Programming 2

Object Oriented Programming

Daniel POP

Course Outline

1. What is object-oriented programming and C++
2. Data types
3. Object-oriented concepts: classes, objects, messages
4. Classes
5. Type of objects
6. Class relationships
7. Inheritance. Multiple inheritance. Class hierarchies
8. Exception handling
9. Generic programming. Template class & functions
10. C++ Standard Template Library (STL)
11. Object-oriented analysis and design

Unit 1

Agenda

1. How did we get here?
2. What is Object-Oriented Programming?
3. Short history of OOP
4. What is C++?
5. Short history of C++

The Road to Object-Oriented Programming and Beyond

- Unstructured programming
- Procedural programming
- Modular programming
- Data abstraction
- Object-oriented programming
- Generic programming
- Functional programming
- Logic and symbolic programming

Unstructured Programming

- Simple / small application consisting of **one main program**
- Program = sequence of commands (statements) which modify global data
- Machine code
- Drawback: unmanageable as program gets bigger; a lot of copy/pasted code
- Example: in Assembler, C, Pascal etc.

test.c

```
// data
void main(int argc, char* argv[]) {
    // local data
    // statements
}
```

Procedural (Structured) Programming

- Based on the notion of procedure (function)
- **Decide which procedures you want; use the best algorithms you can find.**
- Drawback: handling different the data structures and the algorithms operating on data
- Example: programs written using C, Pascal, Fortran, Algol

test.c

```
double sqrt(double arg1)  {
    ....
    ....
}

void main(int argc, char* argv[]) {
    // local data
    f(10, data1);
    // statements
    sqrt(14.6);
}
```

```
void f(double arg1, sometype arg2)  {
    ....
    sqrt(arg1);
    ....
}
```

Modular Programming

- Program size grows => Organizing data
- **Decide which modules you want; partition the program so that data is hidden in modules. (<u>data hiding principle</u>)**
- Drawback: only one state per module + each module exists only once in one program user-defined types doesn't behave the same way as built-in types
- Example: programs written in C, Modula-2

<u>stack.h</u>

```
// declaration of the interface of module
char pop();
void push(char);
const stack_size = 100;
```

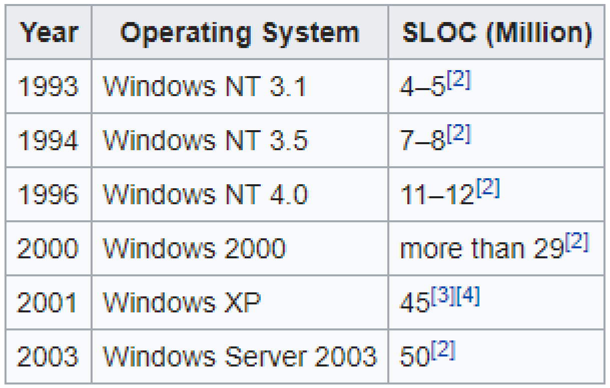
<u>main.c</u>

```
#include "stack.h"
void some_function() {
    push('c');
    char c = pop();
    if (c != 'c') error("impossible");
}
```

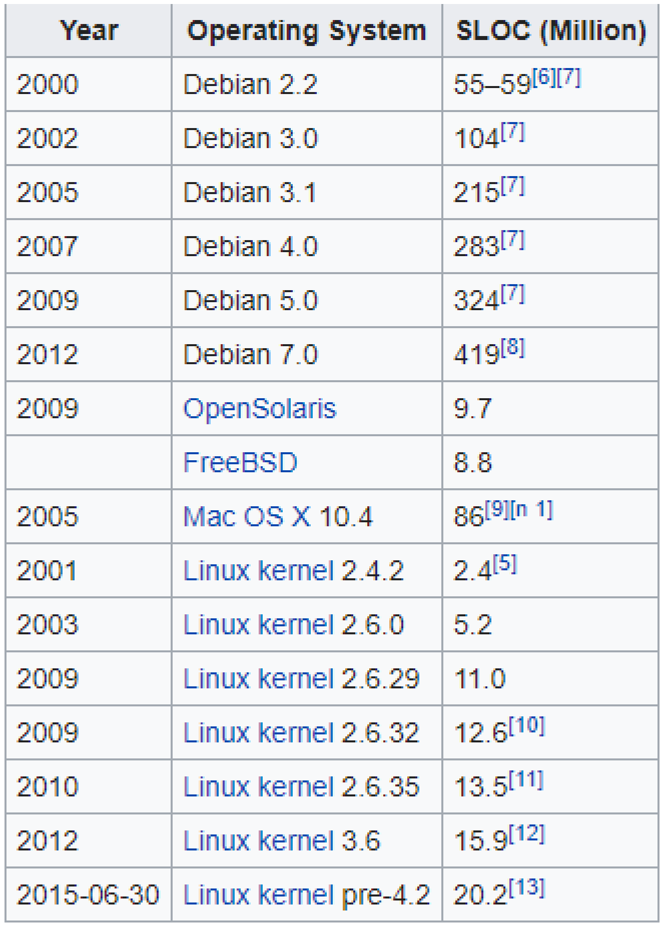
<u>stack.c</u>

```
#include "stack.h"
// ''static'' means local to this file/module
static char v[stack_size];
static char* p = v; // the stack is initially empty
char pop() {
        // check for underflow and pop
}
void push(char c) {
        // check for overflow and push
}
```

Software systems size



Year	Operating System	SLOC (Million)
1993	Windows NT 3.1	4–5[2]
1994	Windows NT 3.5	7–8[2]
1996	Windows NT 4.0	11–12[2]
2000	Windows 2000	more than 29[2]
2001	Windows XP	45[3][4]
2003	Windows Server 2003	50[2]



Year	Operating System	SLOC (Million)
2000	Debian 2.2	55–59[6][7]
2002	Debian 3.0	104[7]
2005	Debian 3.1	215[7]
2007	Debian 4.0	283[7]
2009	Debian 5.0	324[7]
2012	Debian 7.0	419[8]
2009	OpenSolaris	9.7
	FreeBSD	8.8
2005	Mac OS X 10.4	86[9][n 1]
2001	Linux kernel 2.4.2	2.4[5]
2003	Linux kernel 2.6.0	5.2
2009	Linux kernel 2.6.29	11.0
2009	Linux kernel 2.6.32	12.6[10]
2010	Linux kernel 2.6.35	13.5[11]
2012	Linux kernel 3.6	15.9[12]
2015-06-30	Linux kernel pre-4.2	20.2[13]

Source: https://en.wikipedia.org/wiki/Source_lines_of_code

Modular Programming

- Program size grows => Organizing data
- **Decide which modules you want; partition the program so that data is hidden in modules. (<u>data hiding principle</u>)**
- Drawback: only one state per module + each module exists only once in one program user-defined types doesn't behave the same way as built-in types
- Example: programs written in C, Modula-2

<u>stack.h</u>

```
// declaration of the interface of module
char pop();
void push(char);
const stack_size = 100;
```

<u>main.c</u>

```
#include "stack.h"
void some_function() {
    push('c');
    char c = pop();
    if (c != 'c') error("impossible");
}
```

<u>stack.c</u>

```
#include "stack.h"
// ''static'' means local to this file/module
static char v[stack_size];
static char* p = v; // the stack is initially empty
char pop() {
        // check for underflow and pop
}
void push(char c) {
        // check for overflow and push
}
```

Data Abstraction (I)

