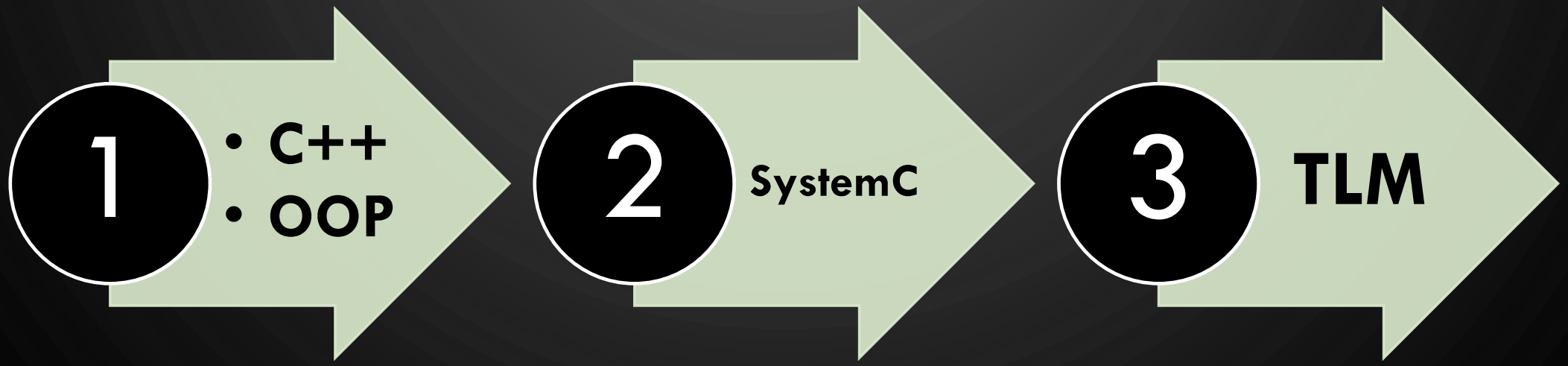

System C
Anis Hassen

MAIN POINTS



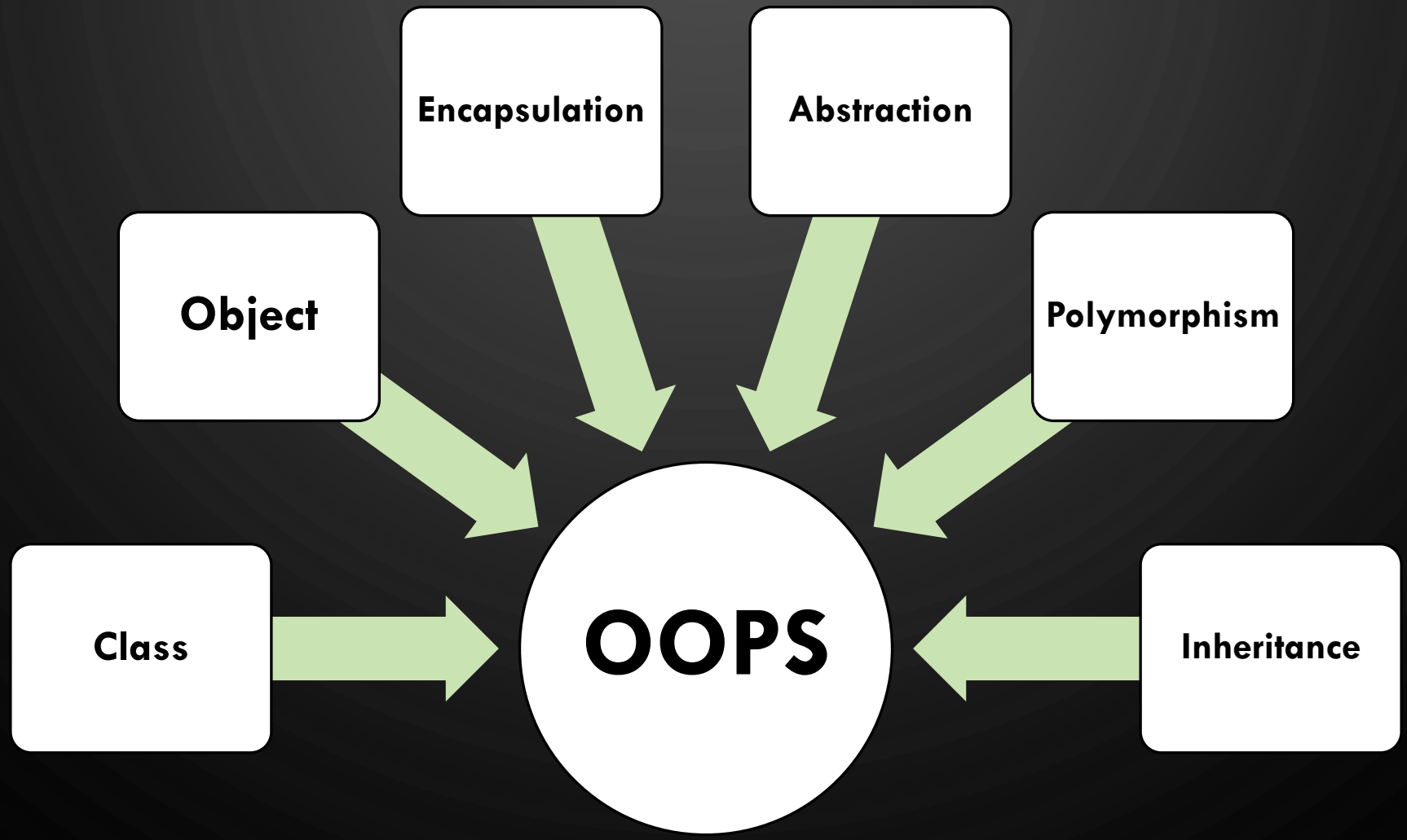
DIFFERENCE BETWEEN C AND C++

C	C++
• C Supports Procedural Program	• C++ is known as hybrid language, because it support both procedural and OOP
• As C doesn't support the OOPS concept , so it has no support for polymorphism, encapsulation and inheritance	• C++ has support for polymorphism, encapsulation and inheritance as it is an OOPS language
• C is a subset of C++	• C++ is a subset of C
• C contains 32 Keywords	• C++ contains 52 keywords (Public, Private, Protected, try, catch, throw)
• C is a function driven language	• C++ is an object driven language
• Function and Operator overloading is not support in c	• C++ supports function & operator overloading
• C doesn't support exception handling	• C++ supports exception handling using try and catch

OOPS IN C++

The Main aim of OOP , is to bind together the data and functions the operate on them . So that, no other part of the code can access this data except this Function

