CLASSES I

OBJECT-ORIENTED PROGRAMMING (OOP)

Object-oriented programming (OOP) is a programming paradigm based upon the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods). A feature of objects is that an object's own procedures can access and often modify the data fields of itself (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.

WIKIPEDIA, "Object-Oriented Programming" n.d., para. 1

CLASSES AND OBJECTS

Class: **An extensible template for creating objects.**

Object: **A complex variable which is an instance of a class.**

OOP CORE FEATURES

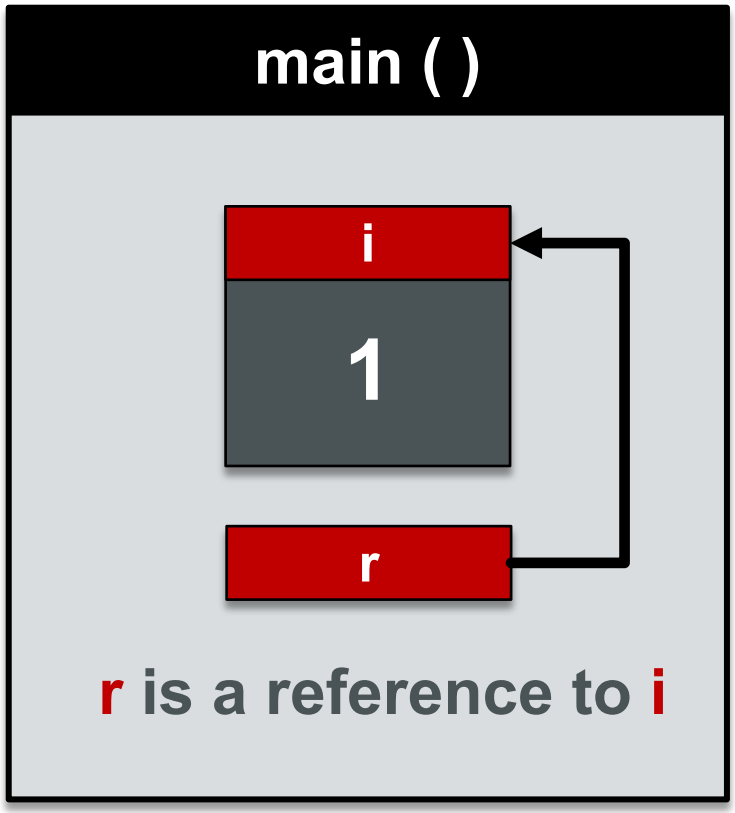
Encapsulation: **Controlling access to object data and behavior.**

Abstraction: **Only exposing relevant object data and behavior per interaction.**

Inheritance: **Sharing object data and behavior to eliminate code repetition.**

Polymorphism: **Sharing a common interface for related object types.**

REVIEW: REFERENCES



cout << &i; prints address 0x7ffeea769a68
cout << &r; prints address 0x7ffeea769a68

Variables i and r have the same memory address because they are different aliases or names for the same memory location