- Based on user-defined types that behave the same way as built-in types (Abstract Data Types)
- **Decide which types you want; provide a full set of operations for each type.**
- Drawback: no way of adapting an ADT to new uses except modifying its definition (need for "type fields" that distinguish between various instantiations)
- Example: programs written using Ada, C++, Clu, Java

complex.h

```
class complex {
        double re, im;
public:
        complex(double r, double i) { re=r; im=i; }
        complex(double r) { re=r; im=0; }
        friend complex operator+(complex, complex);
        friend complex operator-(complex, complex);
        friend complex operator-(complex);
        // ...
};
```

main.c

```
void f() {
   int ia = 2,ib = 1/a;
   complex a = 2.3;
   complex b = 1/a;
   complex c = a+b*complex(1,2.3)

   c = -(a/b)+2;
}
```

Data Abstraction (II)

- Drawback: no way of adapting an ADT to new uses except modifying its definition (e.g using “type fields” that distinguish between various instantiations)
- Example:

shape.h

```
enum kind { circle, triangle, square };

class shape {
    point center;
    color col;
    kind k;
    // representation of shape
public:
   point where() { return center; }
   void move(point to) { center = to; draw(); }
   void draw();
};
```

shape.cpp

```
void shape::draw() {
    switch (k) {
         case circle: // draw a circle
                      break;

         case triangle: // draw a triangle
                      break;

         case square: // draw a square
                      break;

         default: // unknown shape
    }
}
```

Object-Oriented Programming

- World of interacting objects, each one managing its own state
- **Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance.**
- Example: programs written using Simula, C++, Java, Eiffel, Smalltalk etc.

shape.h

```
class shape {
     point center;
     color col;
     // representation of shape
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
};
```

rectangle.h

```
class rectangle : public shape {
     double width, height;
     // representation of rectangle
public:
     void draw() {
          // draw the rectangle
     }
};
```

Generic Programming

- Express algorithms independently of representation details
- **Decide which algorithms you want; parameterize them so that they work for a variety of suitable types and data structures.**
- Example: programs written using C++, Java (≥ 1.5)

stack.h

```
template<class T> class stack {
    T* v;
    int max_size, top;
Public:
    stack(int s);
    ~stack();
    void push(T v);
    T pop();
};
```

file.cpp

```
void f() {
    stack<char> schar;
    stack<complex> scomplex;
    stack<list<int>> slistint;

    schar.push('c');
    if(schar.pop()!='c') throw Impossible();
    scomplex.push(complex(3, 2));
}
```

Functional Programming

- Express processing in terms of mathematical functions and their composition
- **Decide what processing you need and express it as a composition of functions.**
- Favor recursive processing
- Avoids state and mutable data
- Example: Lisp (and derived), Scala, F#, Haskell

<u>file.cpp</u>

```
auto lambda_echo = [](int i ) { std::cout << i << std::endl; };

std::vector<int> col{20,24,37,42,23,45,37};

for_each(col, lambda_echo);
```

Logic and Symbolic Programming

The logic programming paradigm views computation as automated reasoning over a body of knowledge. Facts about the problem domain are expressed as logic formulae, and programs are executed by applying inference rules over them until an answer to the problem is found, or the set of formulae is proved inconsistent.

Symbolic programming is a paradigm that describes programs able to manipulate formulas and program components as data.[3] Programs can thus effectively modify themselves, and appear to "learn", making them suited for applications such as artificial intelligence, expert systems, natural-language processing and computer games.

Languages that support this paradigm include Lisp and Prolog

Source: https://en.wikipedia.org/wiki/Programming_paradigm

Imperative Programming vs Declarative Programming

- Imperative – how to solve the problem

```
struct ResultDataStructure {
     string Name;
     string Address;
}

list<ResultDataStructure> computeResult() {
    list<ResultDataStructure> list;
    for(int i=0; i<Students.length; i++) {
        if (0==strcmp(Students[i].Major, “CS”) {
           result.add(new ResultDataStructure(Students[i].getName(),
                                               Students[i].getAddress());
     }
     return list;
}
```

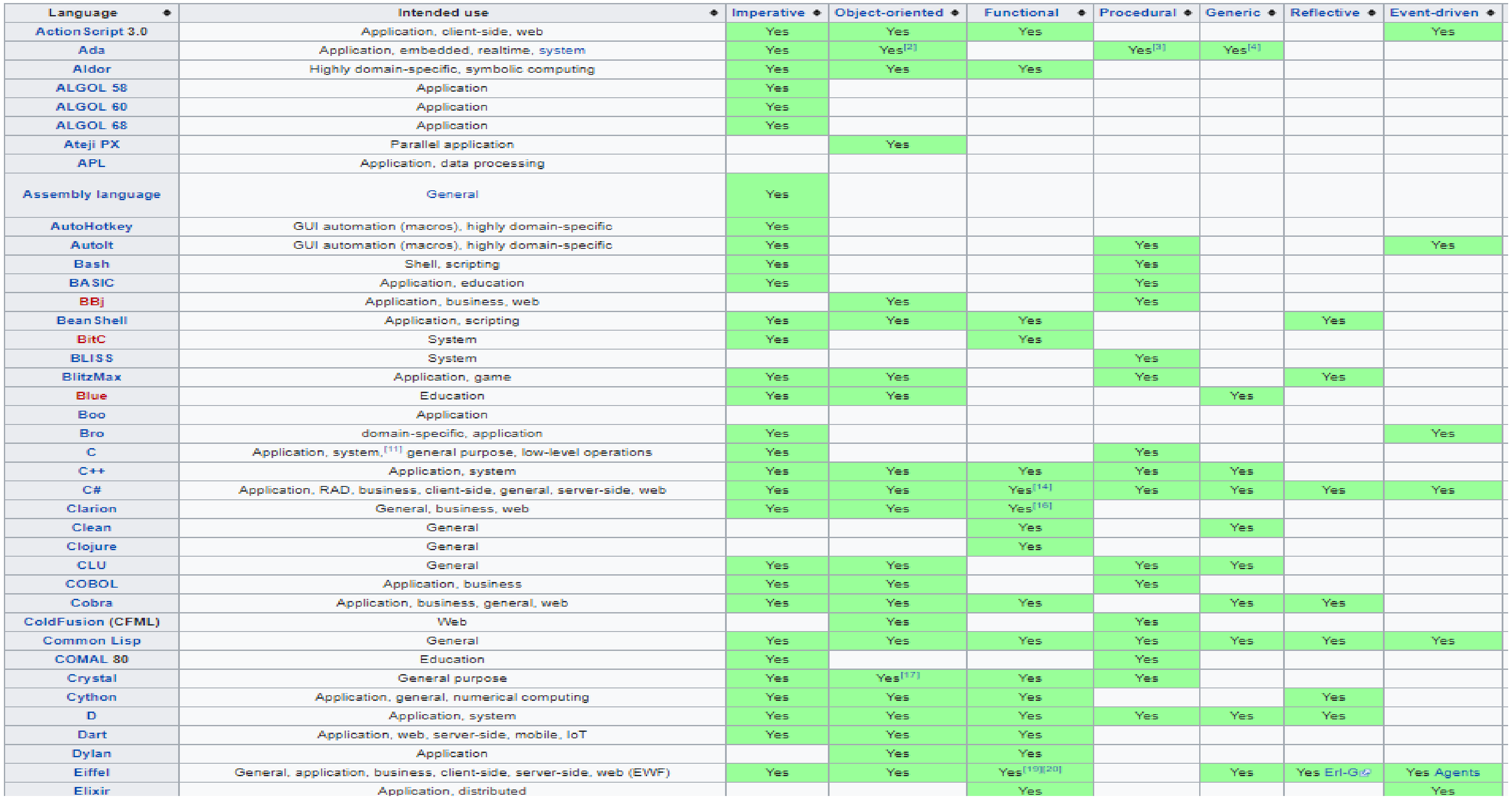
- Declarative programming – what the problem is

```
Ex: SQL, SPARQL, OWL, Prolog
SELECT Name, Address FROM Students WHERE Major=‘CS’
```

```
SELECT ?name ?address
WHERE { ?person a foaf:Person .
        ?person foaf:name ?name .
        ?person foaf:mbox ?address }
```