TYPES OF OOPS



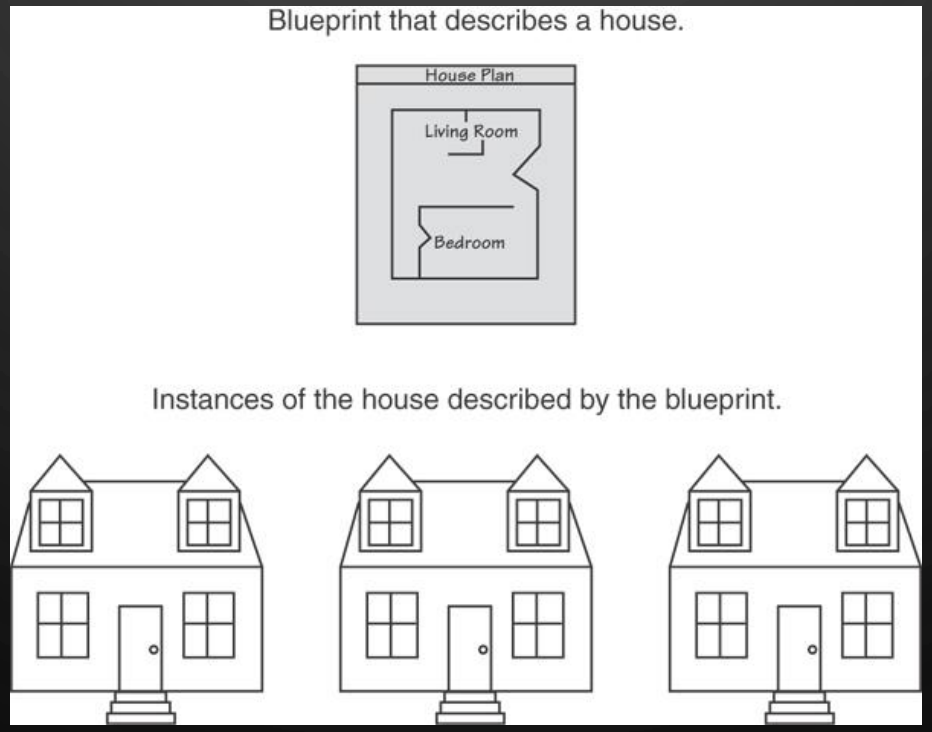
OBJECT-ORIENTED PROGRAMMING TERMINOLOGY

- class: like a struct (allows bundling of related variables), but variables and functions in the class can have different properties than in a struct

- object: an instance of a class, in the same way that a variable can be an instance of a struct

CLASSES AND OBJECTS

- A Class is like a blueprint and objects are like houses built from the blueprint



OBJECT-ORIENTED PROGRAMMING TERMINOLOGY

- attributes: members of a class
- methods or behaviors: member functions of a class

MORE OBJECT TERMS

- data hiding: restricting access to certain members of an object

- public interface: members of an object that are available outside of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides some protection from data corruption

CREATING A CLASS

- Objects are created from a `class`
- Format:

```
class ClassName
{
        declaration;
        declaration;
};
```

CLASSIC CLASS EXAMPLE

```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

ACCESS SPECIFIERS

- Used to control access to members of the class
- `public:` can be accessed by functions outside of the class
- `private:` can only be called by or accessed by functions that are members of the class
- In the example on the next slide, note that the functions are prototypes only (so far)

CLASS EXAMPLE

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```

ACCESS SPECIFIERS

```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

Private Members

Public Members

ACCESS SPECIFIERS (CONTINUED)

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is `private`

USING CONST WITH MEMBER FUNCTIONS

- `const` appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.

```
double getWidth() const;
double getLength() const;
double getArea() const;
```

DEFINING A MEMBER FUNCTION

- When defining a member function:
  - Put prototype in class declaration
  - Define function using class name and scope resolution operator (::)

```
int Rectangle::setWidth(double w)
{
     width = w;
}
```

GLOBAL FUNCTIONS

- Functions that are not part of a class, that is, do not have the `Class::name` notation, are global. This is what we have done up to this point.

ACCESSORS AND MUTATORS

- Mutator: a member function that stores a value in a private member variable, or changes its value in some way

- Accessor: function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.

DEFINING AN INSTANCE OF A CLASS

- An object is an instance of a class
- Defined like structure variables:

```
Rectangle r;
```

- Access members using dot operator:

```
r.setWidth(5.2);
cout << r.getWidth();
```

- Compiler error if you attempt to access a `private` member using dot operator

DERIVED ATTRIBUTES

- Some data must be stored as an attribute.
- Other data should be computed. If we stored “area” as a field, its value would have to change whenever we changed length or width.
- In a class about a “person,” store birth date and compute age

POINTERS TO OBJECTS

- Can define a pointer to an object:

```
Rectangle *rPtr;
```

- Can access public members via pointer:

```
rPtr = &otherRectangle;
rPtr->setLength(12.5);
cout << rPtr->getLength() << endl;
```

DYNAMICALLY ALLOCATING OBJECTS

```
Rectangle *r1;

r1 = new Rectangle();
```

- This allocates a rectangle and returns a pointer to it. Then:

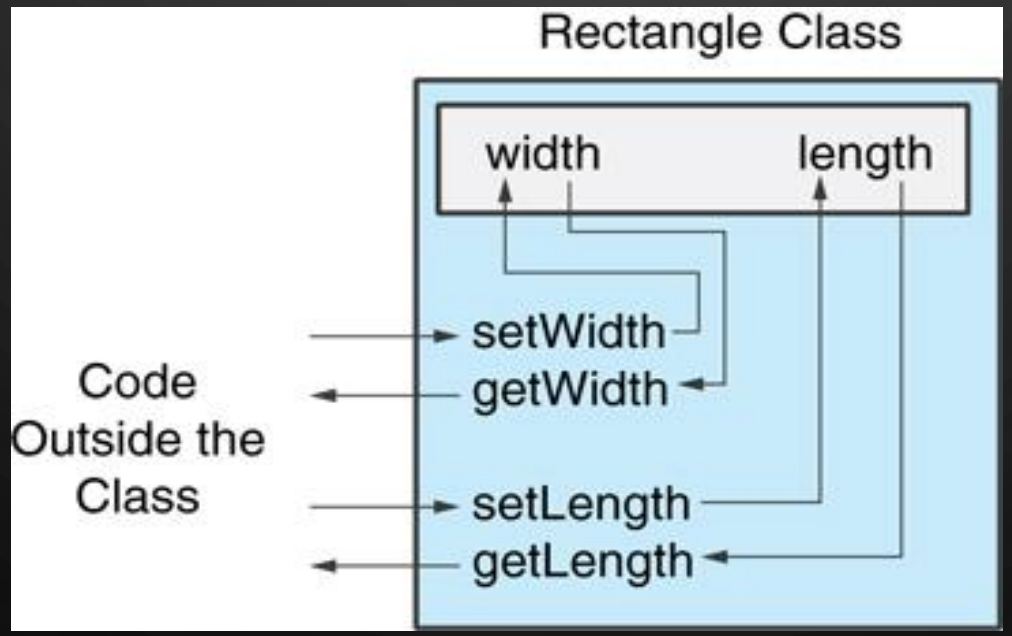
```
r1->setWidth(12.4);
```

PRIVATE MEMBERS

- Making data members `private` provides data protection
- Data can be accessed only through `public` functions
- Public functions define the class's public interface

PRIVATE MEMBERS

Code outside the class must use the class's public member functions to interact with the object.



SEPARATING SPECIFICATION FROM IMPLEMENTATION

- Place class declaration in a header file that serves as the class specification file. Name the file *ClassName*.h, for example, Rectangle.h
- Place member function definitions in *ClassName*.cpp, for example, Rectangle.cpp File should #include the class specification file
- Programs that use the class must #include the class specification file, and be compiled and linked with the member function definitions

INLINE MEMBER FUNCTIONS

- Member functions can be defined
  - inline: in class declaration
  - after the class declaration

- Inline appropriate for short function bodies:

```
int getWidth() const
   { return width; }
```

TRADEOFFS – INLINE VS. REGULAR MEMBER FUNCTIONS

- Regular functions – when called, compiler stores return address of call, allocates memory for local variables, etc.

