Array direct initialization



```
class C {  
    int x[100] ;  
  
    C() ;    // no proper way to initialize x  
} ;
```

- Direct initialization of heap allocated array of POD

```
char *buff = new char[1024] ;    //no proper way to initialize the elements of buff
```

- Direct initialization of STL containers

```
vector <string> vs ;  
vs.push_back("alpha") ;  
vs.push_back("beta") ;  
vs.push_back("gamma") ;
```

➡ C++11 brings universal initialization {}



About initialization... (3/3)

• Universal initialization: {}

```
int a{0} ;
string s{"hello"} ;
string s2{s} ;           // copy construction

vector<string> vs{"alpha", "beta", "gamma"} ;
map<string, string> stars {
    {"Superman", "+1 (212) 545-7890"},
    {"Batman", "+1 (212) 545-0987"} ;

double *pd= new double [3] {0.5, 1.2, 12.99} ;

class C {
    int x[4] ;
public :
    C(): x{0,1,2,3} {}
} ;

vector<int> vi {1,2,3,4,5,6} ;
vector<double> vd {0.5, 1.33, 2.66} ;
```

No sign '=' before {...} ;

```
int a{} ;           // binary zeros for POD
string s{} ;

C obj{} ;           // default constructor
```

Direct **default values**
via {}

Construction through
an initialization list:
`std::initializer_list<T>`

• Attributes direct initialization

```
class C {
    int x = 7 ;           // class member initializer

    C() ;                 // x is initialized to 7 when the default ctor is invoked
    C(int y) : x(y) {}    // overrides the class member initializer
} ;
```

```
C c1 ;           // c1.x = 7
C c2(5) ;        // c2.x = 5
```



Practice



• Managing heap-allocated memory...



- Constructor & allocating resources
- Destructor & releasing resources

• What about copying objects?



- Construction?
- Assignment?

• Going further...



- Runtime user interaction for object type selection and creation
- '<<' operator & streams