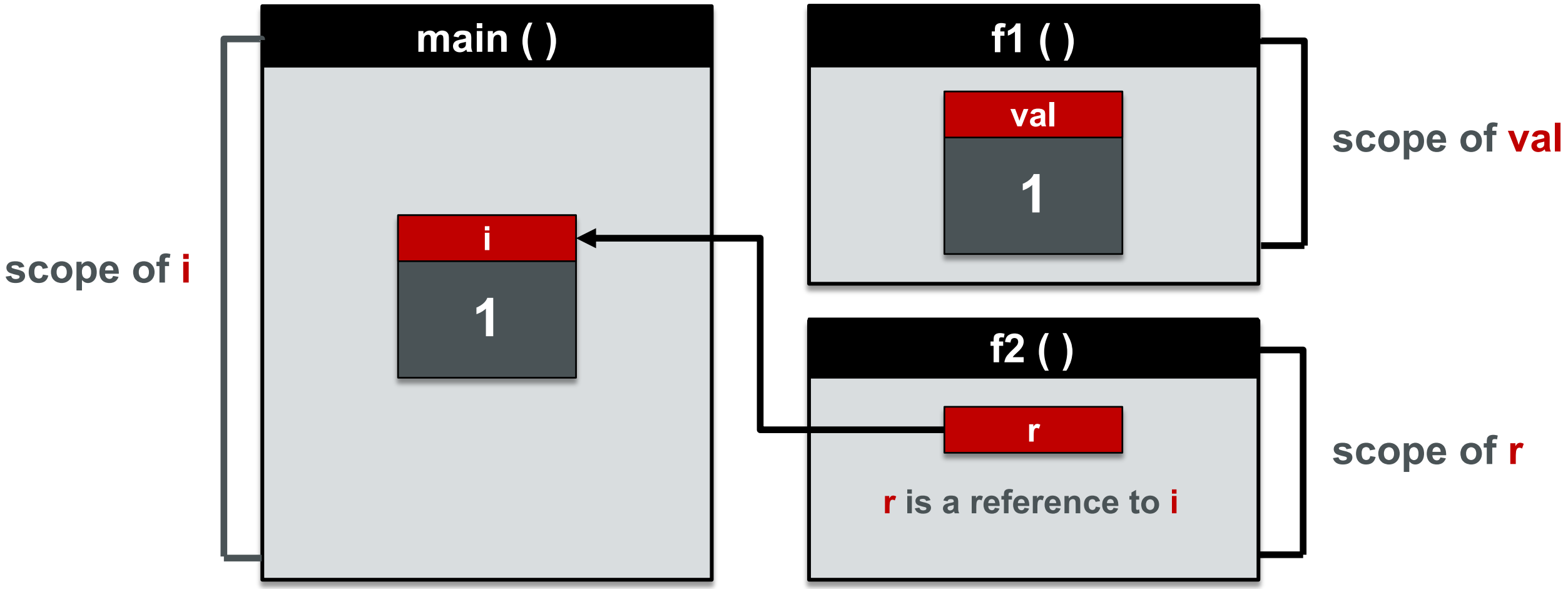
Therefore, they have the same value 1

REVIEW: SCOPE VS LIFETIME

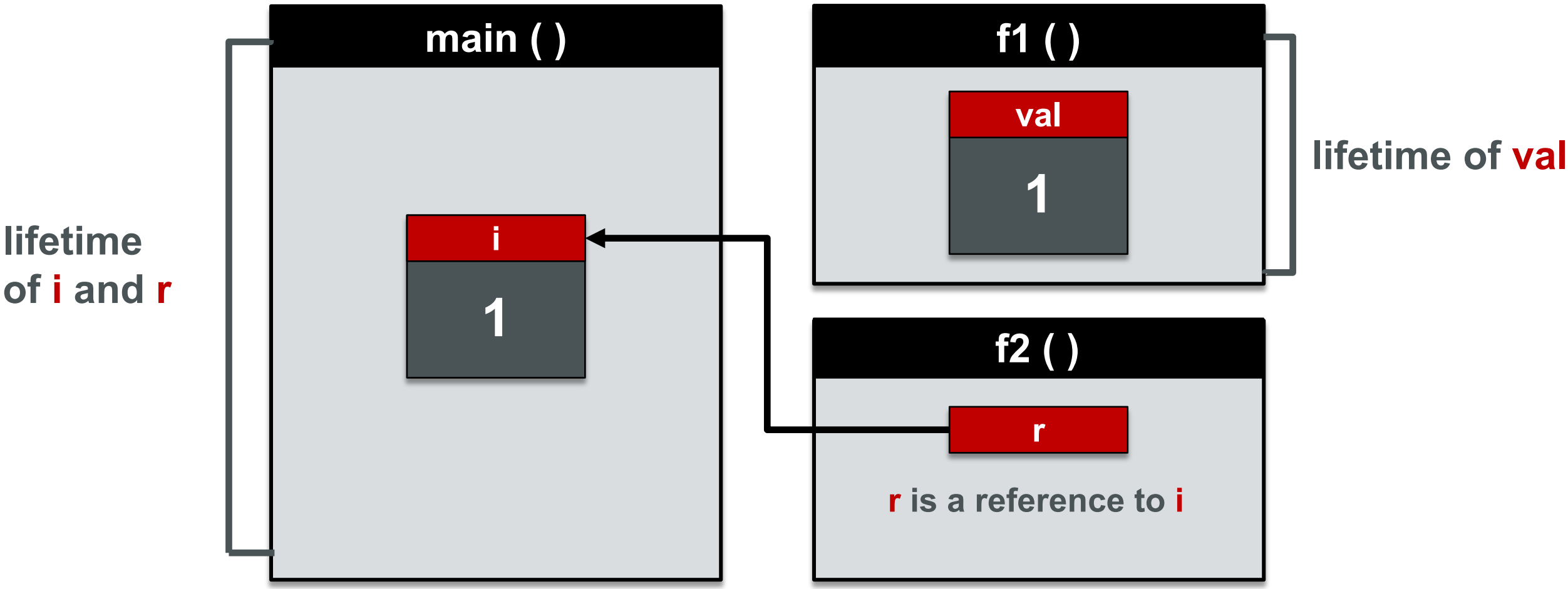
Scope: the area of a program where a variable name is accessible