Classification of programming languages

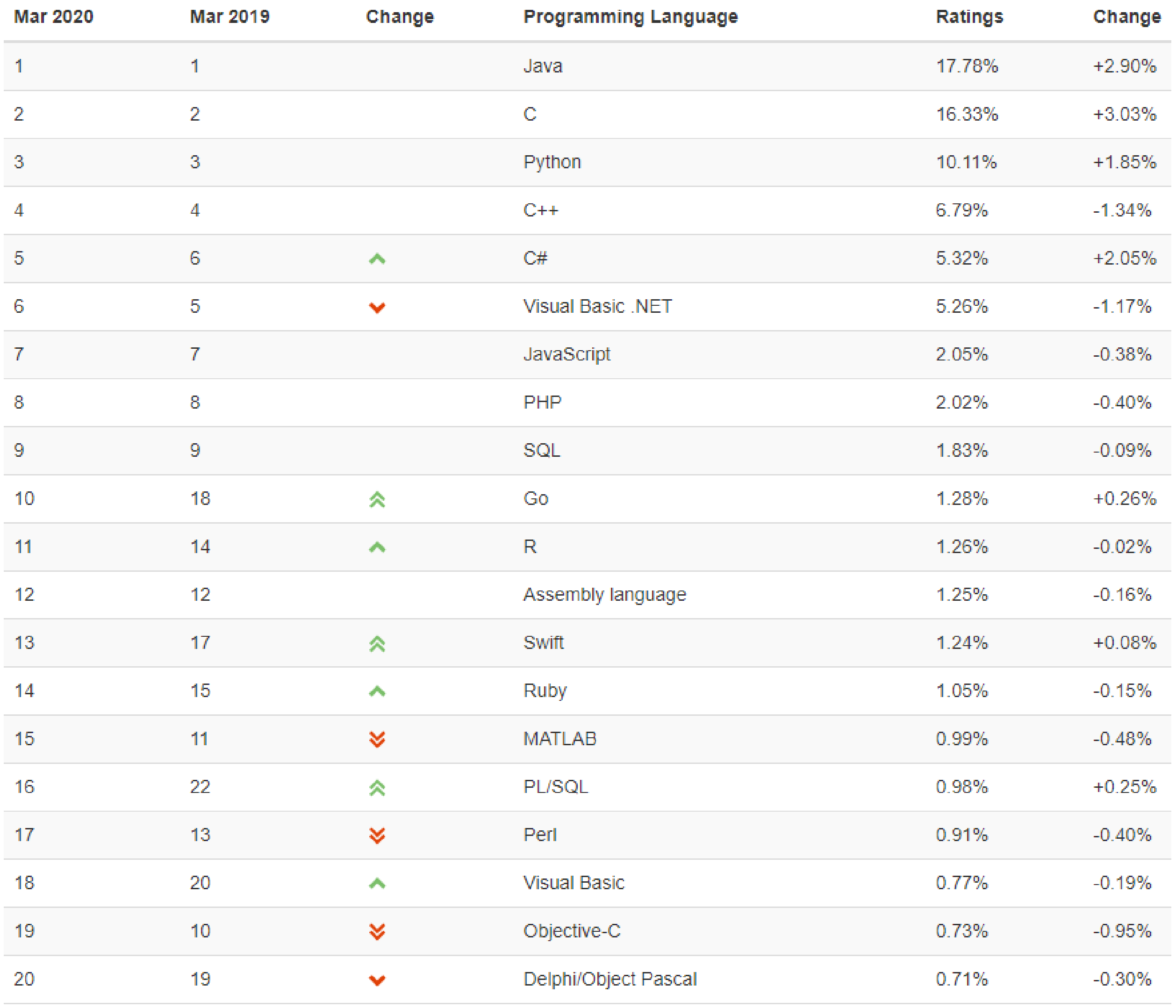
How many programming languages are out there?
https://en.wikipedia.org/wiki/List_of_programming_languages
https://en.wikipedia.org/wiki/Comparison_of_programming_languages



Language	Intended use	Imperative	Object-oriented	Functional	Procedural	Generic	Reflective	Event-driven
ActionScript 3.0	Application, client-side, web	Yes	Yes	Yes				Yes
Ada	Application, embedded, realtime, system	Yes	Yes[2]		Yes[3]	Yes[4]		
Aldor	Highly domain-specific, symbolic computing	Yes	Yes	Yes				
ALGOL 58	Application	Yes						
ALGOL 60	Application	Yes						
ALGOL 68	Application	Yes						
Ateji PX	Parallel application		Yes					
APL	Application, data processing							
Assembly language	General	Yes						
AutoHotkey	GUI automation (macros), highly domain-specific	Yes						
AutoIt	GUI automation (macros), highly domain-specific	Yes			Yes			Yes
Bash	Shell, scripting	Yes			Yes			
BASIC	Application, education	Yes			Yes			
BBj	Application, business, web		Yes		Yes			
BeanShell	Application, scripting	Yes	Yes	Yes			Yes	
BitC	System	Yes		Yes				
BLISS	System				Yes			
BlitzMax	Application, game	Yes	Yes		Yes		Yes	
Blue	Education	Yes	Yes			Yes		
Boo	Application							
Bro	domain-specific, application	Yes						Yes
C	Application, system,[11] general purpose, low-level operations	Yes			Yes			
C++	Application, system	Yes	Yes	Yes	Yes	Yes		
C#	Application, RAD, business, client-side, general, server-side, web	Yes	Yes	Yes[14]	Yes	Yes	Yes	Yes
Clarion	General, business, web	Yes	Yes	Yes[16]				
Clean	General			Yes		Yes		
Clojure	General			Yes				
CLU	General	Yes	Yes		Yes	Yes		
COBOL	Application, business	Yes	Yes		Yes			
Cobra	Application, business, general, web	Yes	Yes	Yes		Yes	Yes	
ColdFusion (CFML)	Web		Yes		Yes			
Common Lisp	General	Yes	Yes	Yes	Yes	Yes	Yes	Yes
COMAL 80	Education	Yes			Yes			
Crystal	General purpose	Yes	Yes[17]	Yes	Yes			
Cython	Application, general, numerical computing	Yes	Yes	Yes			Yes	
D	Application, system	Yes	Yes	Yes	Yes	Yes	Yes	
Dart	Application, web, server-side, mobile, IoT	Yes	Yes	Yes				
Dylan	Application		Yes	Yes				
Eiffel	General, application, business, client-side, server-side, web (EWF)	Yes	Yes	Yes[19][20]		Yes	Yes Erl-G	Yes Agents
Elixir	Application, distributed			Yes				Yes

Classification of programming languages

Do I need to learn them all?