- Code for an inline function is copied into program in place of call – larger executable program, but no function call overhead, hence faster execution

CONSTRUCTORS

- Member function that is automatically called when an object is created
- Purpose is to construct an object and do initialization if necessary
- Constructor function name is class name
- Has no return type *specified*
- (What is the real return type?)

DEFAULT CONSTRUCTORS

- A default constructor is a constructor that takes no arguments.
- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.
- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```

PASSING ARGUMENTS TO CONSTRUCTORS

- To create a constructor that takes arguments:
  - indicate parameters in prototype:

```
Rectangle(double, double);
```

  - Use parameters in the definition:

```
Rectangle::Rectangle(double w, double len)
{
    width = w;
    length = len;
}
```

PASSING ARGUMENTS TO CONSTRUCTORS

- You can pass arguments to the constructor when you create an object:

```
Rectangle r(10, 5);
```

MORE ABOUT DEFAULT CONSTRUCTORS

- If all of a constructor's parameters have default arguments, then it is a default constructor. For example:

```
Rectangle(double = 0, double = 0);
```

- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
```

CLASSES WITH NO DEFAULT CONSTRUCTOR

- When all of a class's constructors require arguments, then the class has NO default constructor
- When this is the case, you must pass the required arguments to the constructor when creating an object

DESTRUCTORS

- Member function automatically called when an object is destroyed
- Destructor name is ~classname, e.g., `~Rectangle`
- Has no return type; takes no arguments
- Only one destructor per class, *i*.e., it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it

CONSTRUCTORS, DESTRUCTORS, AND DYNAMICALLY ALLOCATED OBJECTS

- When an object is dynamically allocated with the new operator, its constructor executes:

```
Rectangle *r = new Rectangle(10, 20);
```

- When the object is destroyed, its destructor executes:

```
delete r;
```

OVERLOADING CONSTRUCTORS

- A class can have more than one constructor
  Overloaded constructors in a class must have different parameter lists:

```
Rectangle();
Rectangle(double);
Rectangle(double, double);
```

ONLY ONE DEFAULT CONSTRUCTOR AND ONE DESTRUCTOR

- Do not provide more than one default constructor for a class: one that takes no arguments and one that has default arguments for all parameters

```
Square();
Square(int = 0);  // will not compile
```

- Since a destructor takes no arguments, there can only be one destructor for a class

MEMBER FUNCTION OVERLOADING

- Non-constructor member functions can also be overloaded:

```
void setCost(double);
void setCost(char *);
```

- Must have unique parameter lists as for constructors

USING PRIVATE MEMBER FUNCTIONS

- A `private` member function can only be called by another member function
- It is used for internal processing by the class, not for use outside of the class
- If you wrote a class that had a public sort function and needed a function to swap two elements, you'd make that private

ARRAYS OF OBJECTS

- Objects can be the elements of an array:

```
Rectangle rooms[8];
```

- Default constructor for object is used when array is defined

ARRAYS OF OBJECTS

Must use initializer list to invoke constructor that takes arguments:

```
Rectangle rArray[3]={Rectangle(2.1,3.2),
Rectangle(4.1, 9.9),
Rectangle(11.2, 31.4)};
```

ARRAYS OF OBJECTS

- It isn't necessary to call the same constructor for each object in an array:

```
Rectangle rArray[3]={Rectangle(2.1,3.2),
Rectangle(),
Rectangle(11.2, 31.4)};
```

ACCESSING OBJECTS IN AN ARRAY

- Objects in an array are referenced using subscripts
- Member functions are referenced using dot notation:

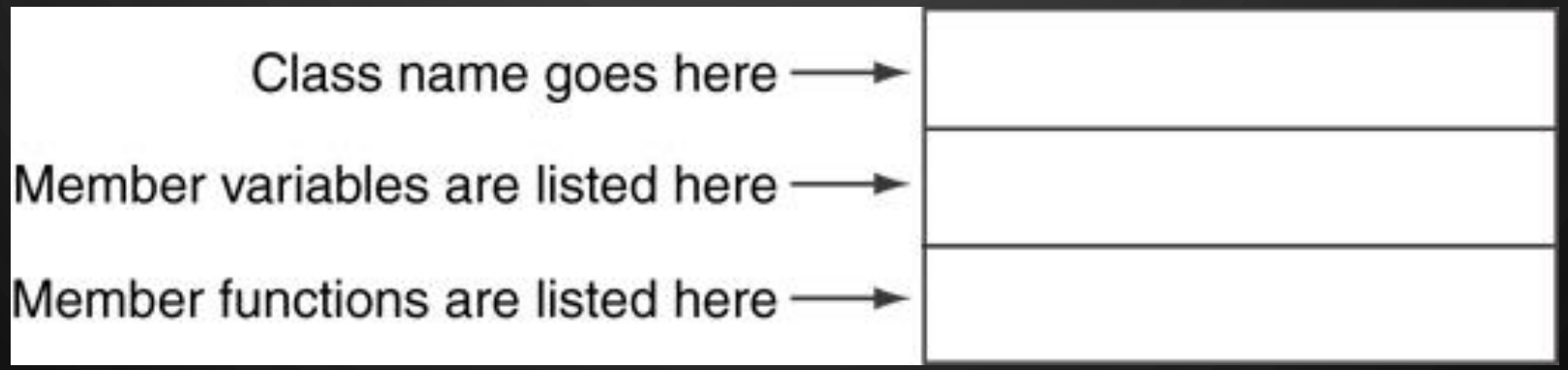
```
rArray[1].setWidth(11.3);
cout << rArray[1].getArea();
```

THE UNIFIED MODELING LANGUAGE

- *UML* stands for *Unified Modeling Language*.
- The UML provides a set of standard diagrams for graphically depicting object-oriented systems

UML CLASS DIAGRAM

- A UML diagram for a class has three main sections.



EXAMPLE: A RECTANGLE CLASS

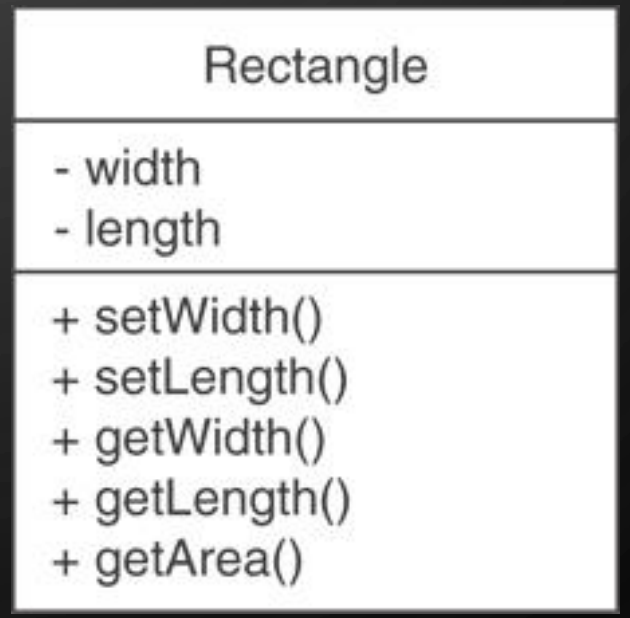

```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      bool setWidth(double);
      bool setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

UML ACCESS SPECIFICATION NOTATION

- In UML you indicate a private member with a minus (-) and a public member with a plus(+).