Lifetime: the period of time in which a memory location exists

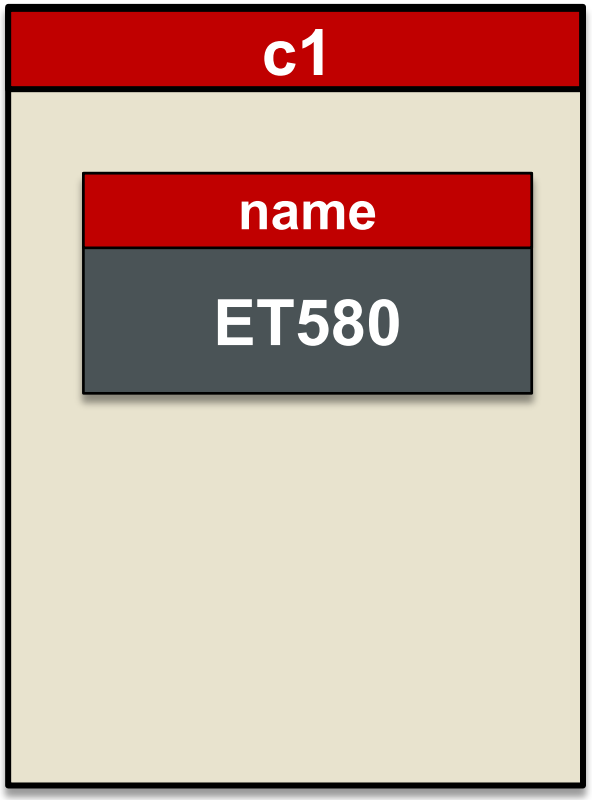
REVIEW: SCOPE



REVIEW: LIFETIME



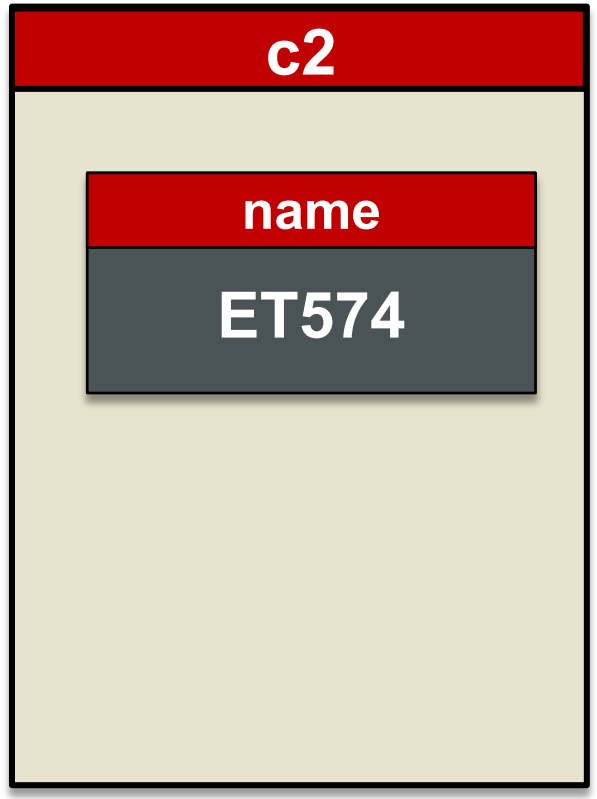
OBJECTS



0x7ffeeb5b6a18

each object is an **instance** of a class

each **object** has a distinct **memory address** and memory space for its data members