Mar 2020	Mar 2019	Change	Programming Language	Ratings	Change
1	1		Java	17.78%	+2.90%
2	2		C	16.33%	+3.03%
3	3		Python	10.11%	+1.85%
4	4		C++	6.79%	-1.34%
5	6	↑	C#	5.32%	+2.05%
6	5	↓	Visual Basic .NET	5.26%	-1.17%
7	7		JavaScript	2.05%	-0.38%
8	8		PHP	2.02%	-0.40%
9	9		SQL	1.83%	-0.09%
10	18	⇈	Go	1.28%	+0.26%
11	14	↑	R	1.26%	-0.02%
12	12		Assembly language	1.25%	-0.16%
13	17	⇈	Swift	1.24%	+0.08%
14	15	↑	Ruby	1.05%	-0.15%
15	11	⇊	MATLAB	0.99%	-0.48%
16	22	⇈	PL/SQL	0.98%	+0.25%
17	13	⇊	Perl	0.91%	-0.40%
18	20	↑	Visual Basic	0.77%	-0.19%
19	10	⇊	Objective-C	0.73%	-0.95%
20	19	↓	Delphi/Object Pascal	0.71%	-0.30%

Very Long Term History of PL

Programming Language	2020	2015	2010	2005	2000	1995	1990	1985
Java	1	2	1	2	3	-	-	-
C	2	1	2	1	1	2	1	1
Python	3	7	6	6	22	21	-	-
C++	4	4	4	3	2	1	2	12
C#	5	5	5	8	9	-	-	-
Visual Basic .NET	6	10	-	-	-	-	-	-
JavaScript	7	8	8	9	6	-	-	-
PHP	8	6	3	4	25	-	-	-
SQL	9	-	-	97	-	-	-	-
Objective-C	10	3	20	37	-	-	-	-
Fortran	30	30	22	14	16	4	3	11
Lisp	31	19	15	13	8	5	5	2
Ada	36	29	25	15	15	6	6	3
Pascal	232	15	13	64	13	3	16	5

What is Object-Oriented Programming?

DEFINITION [Object Oriented Programming] A language or technique is object-oriented if and only if it directly supports [Stroustrup, 1995]:

[1] **Abstraction** – providing some form of classes and objects

[2] **Inheritance** – providing the ability to build new abstractions out of existing ones

[3] **Runtime polymorphism** – providing some form of runtime binding.

This definition includes all major languages commonly referred to as object-oriented: **Ada95, Beta, C++, Java, CLOS, Eiffel, Simula, Smalltalk**, and many other languages fit this definition.

Classical programming languages without classes, such as C, Fortran4, and Pascal, are excluded. Languages that lack direct support for inheritance or runtime binding, such as Ada88 and ML are also excluded.

What is an **object**?

DEFINITION A typical dictionary definition reads: **object**: a visible or tangible thing of relative stable form; a thing that may be apprehended intellectually; a thing to which thought or action is directed [*The Random House College Dictionary*, 1975]

DEFINITION [Object] Samplings from the OO literature include:

[1] An object has identity, state and behavior ([Booch, 1990]).

[2] An object is a unit of structural and behavioral modularity that has properties ([Buhr, 1998]).

[3] An object as a conceptual entity that: is identifiable, has features that span a local state space, has operations that can change the status of the system locally, while also inducing operations in peer objects. ([de Champeaux, 1993])

Very challenging to think “object-oriented”; shift from structural thinking; using classes, methods and attributes is not OO!

A Short History of Object-Oriented Programming

- Simula 67 – the first OO programming language; extension of ALGOL60
- Smalltalk – conceived by Alan Kay (Smalltalk-72, Smalltalk-80); dynamically typed; Strongtalk (1993) – Smalltalk + type system
- mid 80’s – many languages added support for OO: Objective C, C++, Object Pascal, Modula 3, Oberon, Objective CAML, CLOS.
- Eiffel – Bertrand Meyer (1988) – Pascal-like syntax, design-by-contract
- Other “exotic” OO languages: Sather, Trellis/Owl, Emerald, Beta (evolution of Simula), Self
- Java – James Gosling (1995); Java 1.5 (2004) – support for generic programming; Java 1.8 (2014) – support for closures
- Extensions to Java: Groovy,

[Bruce, 2002]

What is C++?

DEFINITION 1: C++ is a general-purpose programming language with a bias towards systems programming that supports efficient low-level computation, data abstraction, object-oriented programming, and generic programming. [Stroustrup, 1999]

DEFINITION 2: C++ is a statically-typed general-purpose language relying on classes and virtual functions to support object-oriented programming, templates to support generic programming, and providing low-level facilities to support detailed systems programming.

[Stroustrup, 1996]

C++ Design Ideas

C++ was designed to support a range of good and useful styles. Whether they were object oriented, and in which sense of the word, was either irrelevant or a minor concern [Stroustrup, 1995]:

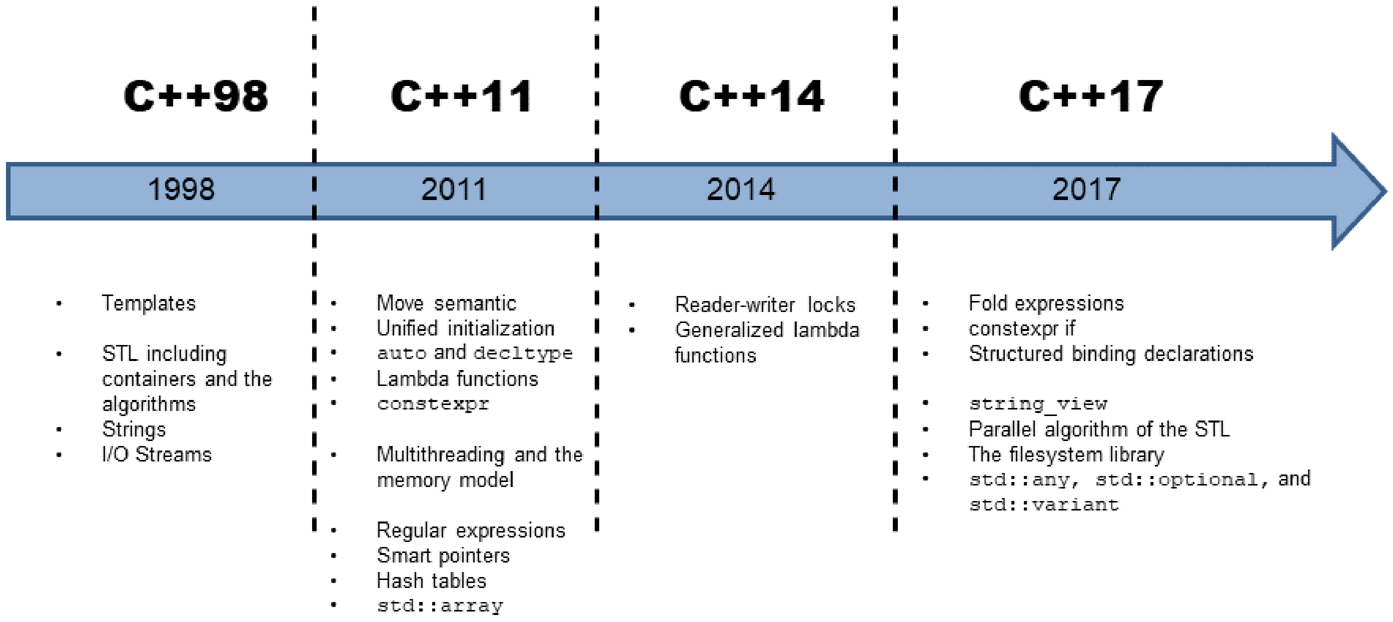
- [1] Abstraction – the ability to represent concepts directly in a program and hide incidental details behind well-defined interfaces – is the key to every flexible and comprehensible system of any significant size.
- [2] Encapsulation – the ability to provide guarantees that an abstraction is used only according to its specification – is crucial to defend abstractions against corruption.
- [3] Polymorphism – the ability to provide the same interface to objects with differing implementations – is crucial to simplify code using abstractions.
- [4] Inheritance – the ability to compose new abstractions from existing one – is one of the most powerful ways of constructing useful abstractions.
- [5] Genericity – the ability to parameterize types and functions by types and values – is essential for expressing type-safe containers and a powerful tool for expressing general algorithms.
- [6] Coexistence with other languages and systems – essential for functioning in real-world execution environments.
- [7] Runtime compactness and speed – essential for classical systems programming.
- [8] Static type safety – an integral property of languages of the family to which C++ belongs and valuable both for guaranteeing properties of a design and for providing runtime and space efficiency.