These member variables are private.

These member functions are public.



UML DATA TYPE NOTATION

- To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable.

```
- width : double
- length : double
```

UML PARAMETER TYPE NOTATION

- To indicate the data type of a function's parameter variable, place a colon followed by the name of the data type after the name of the variable.

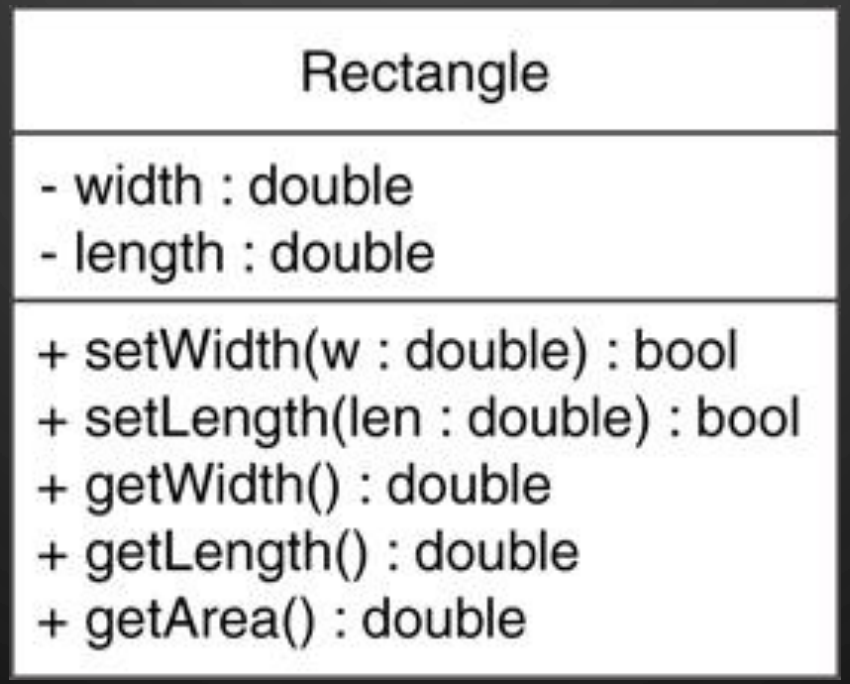
```
+setWidth(w : double)
```

UML FUNCTION RETURN TYPE NOTATION

- To indicate the data type of a function's return value, place a colon followed by the name of the data type after the function's parameter list.

```
+ setWidth(w : double) : void
```