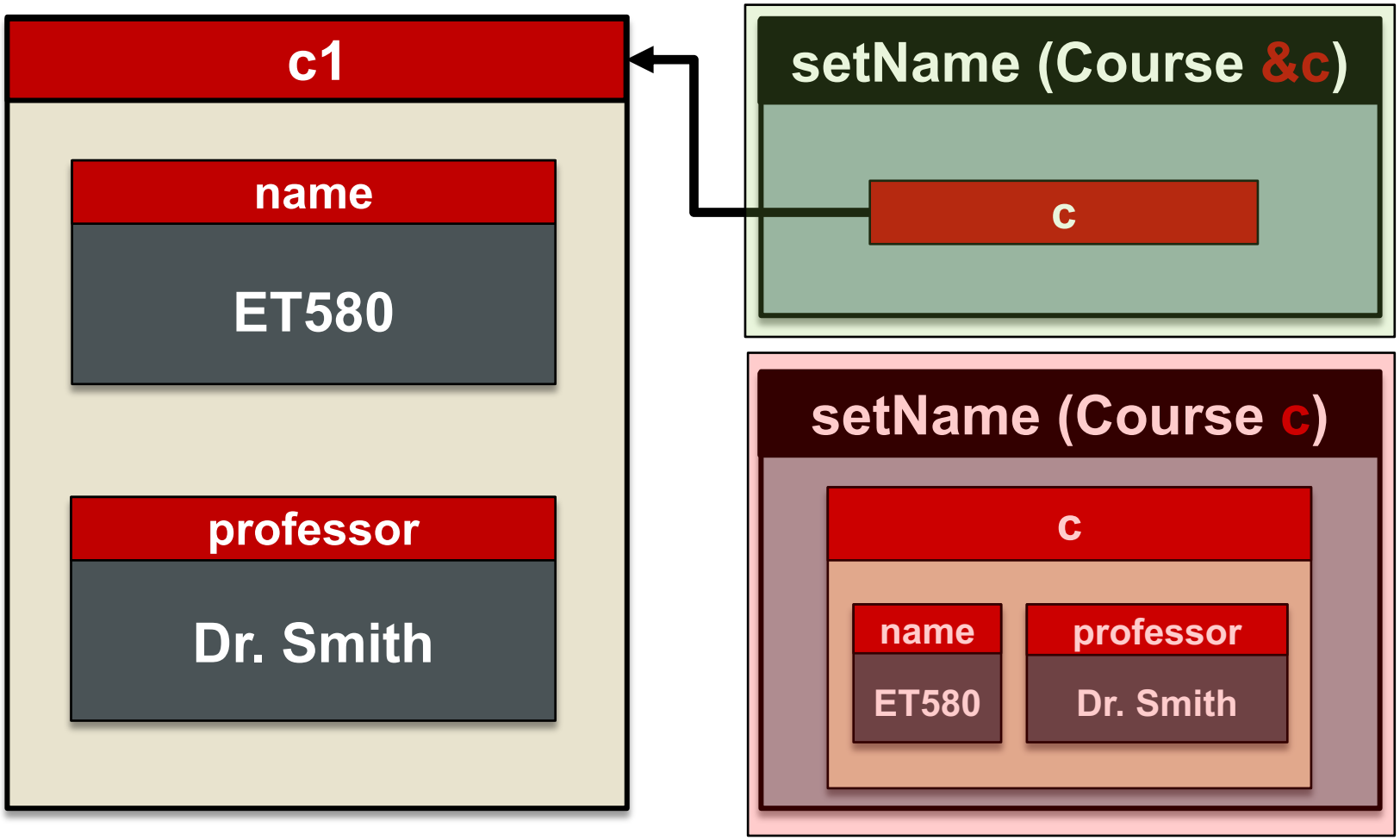
0x7ffee63f79a8

DATA MEMBERS: DOT OPERATOR

Syntax: object.DataMember

Example: c1.name ← access the c1 data member

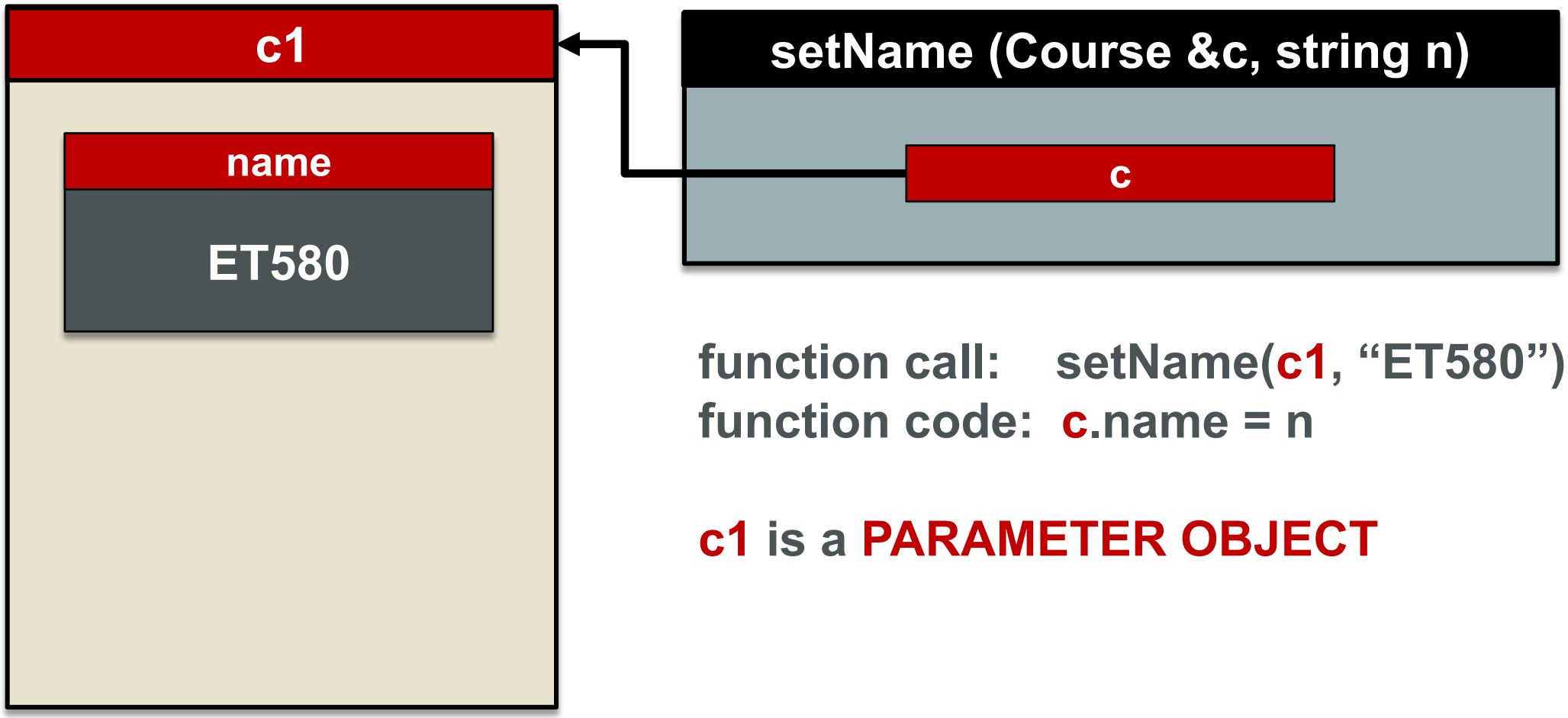
PASS OBJECT BY REFERENCE VS VALUE



OK
original object is modified

WRONG
original object is not modified

NON-MEMBERS: PARAMETER OBJECTS



function call: setName(**c1**, “ET580”)
function code: **c**.name = n

c1 is a **PARAMETER OBJECT**

MEMBER FUNCTIONS : DOT OPERATOR

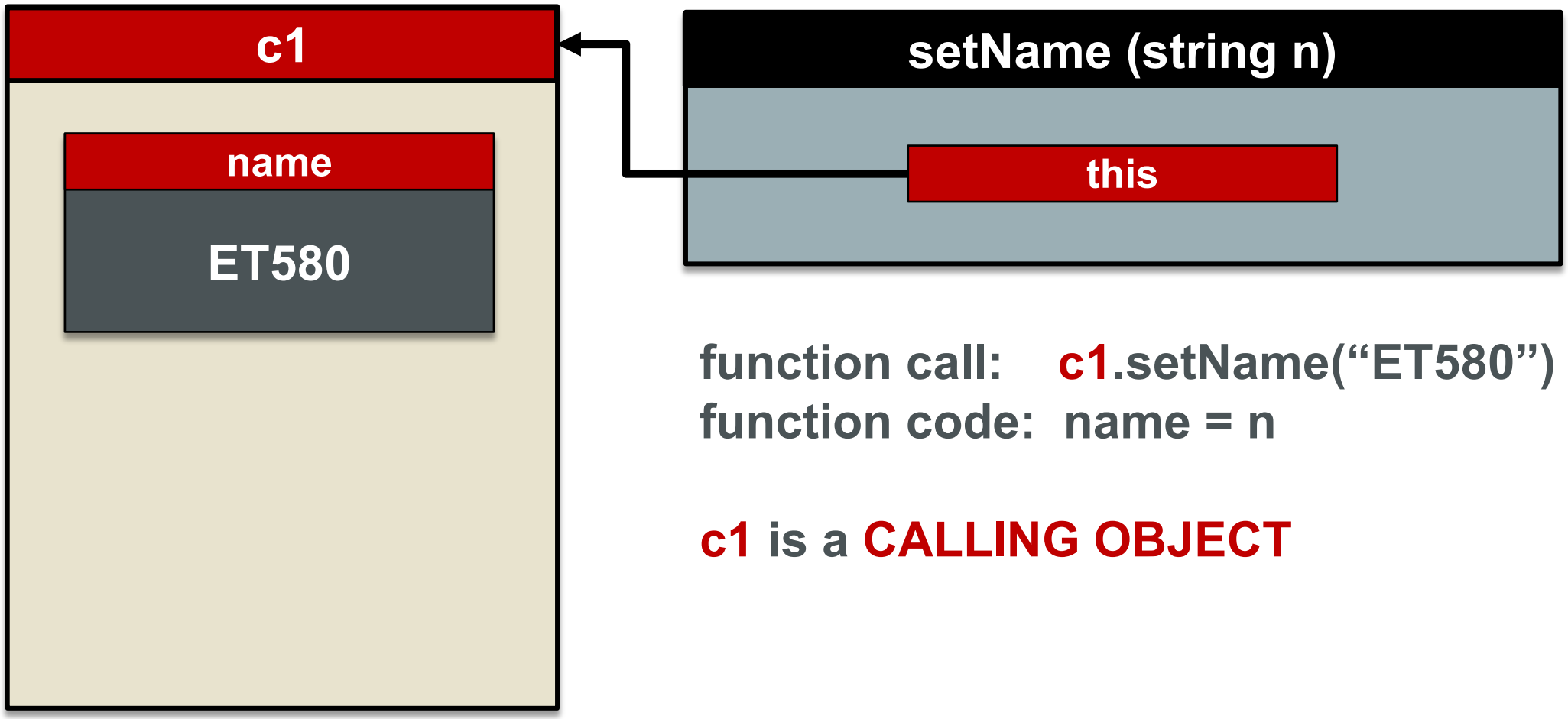
Syntax: callingObject.classMemberFunction( )

Example: c1.getName() – access the c1 class member function

Dot operator is used to access class members of the calling object
The calling object calls the function with the object as an implicit reference

```
