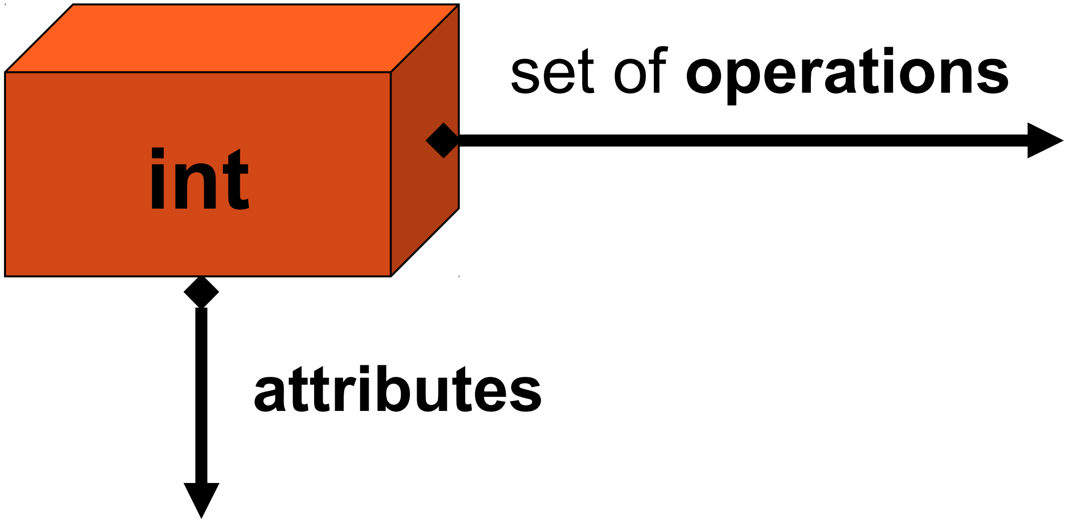
Structs and Classes

- Version 1: Dr. Ofir Pele
- Version 2: Dr. Miri Ben-Nissan
- Version 3: Dr. Erel Segal-Halevi

Abstract Data Type (ADT)



int

set of **operations**

attributes

Operations:

- numerical operators
- logical operations
- bit operations
- I/O operations

Attributes:

- 4 bytes.
- Integer numbers.

Data Types define the way you use storage (memory) in the .programs you write

Classes

In C++ we use **classes** to define new ADTs:

```
class ClassName
{
  //attributes and operations
};
```

Objects are instances of classes.

objects are to classes what
variables are to types.

	C	C++	Java
Keyword	struct	class or struct	class
Filename	any (usually: *name*.h)	any (usually: *name*.hpp *name*.cpp)	*name*.java
Attributes	Yes	Yes	Yes
Methods	No	Yes	Yes
Access control	all public	public or private	public or private
Memory	stack	stack	heap
Operators	No	Yes	No

structs and classes

Where did structs go?

- In C++ class==struct, except that by default struct members are **public** and class members are **private**:

```
struct MyStruct
{
    int x;
};
class MyClass
{
    int x;
};
```

```
int main()
{
    MyStruct s;
    s.x = 1; // ok
    MyClass c;
    c.x = 1; // error
}
```

structs & classes *(folder 1)*

All of these are the same:

```
struct A
{
    int x;
};


struct A
{
    public:
    int x;
};


class A
{
    public:
    int x;
};
```

All of these are the same (and useless):