THE RECTANGLE CLASS

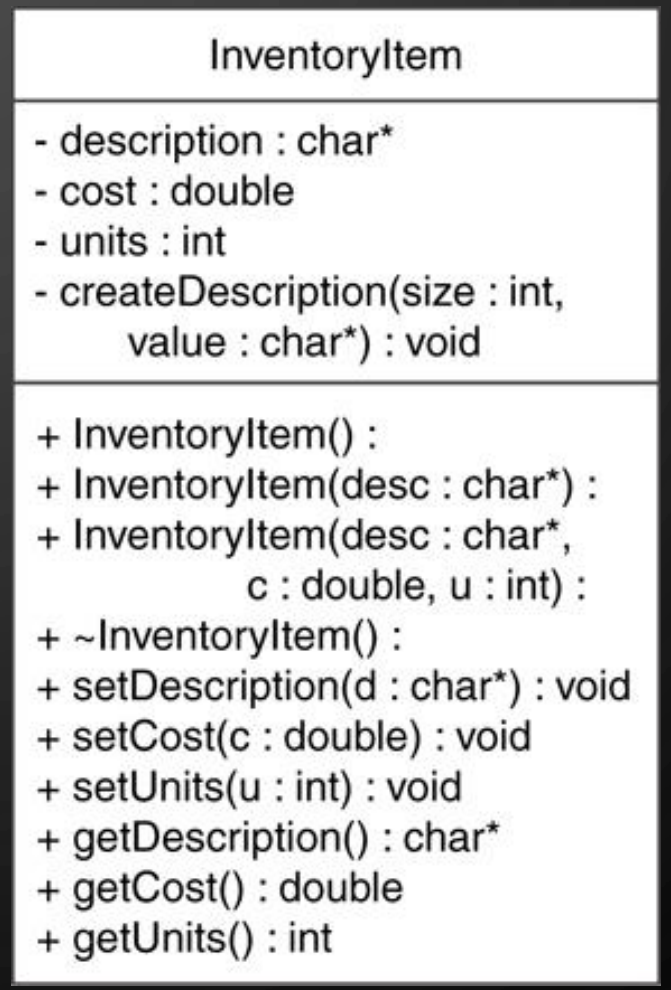


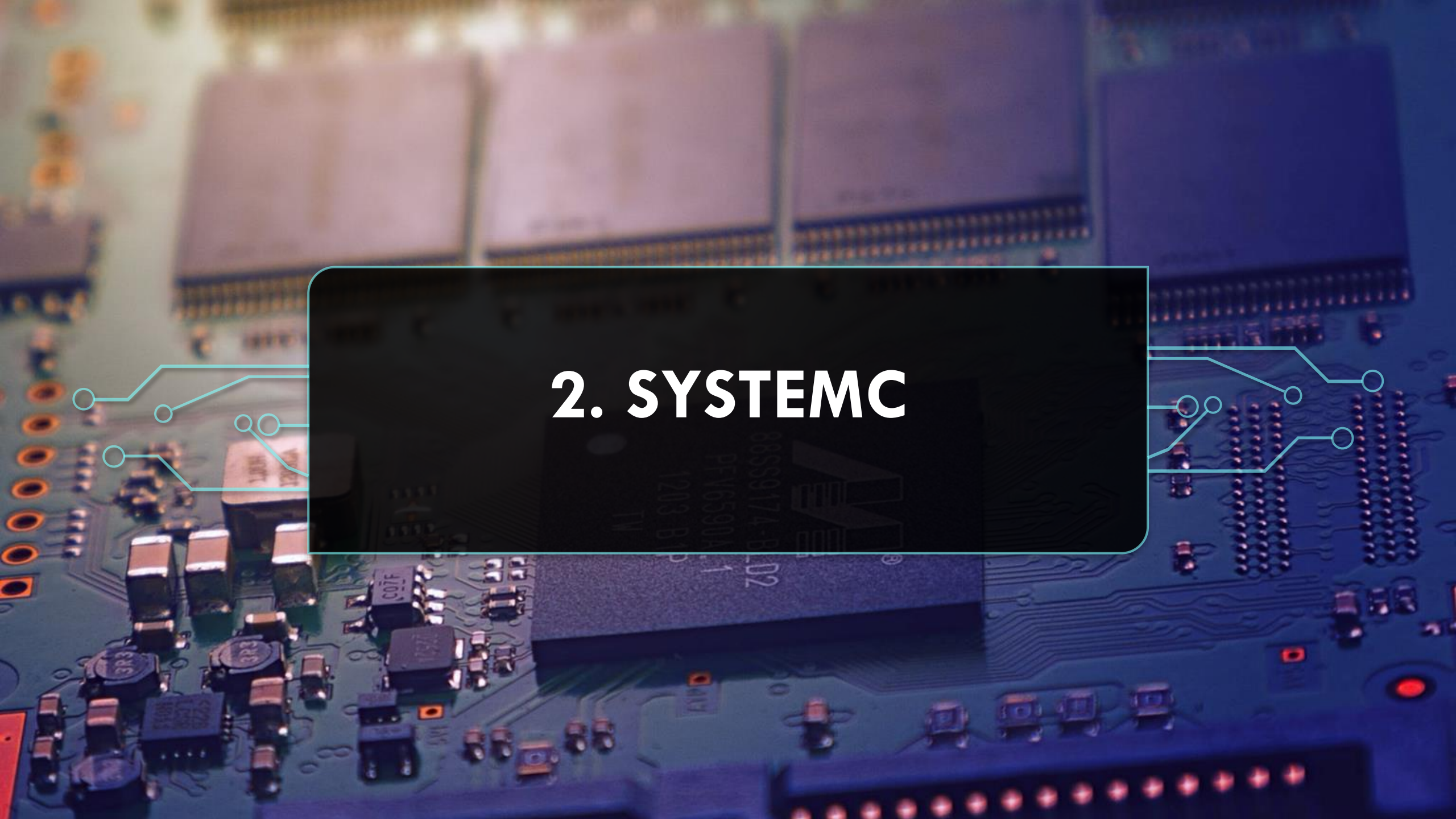
SHOWING CONSTRUCTORS AND DESTRUCTORS

No return type listed for constructors or destructors

Constructors

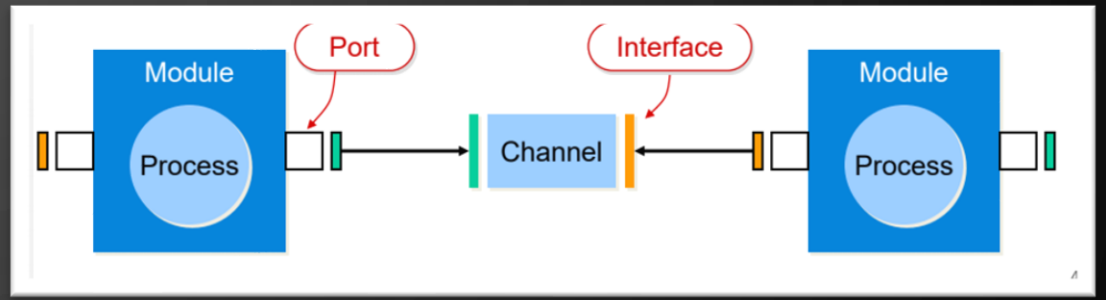
Destructor





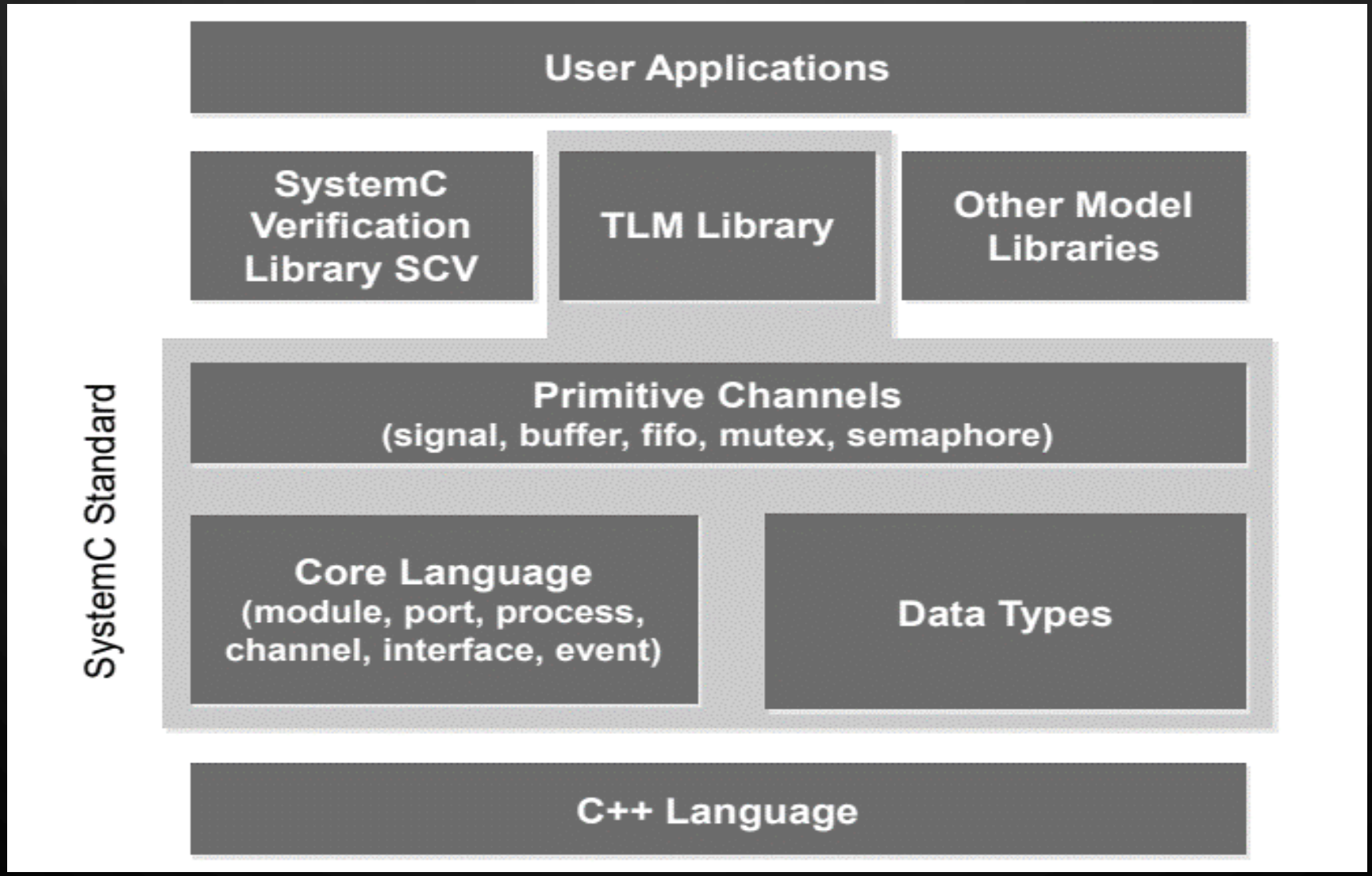
2. SYSTEMC

FEATURES OF SYSTEMC

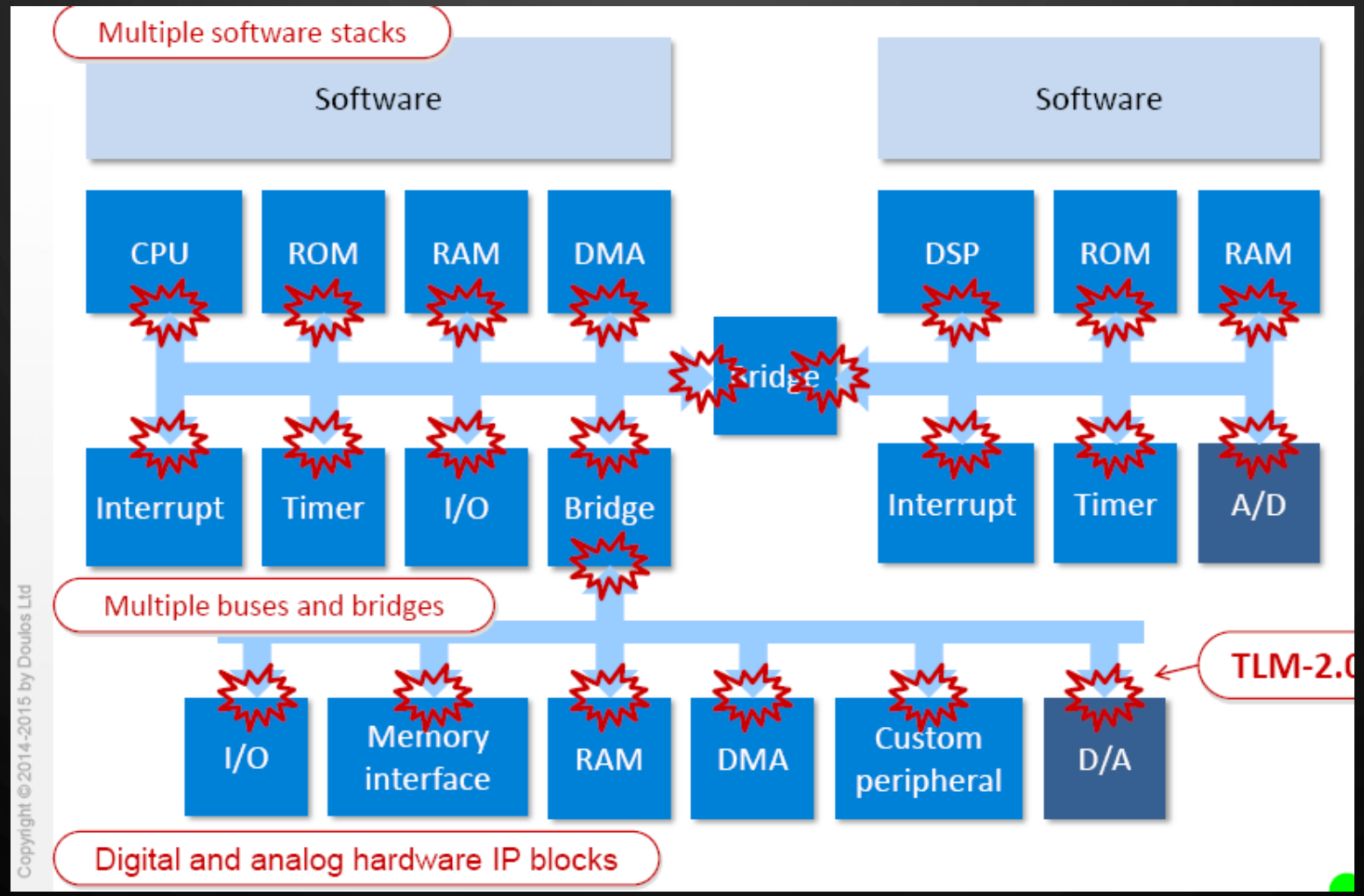