void someMemberFunction( string n ) {
    name = n;    ← direct access to the calling object's data members
    output( );   ← direct access to the calling object's member functions
}
```

MEMBERS: CALLING OBJECTS



function call: c1.setName("ET580")
function code: name = n

c1 is a CALLING OBJECT

COMMON MEMBER FUNCTIONS

Constructor: initializes object data members upon creation
Declaration: `Classname( )`

Accessor: reads a specific data member of an object
Declaration: `datatype getDataName( )`

Mutator: edits a specific data member of an object
Declaration: `void setDataName( datatype data )`

INTERNAL/EXTERNAL DEFINITIONS

Declaration: **specify the function name, return type and parameters**

Definition: **specify the body of the function**

Internal: **declare/define within the class in one block of code**

External: **declare within the class, define outside of the class**

External definitions require the **:: scope resolution operator**
External Function Name Syntax: **ClassName::FunctionName( )**

ENCAPSULATION

Private: only accessible by member functions

Public: accessible by any function

Accessors and Mutators provide managed access to private area

By default, all class members are private unless specified otherwise

STRUCTURES

Structures existed before **classes** in the C programming language
Structures are still supported in C++ and are comparable to classes

Class members are **private** by default
Structure members are **public** by default

Structures in C++ are frequently used as complex data types
where core OOP features such are not needed