A Brief History of C++: Early “life”

- *1979 – 1983 C with Classes*: Bjarne Stroustrup (AT&T Bell Labs) ports concepts (e.g. classes, inheritance) from Simula67 to C
- *1982 – 1985 From C with Classes to C++:* the first commercial release and the printing of the book (The C++ Programming Language) that defined C++ was released in October 1985
- *1985 – 1991 Release 2.0 (1989)* Evolutions from the first release and 2nd Edition of the book in 1991
- *1987 – 2000*: *The Explosion in Interest and Use:* growth of C++ tools and library industry.
- *1994* : Standard Template Library

A Brief History of C++: Standardization

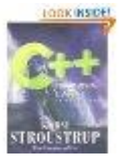
- *1998* – C++98 (ISO/IEC 14882:1998) – 1st international standard for C++; it includes STL
- *2003* – C++03 (ISO/IEC 14882:2003)
- *2007* – C++TR1 (ISO/IEC TR 19768)
- *2011* – C++11 (ISO/IEC 1488)
- *2014 December* – C++14 (ISO/IEC 1488:2014)
- *2017 December* – C++17 (ISO/IEC 14882:2017)



Further Reading

1. [Bruce, 2002] Kim B. Bruce - Foundations of Object-Oriented Languages, MIT Press, 2002 [Section 1.1, Page 4-7]

2. [Stroustrup, 1999] Bjarne Stroustrup - An Overview of the C++ Programming Language in “The Handbook of Object Technology” (Editor: Saba Zamir). CRC Press LLC, Boca Raton. 1999. ISBN 0-8493-3135-8 [Section 4.2, Page 10-11]

3. 

[Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Section 4.9.4]

4. [Goguen, 1978] J.A. Goguen, J.W. Thatcher, and E.G. Wagner - An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4. Prentice-Hall, 1978

5. [Guttag, 1977] J.V. Guttag - Abstract data types and the development of data structures. *Communications of ACM*, 20(6):396–404, 1977

Unit 2

Agenda

1. Basic OOP concepts
    1. Class
    2. Object
    3. Message

2. Scopes

3. Classes
    1. Declaration
    2. Implementation
    3. Accessing class members
    4. Access control

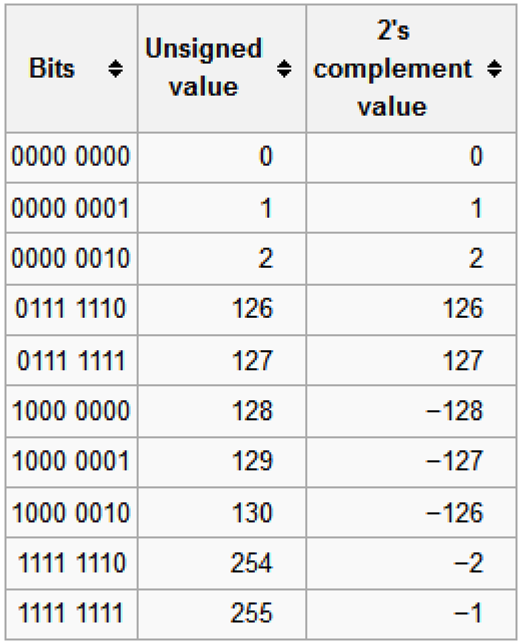
Data type

DEFINITION [Data type] Data type =

- **possible values of the type**
- **the way values are stored (internal representation in memory)**
- **operations applicable**

Examples:

- predefined (built-in): int, float, double, char, void, pointer etc.
- user-defined: Employee, complex etc.



Bits	Unsigned value	2's complement value
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	−128
1000 0001	129	−127
1000 0010	130	−126
1111 1110	254	−2
1111 1111	255	−1

What is wrong with user-defined types in C?

- User-defined data types (struct) are **not complete** (operations cannot be defined similarly to built-in types)

- Internal representation is **visible and accessible** to everyone

complex.h

```
struct complex {
     double re, im;
};
```

user.c

```
void f(complex c) {
     double d1 = c.re, d2 = c.im;
}
```

complex.h

```
struct complex {
     double v[2];
     // v[0] – real part,
     // v[1] – imaginary part
};
```

user.c

```
void f(complex c) {
    // Error
    double d1 = c.re, d2 = c.im;
}
```

Abstraction

- Abstraction:

  (real-life/human) Problem => (computer/machine) Model

- Model
  - Data
  - Operations

- **DEFINITION [Abstraction] Abstraction means structuring a real-life problem into well-defined entities by defining their <u>data</u> and <u>operations</u>.**

- Remark: It’s a general process with no direct link with a programming language.

- We get Abstract Data Types (models)

Abstract Data Types

- Introduced by Goguen 1978 and Guttag 1977 [Goguen 1978, Guttag 1977]
- **DEFINITION [Abstract Data Type] An ADT is characterized by the following properties:**
  - **It exports a type;**
  - **It exports a set of operations (the <u>interface</u>) that are the one and only access mechanism to the type's data structure (Encapsulation)**
  - **Axioms and preconditions define the application domain of the type.**

- Abstract Data Type = class of objects whose logical behavior is defined by a set of values and a set of operations