- MODULES (STRUCTURE)
- PORTS (STRUCTURE)
- PROCESSES (COMPUTATION, CONCURRENCY)
- CHANNELS (COMMUNICATION)
- INTERFACES (COMMUNICATION REFINEMENT)
- EVENTS (TIME, SCHEDULING, SYNCHRONIZATION)
- DATA TYPES (HARDWARE, FIXED POINT)

ARCHITECTURE OF SYSTEMC



TYPICAL USE CASE: VIRTUAL PLATFORM



SYSTEMC DATA TYPES

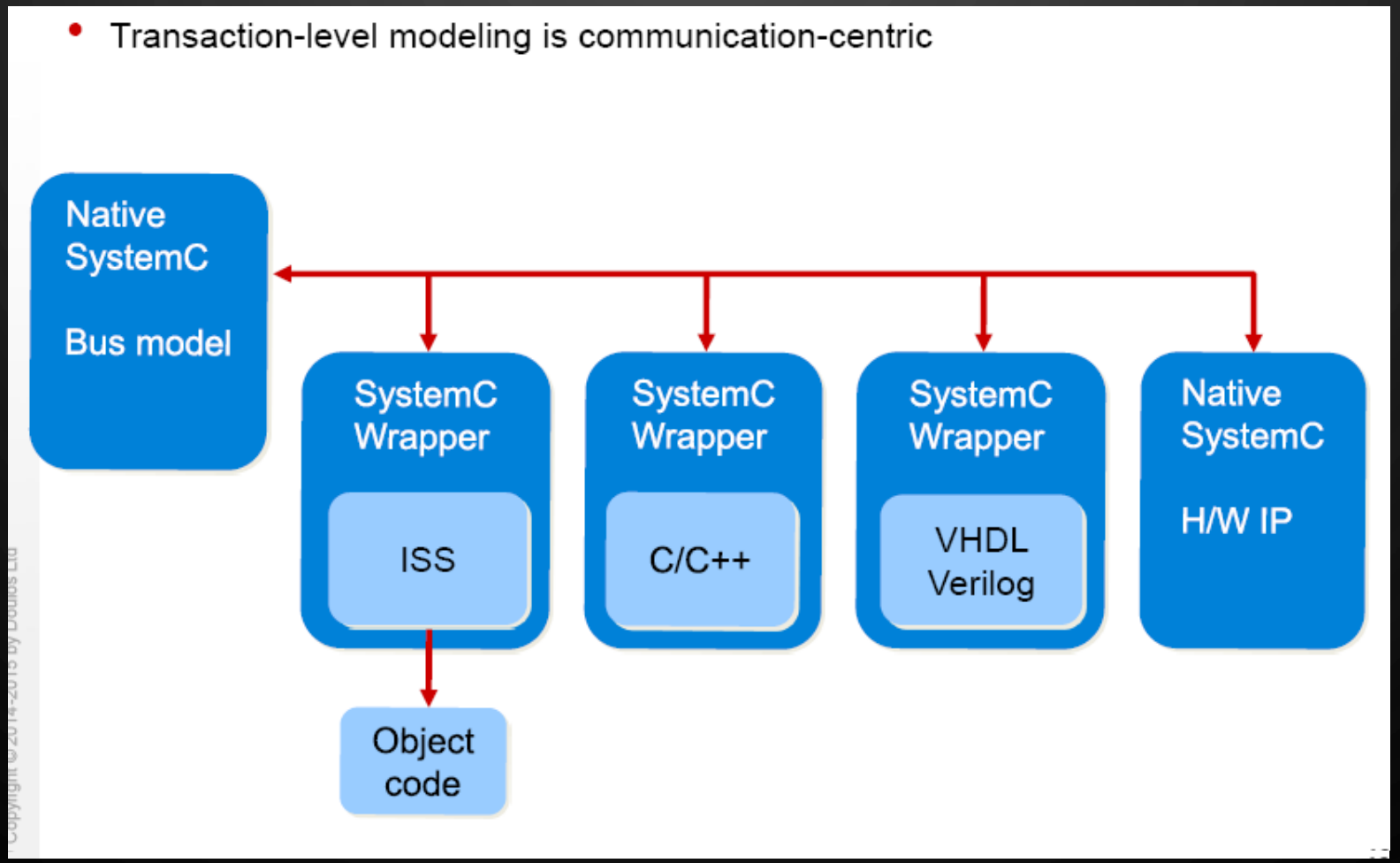
In namespace sc_dt::