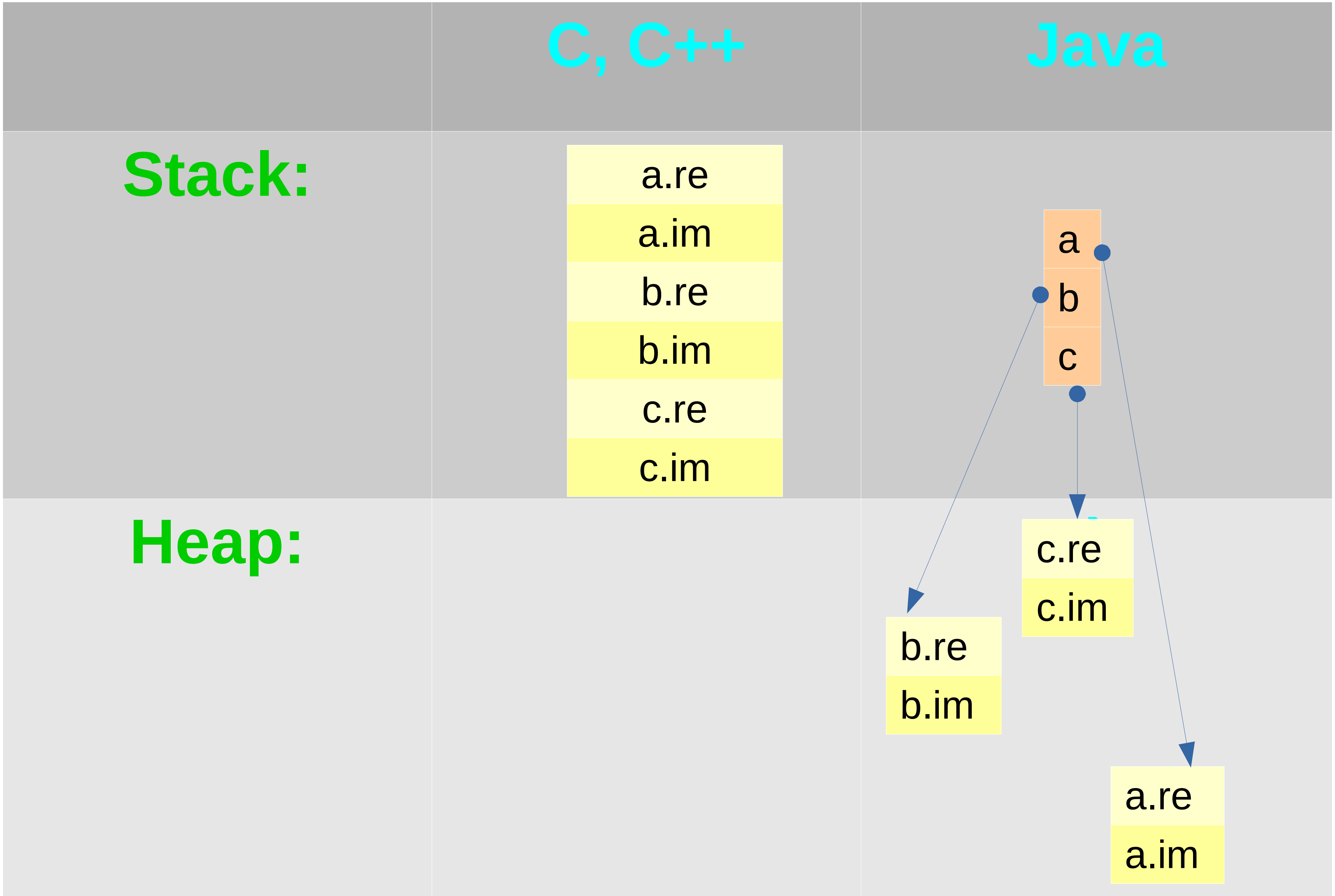
```
class A
{
    int x;
};


class A
{
    private:
    int x;
};


struct A
{
    private:
    int x;
};
```

C	C++	Java

struct Cplx {
 double re, im;
};
 	
Cplx sumCplx(
Cplx a, Cplx b)
{…}
	class Cplx {
	  double re, im;
	public:
	 Cplx sum
	  (Cplx b) {…}
	 Cplx
	  (double re,
	  double im) {…}
	};
		class Cplx {
		 private double
		    re, im;
		 public Cplx sum
		    (Cplx b) {…}
		 public Cplx
		 (double re,
		  double im) {…}
		};

C	C++	Java

int main() {
    Cplx a;
    a.re=5;
    a.im=10;
}
	int main() {
	    Cplx a(5,10);
	}
		void main(…) {
		    Cplx a =
		    new
		      Cplx(5,10);
		}