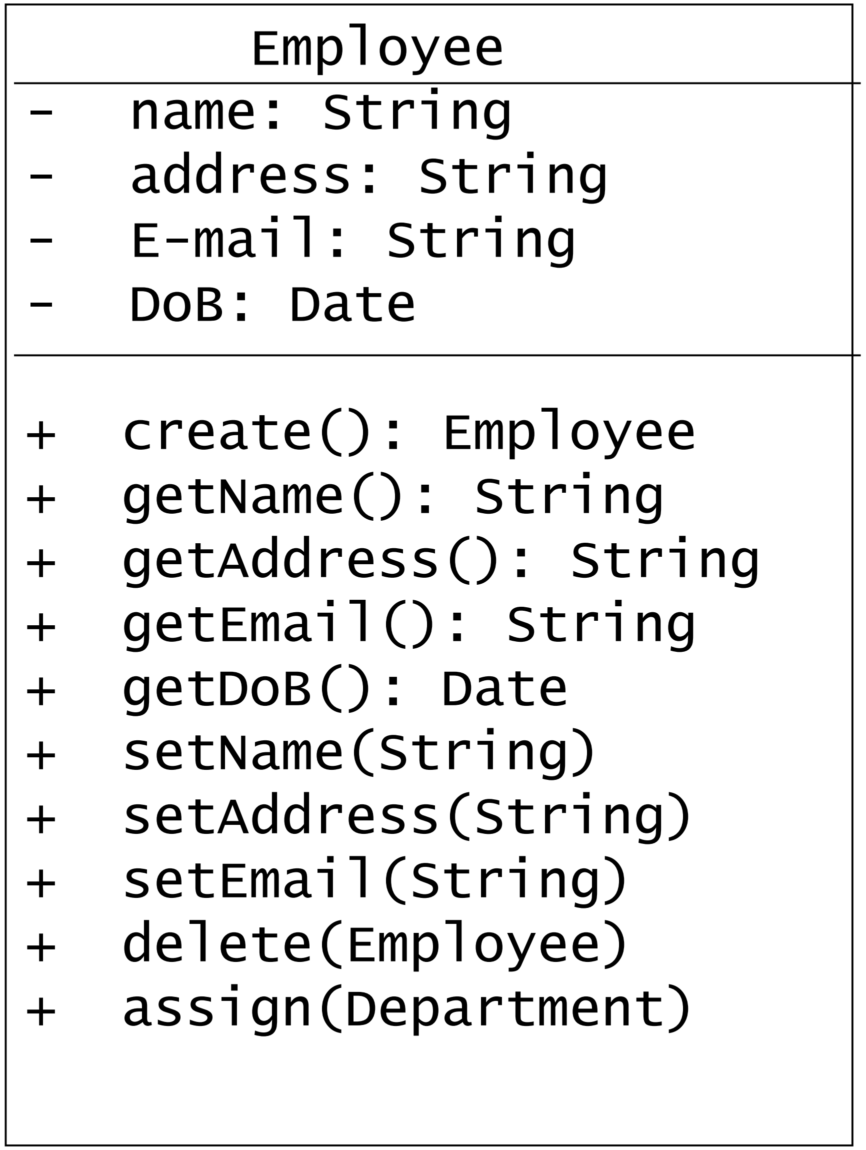
- How to represent an abstraction:
  - using mechanism provided by a programming language (e.g. class in OOP)
  - using graphical representations offered by a modeling language (e.g. rectangle in UML)
  - using textual/non-standard description

Abstraction – an example

- Modeling employees of an institution
- Candidate properties of an employee
  - name, address, date of birth
  - hair color, musical preferences etc.

- Employee (Textual representation)
  - Data
    - Name
    - Address
    - Date of Birth
    - E-mail
  - Operations
    - CRUD: Create/Read/Update/Delete
    - Assign to a particular department
    - Compute salary

An example: Employee (UML representation)



- means private
+ means public

An example: Employee (C++ representation)

Employee.h

```
class Employee  {
     String name;
     String address;
     String email;
     Date DoB;
     Department department;

public:
     Employee create();
     static void delete(Employee& e);
     void assign(Department& d);

     String getName();
     String getAddress();
     String getEmail();
     Date getDoB();

     void setName(String&);
     void setAddress(String&);
     void setEmail(String&);
};
```

Employee.cpp

```
Employee Employee::create() {
   name = “”;
   address = “”;
   email=“”;
   DoB.init(“1/1/1900”);
}

Employee Employee::setName(String& n) {
    name = n;
}

void Employee::delete(Employee& e) {
    // list.remove(e);
    department.remove(e);
}

void Employee::assign(Department& d) {
    d.add(this);
    department = d;
}

String Employee::getName() {
    return name;
}
```

Basic object-oriented concepts: Class

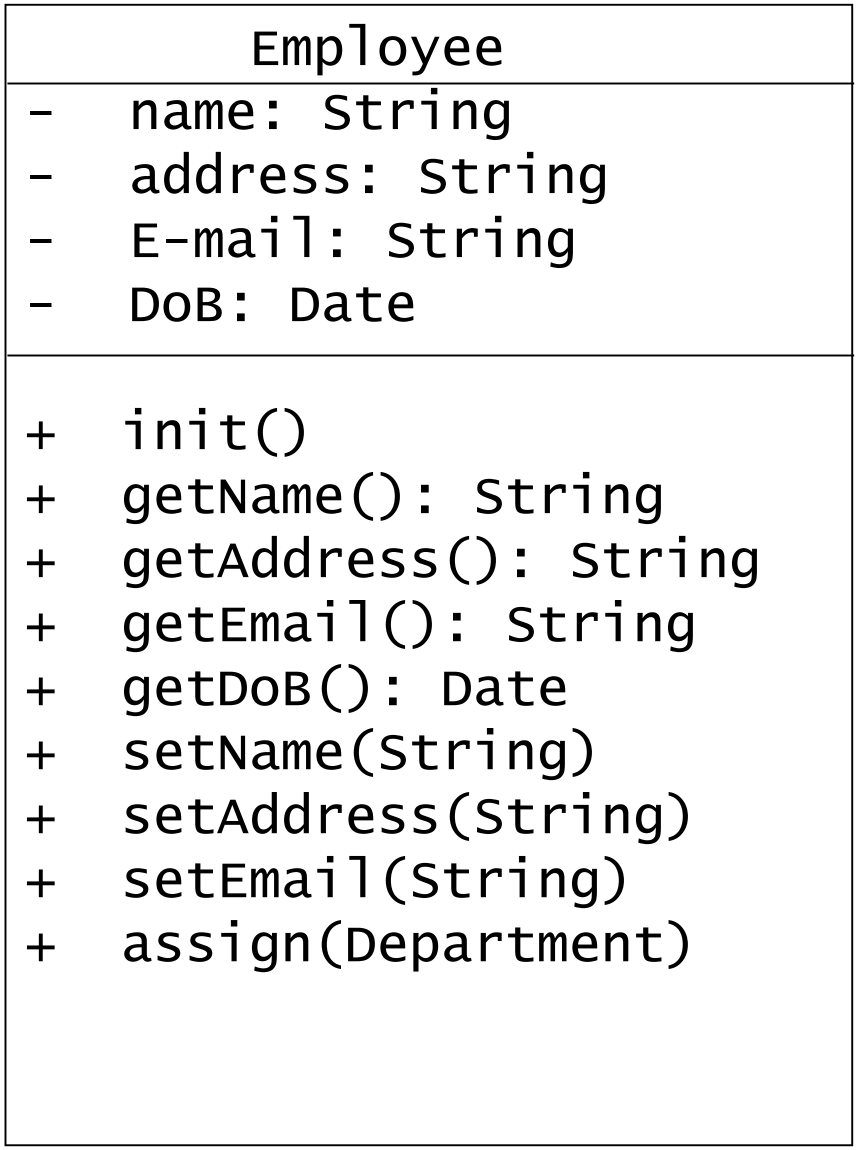
DEFINITION [Class] A class is the implementation of a data type (concrete, abstract or generic). It defines attributes and functions which implement the data structure and operations of the data type, respectively.