Template	Base class	Description
sc_int<W>	sc_int_base	Signed integer, W < 65
sc_uint<W>	sc_uint_base	Unsigned integer, W < 65
sc_bigint<W>	sc_signed	Arbitrary precision signed integer
sc_biguint<W>	sc_unsigned	Arbitrary precision unsigned integer
		(intermediate results unbounded)
sc_logic		4-valued logic: '0' '1' 'X' 'Z'
sc_bv<W>	sc_bv_base	Bool vector
sc_lv<W>	sc_lv_base	Logic vector
sc_fixed<>	sc_fix	Signed fixed point number
sc_ufixed<>	sc_ufix	Unsigned fixed point number

WHAT IS SYSTEMC USED FOR?

- VIRTUAL PLATFORM
- ARCHITECTURAL EXPLORATION, PERFORMANCE MODELING
- SOFTWARE DEVELOPMENT
- REFERENCE MODEL FOR FUNCTIONAL VERIFICATION
- AVAILABLE BEFORE RTL - **EARLY!**
- SIMULATES MUCH FASTER THAN RTL - **FAST!**
- HIGH-LEVEL SYNTHESIS
- USED BY
- ARCHITECTS, SOFTWARE, HARDWARE, AND VERIFICATION ENGINEERS

SYSTEMC IS GLUE!



THE SYSTEMC LANGUAGE

Introduction to SystemC

- Core Concepts and Syntax
- Data
- Modules and connectivity
- Processes & Events
- Channels and Interfaces
- Ports
- Bus Modeling
- Odds and Ends

LIMITED PRECISION INTEGER

SC_INT

```
int        i;
sc_int<8>  j;
i = 0x123;
sc_assert( i == 0x123 );

j = 0x123;
sc_assert( j == 0x23 );

sc_assert( j[0] == 1 );
sc_assert( j.range(7,4) == 0x2 );
sc_assert( concat(j,j) == 0x2323 );
```

Truncated to 8 bits

Bit select

Part select

Concatenation

- Other useful operators: arithmetic, relational, bitwise, reduction, assignment

  length() to_int() to_string() *implicit-conversion-to-64-bit-int*

LOGIC AND VECTOR TYPES

sc_logic and sc_lv<W>

- Values SC_LOGIC_0, SC_LOGIC_1, SC_LOGIC_X, SC_LOGIC
- Initial value is SC_LOGIC_X

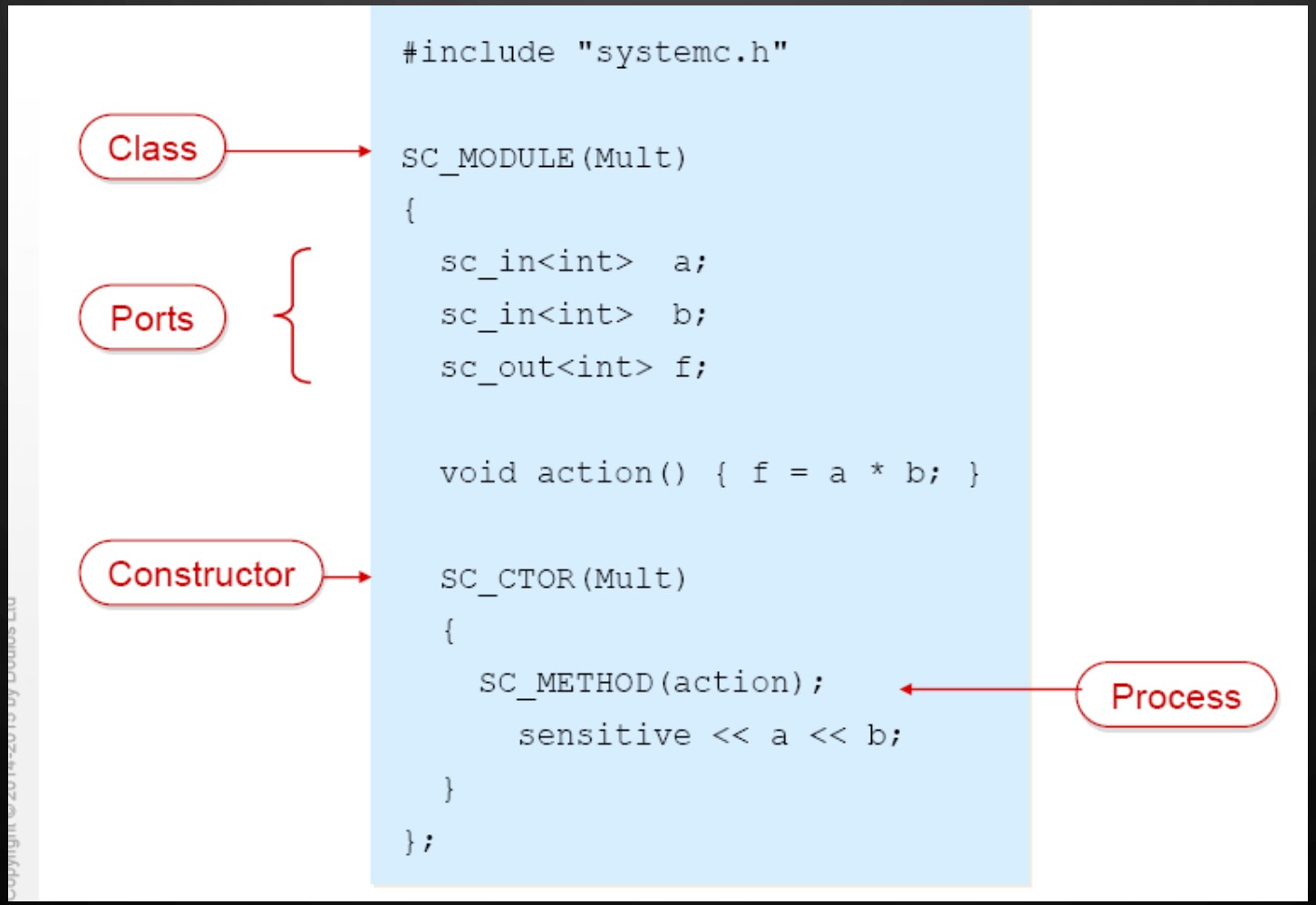
No arithmetic operators

Can write values as chars and strings, i.e. '0' '1' 'X' 'Z'

```
sc_logic R, S;
R = '1';
S = 'Z';
S = S & R;
```

```
sc_int<4>  n = "0b1010";
bool       boo = n[3];
sc_lv<4>   lv = "01XZ";
sc_assert( lv[0] == 'Z' );
n += lv.to_int();
cout << n.to_string(SC_HEX);
```

SC_MODULE


Class
Ports
Constructor
Process
#include "systemc.h"
SC_MODULE(Mult)
{
sc_in<int> a;
sc_in<int> b;
sc_out<int> f;
void action() { f = a * b; }
SC_CTOR(Mult)
{
SC_METHOD(action);
sensitive << a << b;
}
};

SC_MODULE OR SC_MODULE?