	C, C++	Java
Stack:	a.re a.im b.re b.im c.re c.im	a b c
Heap:		c.re c.im b.re b.im a.re a.im

```
int main () {  Cplx a, b, c;  };
```

Two ways to implement a method *(folder 2)*

```
class Complex {
  double re, im;
public:
 Complex () { re=0; im=0; }// inline constructor
 Complex (double re, double im);   // “outline”

 Complex sum  (Complex b) { return
Complex(a.re+b.re, a.im+b.im); } // inline method
 Complex diff (Complex b);      // “outline”
};
```

Implementing methods out-of-line

```
Complex::Complex (double re, double im) {

    this→re = re;
    this→im = im;
}
```

Scope operator

The address of the instance for which the member method was invoked.

```
Complex Complex::diff(Complex b) {
    return Complex(a.re-b.re, a.im-b.im);
}
```

Class Basics – member/static *(folder 3)*

```
class List
{
public:
   static int getMaxSize();
   int getSize();
   // static int max_size=1000; //error! (declare outside)
   int size=0;
};

int List::max_size=1000; //ok, in one cpp file

int main()
{
   List l;
   l.getSize();
   List::getMaxSize();
   l.getMaxSize(); //compiles ok, but bad style
}
```

this

```
static int List::getMaxSize() //no this!
{
   return this->size; // compile error!
   return max_size; // ok
}
int List::getSize()
{
   return this->size; //ok
}
```

C++ Laws of Construction and Destruction

1. Every object must be **constructed** before it is used.

- Stack object: when it is defined.
- Heap object: when it is created.

2. Every object must be **destructed** after it stops being of use.

- Stack object: when gets out of scope.
- Heap object: when it is deleted.

What file-names should we use?

- The C++ compiler does not care how your files are called.
- It is common to put a class declaration in file ClassName.hpp (or ClassName.h) and the class implementation in file ClassName.cpp.
- Why is it better?
  - Hiding implementation details.
  - Saving comiplation time – when you have a good **Makefile** *(see folder 4)*.

Constructors *(folder 4)*

```
class MyClass
{
public:
①   MyClass();
②   MyClass( int i );
③   MyClass( double x, double y );
    ...
};

int main() {
    MyClass a; // Calls 1
    MyClass b {5}; // Calls 2
    MyClass c {1.0, 0.0}; // Calls 3
}
```

Constructors and Arrays *(folder 4)*

```
class MyClass
{
public:
1   MyClass();
2   MyClass( int i );
3   MyClass( double x, double y );
    ...
};

int main() {
    MyClass a[5]; // Calls 1 five times
    MyClass b[5] {11, 22}; // Calls 2 two times
    MyClass c[5] { {11,22}, 33}; // Calls 3 then 2
}
```

Constructors – parameterless ctor

```
class MyClass {
public:
   MyClass(); // parameterless ctor.
   //...
};
//...
int main() {
   MyClass a; // parameterless ctor called
   // …
}
```

Constructors – default parameterless ctor

```
class MyClass {
public:
   // No ctors
};

int main() {
  MyClass a; // default parameterless ctor:
  // Calls parameterless ctors of members
}
```

Constructors – no default parameterless ctor

```
class MyClass {
public:
   MyClass(int x); // no parameterless ctor.

};

int main() {
   MyClass a;     // compiler error –
   MyClass b[5];  // no parameterless ctor.
}
```

Constructors – explicit default parameterless ctor

```
class MyClass {
public:
   MyClass(int x);
   MyClass() = default;
};

int main() {
   MyClass a; // default parameterless ctor
}
```

Constructors – deleted default parameterless ctor

```
class MyClass {
public:
   MyClass() = delete;
};

int main() {
   MyClass a; // compiler error –
              // no parameterless ctor.
}
```

Destructors

Goal: Ensure proper “cleanup”:

- Free allocated memory;
- Close opened files or db connections;
- Notify related objects, etc.