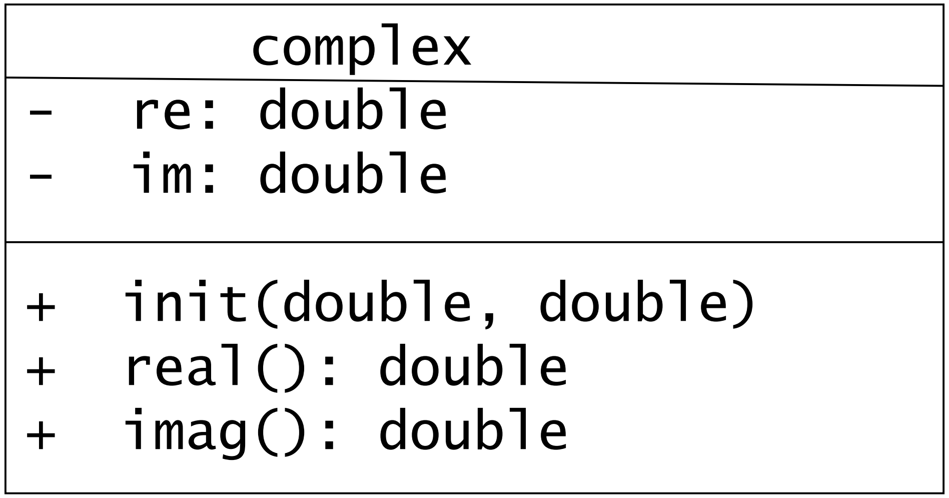
Remarks:

1. Functions are also called methods. In [Stroustrup, 1997] a clear distinction is made between functions and methods. A method is a virtual function. This terminology will be followed throughout this course as well.
2. Classes define properties and behaviour of sets of objects.

Examples: Person, Employee, Window, House, complex, Date etc.

Basic object-oriented concepts: Class (cont.)





- private
+ public

Basic object-oriented concepts: Class (cont.)

- Classes are defined in C++ using **struct** or **class** keywords.

```
struct Date {
     int day, month, year;

    // Date initialization
    void init(int day, int month, int year);
};
```

```
class Employee {
     String name;
     String address;
     String email;
     Date dob;

public:
     void init();
     void assign(Department& d);

     String getName();
     String getAddress();
     String getEmail();
     Date getDoB();

     void setName(String);
     void setAddress(String);
     void setEmail(String);
};
```

Syntax

```
class X {
     // member variables
     // member functions
};

struct X {
     // member variables
     // member functions
};
```

Remark: In case of using struct, by default, all members are public. For class, they are private by default. The programmer is free to choose which keyword he/she prefers, with a *recommendation* to use struct for data types that doesn't provide behaviour.

EXERCISE: Write the declaration for class complex.

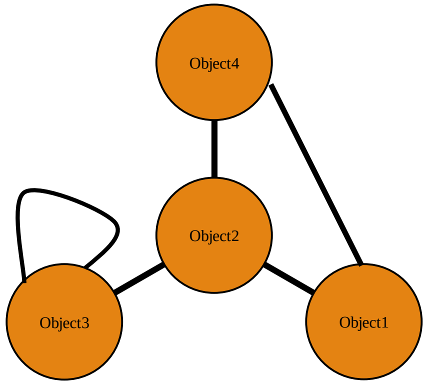
Basic object-oriented concepts: Object

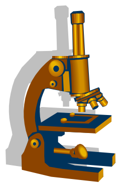
- DEFINITION [Object] An object is an instance of a class. It can be uniquely identified by its name, it defines a state which is represented by the values of its attributes at a particular time and it exposes a behaviour defined by the set of functions (methods) that can be applied to it.

- An OO Program = collection of objects that interact one with another.

- Example:

```
Employee e1;

complex a, b, c;

Employee team[10];
```

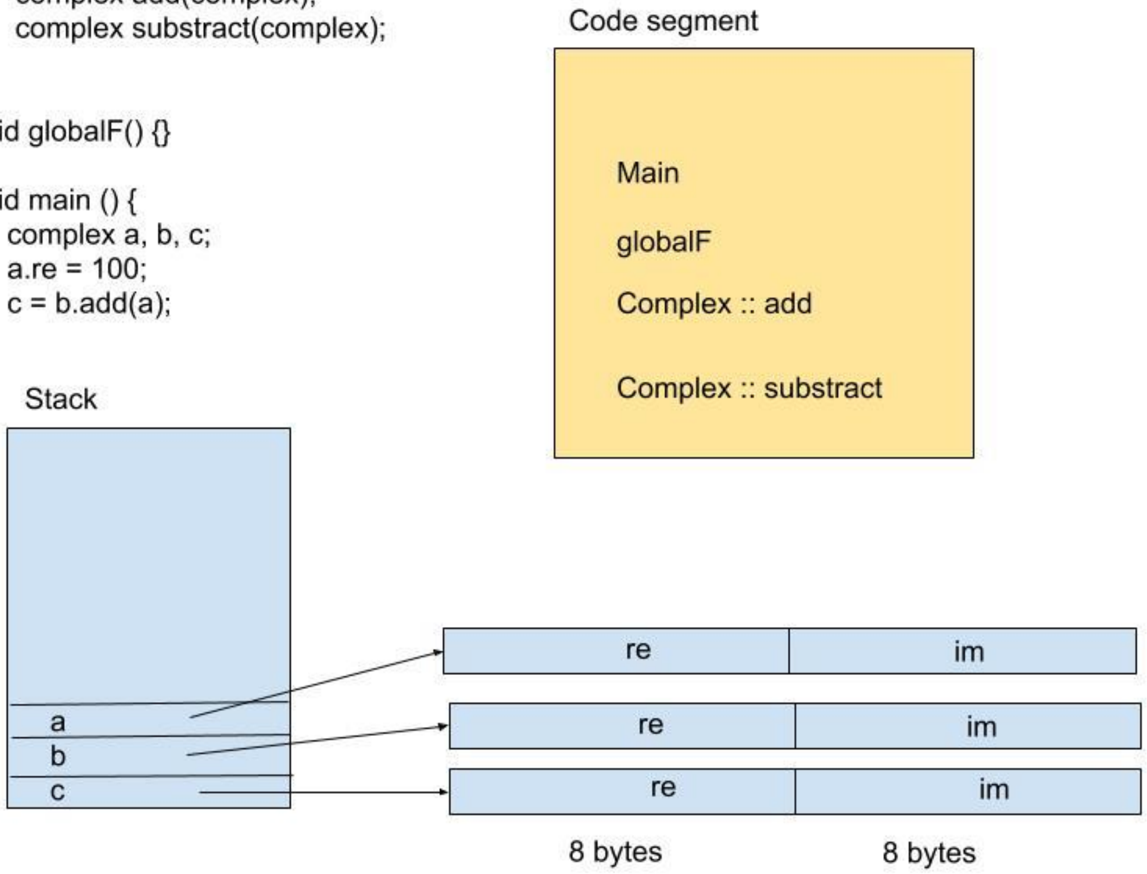




How are objects internally represented in C++?