- Equivalent :

```
SC_MODULE(Name)
{
  ...
};
```

```
struct Name: sc_module
{
  ...
};
```

```
class Name: public sc_module
{
public:
  ...
};
```

SEPARATE HEADER FILE

```
// mult.h
#include "systemc.h"

SC_MODULE(Mult)
{
  sc_in<int>  a;
  sc_in<int>  b;
  sc_out<int> f;

  void action();

  SC_CTOR(Mult)
  {
    SC_METHOD(action);
      sensitive << a << b;
  }
};
```

```
// mult.cpp

#include "mult.h"

void Mult::action()
{
  f = a * b;
}
```

- Define constructor in .cpp?
  Yes - explained later

PORT BINDING

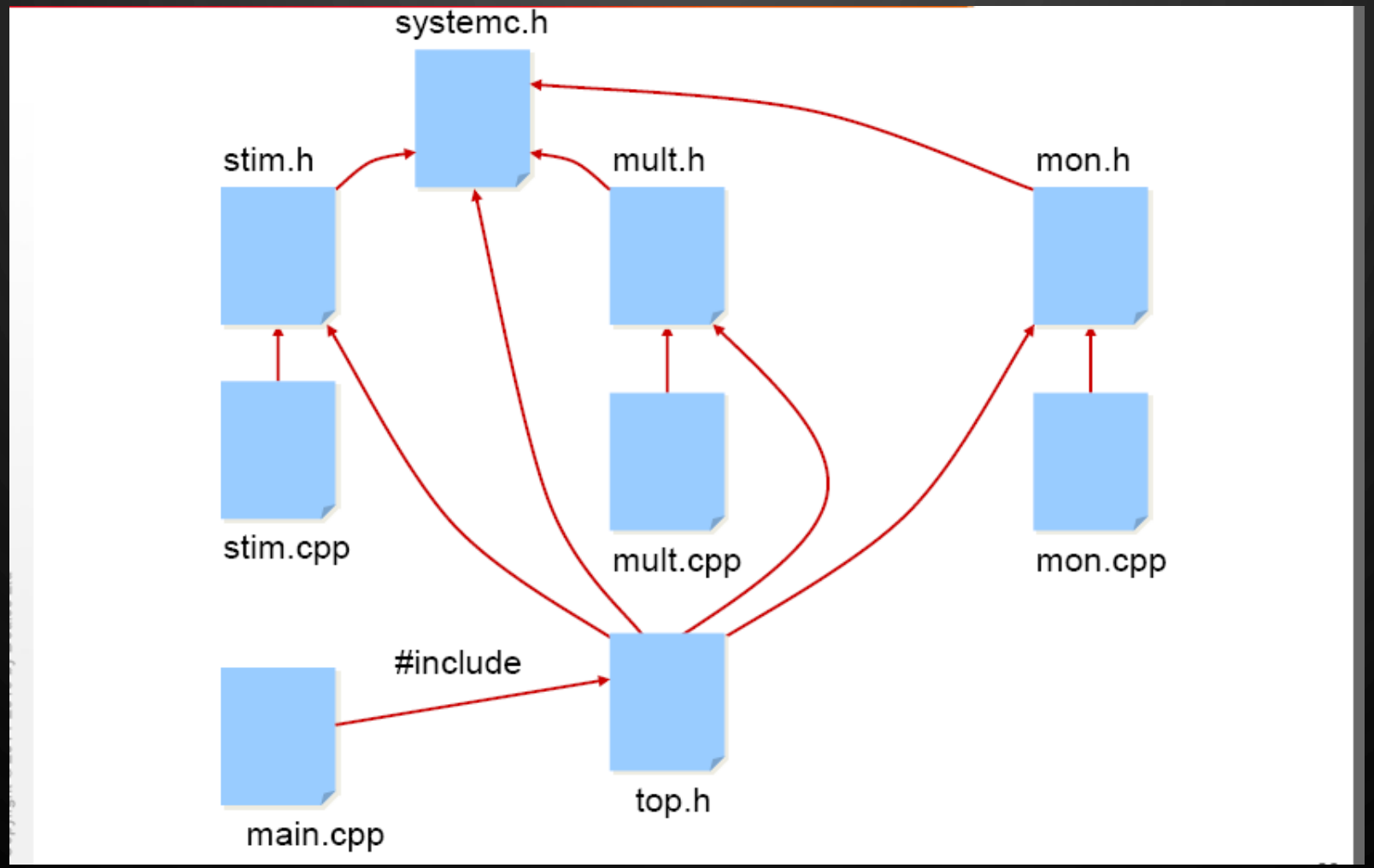
```
SC_CTOR(Top)
: testclk("testclk", 10, SC_NS),
  stim1("stim1"),
  uut("uut"),
  mon1("mon1")
{
  stim1.a(asig);
  stim1.b(bsig);
  stim1.clk(testclk);

  uut.a(asig);
  uut.b(bsig);
  uut.f(fsig);

  mon1.a.bind(asig);
  mon1.b.bind(bsig);
  mon1.f.bind(fsig);
  mon1.clk.bind(testclk);
}
```

Alternative function

Port name

Channel name

SUMMARY OF FILES



KINDS OF PROCESS

- Processes
  - Must be within a module (not in a function)
  - A module may contain many processes
- Three different kinds of process
  - Methods SC_METHOD
  - Threads SC_THREAD
  - Clocked threads SC_CTHREAD (for synthesis)
- Processes can be *static* or *dynamic*

SC_METHOD EXAMPLE

```
#include <systemc.h>

template<class T>
SC_MODULE(Register)
{
  sc_in<bool> clk, reset;
  sc_in<T>    d;
  sc_out<T>   q;

  void entry();

  SC_CTOR(Register)
  {
    SC_METHOD(entry);
      sensitive << reset;
      sensitive << clk.pos();
  }
};
```

```
template<class T>
void Register<T>::entry()
{
  if (reset)
    q = 0; // promotion
  else if (clk.posedge())
    q = d;
}
```

- SC_METHODs execute in zero time
- SC_METHODs cannot be suspended
- SC_METHODs should not contain infinite loops

SC_THREAD EXAMPLE

```
#include "systemc.h"

SC_MODULE(Stim)
{
  sc_in<bool> Clk;
  sc_out<int> A;
  sc_out<int> B;

  void stimulus();

  SC_CTOR(Stim)
  {
    SC_THREAD(stimulus);
      sensitive << Clk.pos();
  }
};
```

```
#include "stim.h"

void Stim::stimulus()
{
  wait();
  A = 100;
  B = 200;
  wait();
  A = -10;
  B = 23;
  wait();
  A = 25;
  B = -3;
  wait();
  sc_stop();
}
```

for Clk edge

Stop simulation

- More general and powerful than an `SC_METHOD`
- Simulation may be slightly slower than an `SC_METHOD`
- Called once only: hence often contains an infinite loop

REFERENCES

•IEEE 1666
standards.ieee.org/getieee/1666/download/
1666-2011.pdf

•ASI SystemC 2.3.1
www.accellera.org

•On-line tutorials
www.doulos.com/knowhow/systemc