Use: Called for:

- A stack object – when it goes out of scope.
- A heap object – when it is explicitly deleted.

Destructors *(folder 5)*

```
#include <cstdlib>
class MyClass
{
public:
   MyClass();  // constructor
   ~MyClass(); // destructor
private:
   char* _mem;
};
MyClass::MyClass()
{
   _mem = new char[1000];
}
MyClass::~MyClass()
{
   delete[] _mem;
}
```

```
  int main()
  {
⇨     MyClass a;
      if( ... )
      {
⇨         MyClass b;

⇨     }

⇨ }
```

Destructors – common errors *(folder 5)*

1. Forgetting to write a destructor – causes a memory leak.

2. Shallow copy – causes destructor to be called twice.

C struct and functions

```
struct IntList;
typedef struct IntList IntList;
IntList* intListNew();
void intListFree(      IntList* List );
void intListPushFront( IntList* List, int x);
void intListPushBack(  IntList* List, int x);
int  intListPopFront(  IntList* List );
int  intListPopBack(   IntList* List );
int  intListIsEmpty(   IntList const* List);

typedef void (*funcInt)( int x, void* Data );
void intListMAPCAR(    IntList* List,
                      funcInt Func, void* Data );
```

C++ Class

In header file:

```
class IntList
{
public:
  IntList();
  ~IntList();
  void pushFront(int x);

  void pushBack(int x);
  int popFront();
  int popBack();
  bool isEmpty() const;
private:
  struct Node
  {
      int value;
      Node *next;
      Node *prev;
  };
  Node* m_start;
  Node* m_end;
};
```

Classes & Memory allocation

Consider this C++ code

```
main()
{
    IntList L;
    …
}
```

What is the difference?

Compare to C style:

```
main()
{
    IntList* L =
    intListNew()
    …
    intListFree(L)
}
```

Memory allocation in C

```
IntList* L =
(IntList*)malloc(sizeof(IntList));
```

Does not call constructor!

Internal data members are not initialized

```
free(L);
```

Does not call destructor!

Internal data members are not freed

Memory allocation in C++

Special operators:

```
IntList *L = new IntList;
```

1. Allocate memory
2. Call constructor

```
delete L;
```

3. Call destructor
4. Free memory

new

Can be used with any type:

```
int *i = new int;
char **p = new (char *);
```

- new is a global operator
- new ***expression*** invokes the new ***operator*** to allocate memory, and then calls ctor
- Can be overloaded (or *replaced*)
- By default, failure throws exception. Can be changed.
- See <new> header

New & Constructors

```
class MyClass
{
public:
 ① MyClass();
 ② MyClass( int i );
 ③ MyClass( double x, double y );


};


MyClass* a;
a = new MyClass; // Calls  ①
a = new MyClass {5}; // Calls ②
a = new MyClass { 1.0, 0.0 }; // Calls ③
```

New & arrays

To allocate arrays, use

```
int *a = new int[10]; // array of 10
                      //ints
size_t n = 4;
IntList *b = new IntList[n];
             // array of n IntLists
```

Objects in allocated array must have an argument-less constructor!

Delete & arrays

Special operation to delete arrays

```
int *a = new int[10];
int *b = new int[10];

delete [] a; // proper delete command
delete b;    // apparently works,
// but may cause segmentation fault
// or memory leak (folder 6)
```

Allocate array of objects w/o def. cons.

```
size_t n = 4;
MyClass **arr = new MyClass *[n];
// array of n pointers to MyClass (no
// cons. is invoked)

for (size_t i=0; i<n; ++i)
{
   arr[i] = new MyClass (i);
   // each pointer points to a MyClass
   // object allocated on the heap, and
   // the cons. is invoked.
}
```

Free an allocated array of pointers to objects on the heap

```
size_t n = 4;
for (size_t i=0; i<n; ++i)
{
   delete (arr[i]);
   // invoked the dest. of each MyClass
   // object allocated on the heap, and
   // free the memory.
}
delete [] arr;
// free the memory allocated for the
// array of pointers. No dest. is invoked
```