Objects Representation in Memory

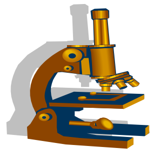
```
class complex {
      double re, im;
   public:
      complex add(complex);
      complex substract(complex);
}

void globalF() {}

void main () {
     complex a, b, c;
     a.re = 100;
     c = b.add(a);
}
```



Basic object-oriented concepts: Message

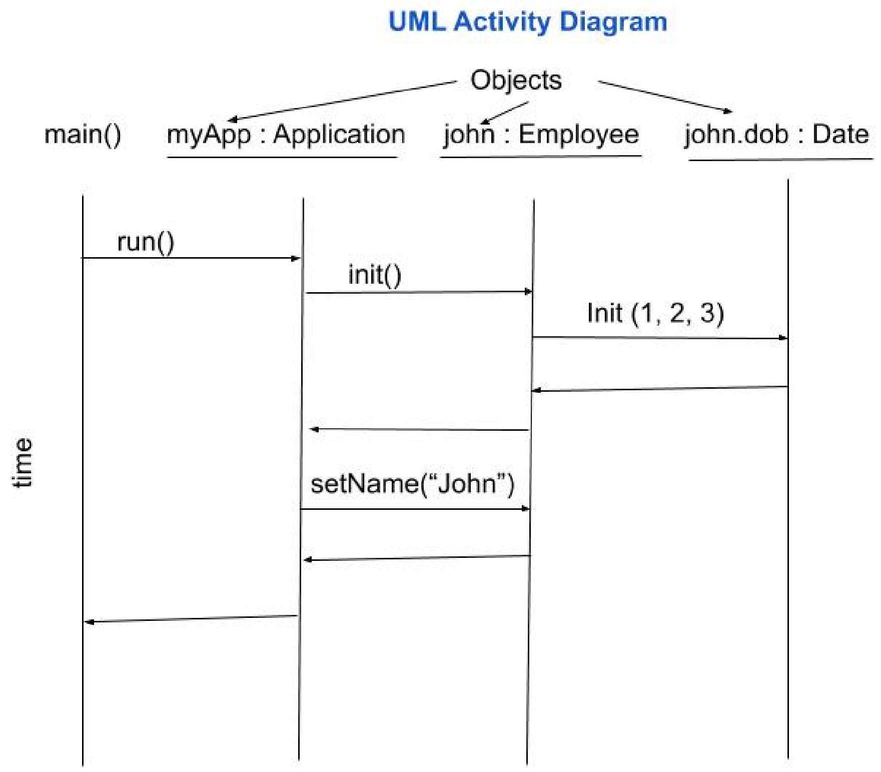
- Q: How does objects interact?
- A: By sending messages from one object to another asking the recipient to apply an operation on itself.
- DEFINITION [Message] A message is a request to an object to invoke one of its functions. A message contains the name of the function and the arguments of the function.
- Sending a message is achieved in C++ by '.' or '->' operators.

```
Employee Employee::init() {
    dob.init(1, 1, 1980);
}

void Application::run() {
    Employee john;
    john.init();
    john.setName("John");
}

int main() {
     Application myApp;
     myApp.run();
}
```



How objects interact in C++?

Communication between Objects



```
Employee Employee::init() {
    dob.init(1, 1, 1980);
}

void Application::run() {
    Employee john;
    john.init();
    john.setName(“John”);
}

int main() {
     Application myApp;
     myApp.run();
}
```

Scopes

DEFINITION [Scope] A scope is a specific part of the source code.

- Example: any block (delimited by { and }) defines a scope; each function defines a scope.
- Each class defines a new scope. The name of the scope is the same as class name.
- A name is defined in the global scope if it is defined outside any function, class or namespace. It is visible from the point of declaration down to the end of the file.
- Every declaration introduces a name in the scope => this name is visible only in that scope and hides names declared in enclosing scopes or at global scope.

Scope resolution operator

DEFINITION [Scope resolution operator]

The scope resolution operator (::) allows to access an identifier (name) defined in another namespace than the current one.

Syntax

```
ScopeName :: Identifier
```

```
int x; // global x

void f() {
     int x; // local x hides the global x
     x = 1; // local x = 1
     ::x = 2; // accessing the global x
}

int* px = &x; // address of global x
```

<== Hiding x

... and what is going on here? ==>

```
int x = 0;

void f() {
     int y = x;
     int x = 1;
     y = x;
}
```

[Stroustrup, 1997 - Section 4.9.4]

Class declaration

- Classes are defined in C++ using **struct** or **class** keywords.

```
struct Date {
     int day, month, year;

    // Date initialization
    void init(int day, int month, int year);
};
```

```
class Employee {
     String name;
     String address;
     String email;
     Date DoB;

public:
     void init();
     void assign(Department& d);

     String getName();
     String getAddress();
     String getEmail();
     Date getDoB();

     void setName(String);
     void setAddress(String);
     void setEmail(String);
};
```

Syntax

```
class X {
     // member variables
     // member functions
};

struct X {
     // member variables
     // member functions
};
```

Remark: In case of using struct, by default, all members are public. For class, they are private by default. The programmer is free to choose which keyword he/she prefers, with a *recommendation* to use struct for data types that doesn't provide behaviour.

EXERCISE: Write the declaration for class complex.

Implementing member functions

- In C++, each member function is handled as any other ordinary C function.
- A member function is a name introduced in the scope defined by its parent class. Therefore, to access this name outside its scope of definition, we need to **fully qualify** the name using the scope resolution operator and the class name (namespace).

This indicates that **init** is the identifier declared inside Date class/namespace and not defined at global scope!

```
void Date:: init(int _day, int _month, int _year) {
     day = _day;
     month = _month;
     year = _year;
}
```

Accessing class members

- Access to members of a class – the same syntax as for struct in C, i.e. using '.' or '->' operators.
- One need to have an object through which members are "*incarnated*" and accessed (of course, there are some exceptions… but we're saving them for later).

Syntax

```
object.member

pointer_to_object->member
```

```
void f() {
    Date today;
    Date *pdate = &today;

    today.init(4, 3, 2013); // March 7th, 2007

    printf("Today is %d/%d/%d.",
              pdate->day,
              pdate->month,
              today.year);
}
```

```
void f() {
    day = 4; // ERROR
    // day is not in Date scope
}

void Date::init(int _day,
          int _month,
          int _year) {
     day = _day; // OK
     month = _month; // OK
     year = _year; // OK
}
```

Accessing class members (cont.)

New operators in C++: ‘.*’ and ‘->*’

```
// pointer to int data member of Date
typedef int Date::*PM;

// pointer to member function of Date taking 3 arguments of type int
typedef void (Date::*PMF)(int, int, int);

void f() {
    Date today;
    Date *pdate = &today;

    PM pm = &Date::day;
    PMF pmf = &Date::init;

    (today.*pmf)(7, 3, 2007); // ⇔ today.init(7, 3, 2007);
    today.*pm = 8; // ⇔ today.day = 8
    pm = &Date::month;
    today.*pm = 8; // ⇔ today.month = 8

     (pdate->*pmf) (7, 3, 2007); // ⇔ pdate->init(7, 3, 2007);
     pdate->*pm = 8; // ⇔ pdate->month = 8
}
```

Access Control

- To achieve encapsulation in C++, various level of access are defined for class members (data or functions):
  - **private** - accessible only to member functions and friends;
  - **protected** - accessible to member functions and friends + member functions and friends of derived classes;
  - **public** - accessible from everywhere.

- Public members define the **public interface** of a class.

```
class MyClass {
     int x;    // private, by default
public:
     int y, z; // public
     int f();  // public member function
private:
     int w;    // private data
protected:
     int f(int, int); // protected member function
};
```

```
void main() {
     MyClass obj;

     // ERROR: private member
     obj.x = 10;

     // OK!
     obj.f();

     // ERROR:protected members
     obj.f(1,1);
}
```

Encapsulation

- **DEFINITION [Encapsulation] The principle of hiding the data structures and to provide a well-defined, public interface to access them.**
- Example: representation of complex numbers

complex.h

```
struct complex {
private: // hide internal representation
    double v[2];

public: // public interface to access data
    double real();
    double imaginary();
};
```

user.c

```
void f(complex c) {
    double d1 = c.real(); // fixed, regardless internal representation
    double d2 = c.imaginary(); // fixed, regardless internal representation
}
```

Further Reading

1. [Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Section 4.9.4, Chapter 10]

2. [Stroustrup, 2013] Bjarne Stroustrup – The C++ Programming Language 4th Edition, Addison Wesley, 2013 [Chapter 16]