12/06/2017 afternoon

Base grammar

- The minimal code: *int main() { return 0;}*
- Defines the names, *i.e.* variables, are valid inside the module.
- The part of a code delimited by a *{ }* pair, it is called: block.

```
int main() {
  {
      // empty block
  }
  return 0;
  }
```

Programming jargon

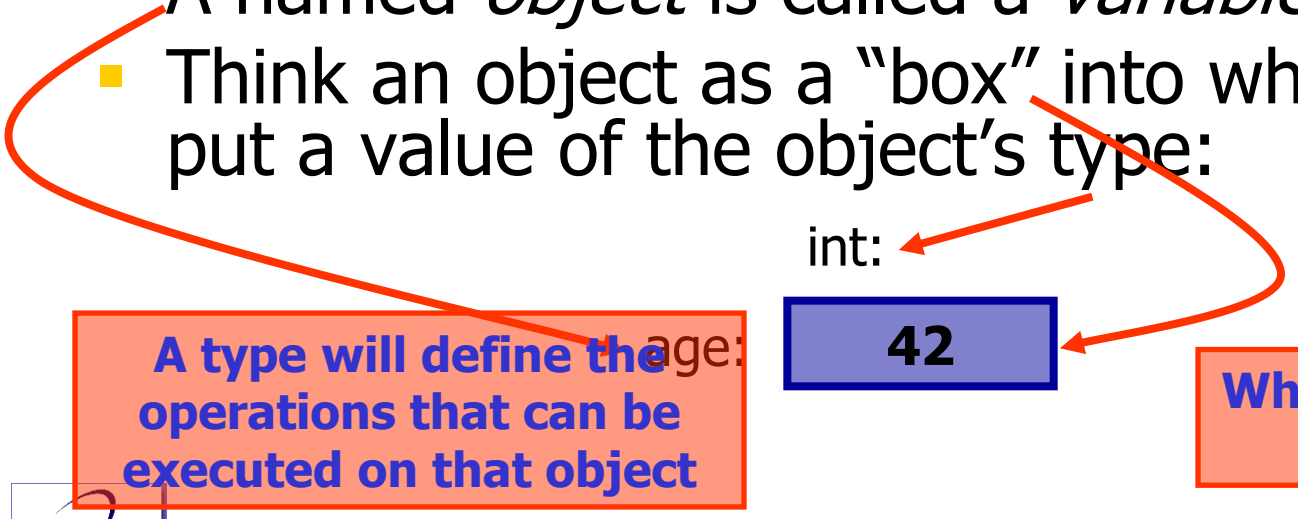
- The most basic building block of a program is an *expression*.
- An *expression* computes a value from a number of operands.
- A part of a code that specifies an action is called a *statement*.
- In C++ an ***expression* ending with a *;*** is a ***statement***: *return 0;*
- A *statement* that introduces a new name into a program is called a ***declaration***.
- A *statement* that introduces a new name into a program and sets aside memory for a variable is called a ***definition***.

Base grammar

- The minimal code: *int main() { return 0;}*
- All the declarations, statements, expressions etc. etc. must be terminated by a *;*
- Free format, you can write where and how you like!
- Case sensitive: *Ciccio != ciccio != ciCcio*
- Double *slashes*: *//* , mark the beginning of a comment line. It ends at the end of the line.

Programming jargon

- To calculate something, we need somewhere to read and write into; i.e. we need a "place" in PC memory to read from or write to. We call such a "place" an *object*.
- An *object* is a region of memory with a *type* that specified what kind of information can be placed in it.
- A named *object* is called a *variable.*
- Think an object as a "box" into which you can put a value of the object's type:



A type will define the operations that can be executed on that object

Why we need to declare a type?

Programming jargon

- Type & Objects:
  - A *type* defines a set of possible *values* and a set of operations (for an *object*).
  - An *object* is some memory that holds a *value* of a given type.
  - A *value* is a set of bits in memory interpreted according to a *type*.
  - A *variable* is a named *object*.
  - A *declaration* is a *statement* that gives a name to an *object*.
  - A *definition* is a *declaration* that sets aside memory for an object.

Base grammar: Types

- Types:
  - Every name and expression is associated to a type determining the operations that can be executed with it:
    - *int prova;*
  - Specifies that *prova* is of integer type, that is, is an integer variable.
  - A types defines the usage that you will have of a name (*prova*) or expression (*prova=5)*
- Declaration
  - It introduces a name (variable, function etc.) in a program (module), specifying the type.
- All names, used by a module, must be declared one and only once.

Base grammar: Names

- A name identifies: a variable, a class, a function, a namespace, a type.
- A name can be a sequence of characters, digits and underlines (_).
- A name must always start with a character.
- A name cannot contain spaces or punctuation marks.
- A name can start with one or more underlines but it is compiler dependent.
- A name cannot ever be none of the C++ reserved word:

  asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

- A name cannot ever be none of the operator names:

  and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

- Rules and exclusions are compiler dependent.

Base grammar: Declaration

- Declaration with multiple names:
  - It is possible to comma saparated list more than one name in the same statement:

int a,b,c,d; float a,f,g;

- Declaration with initialization, i.e. definition:
  - It is possible to declare a name and assign it an initial value:

int a=1;

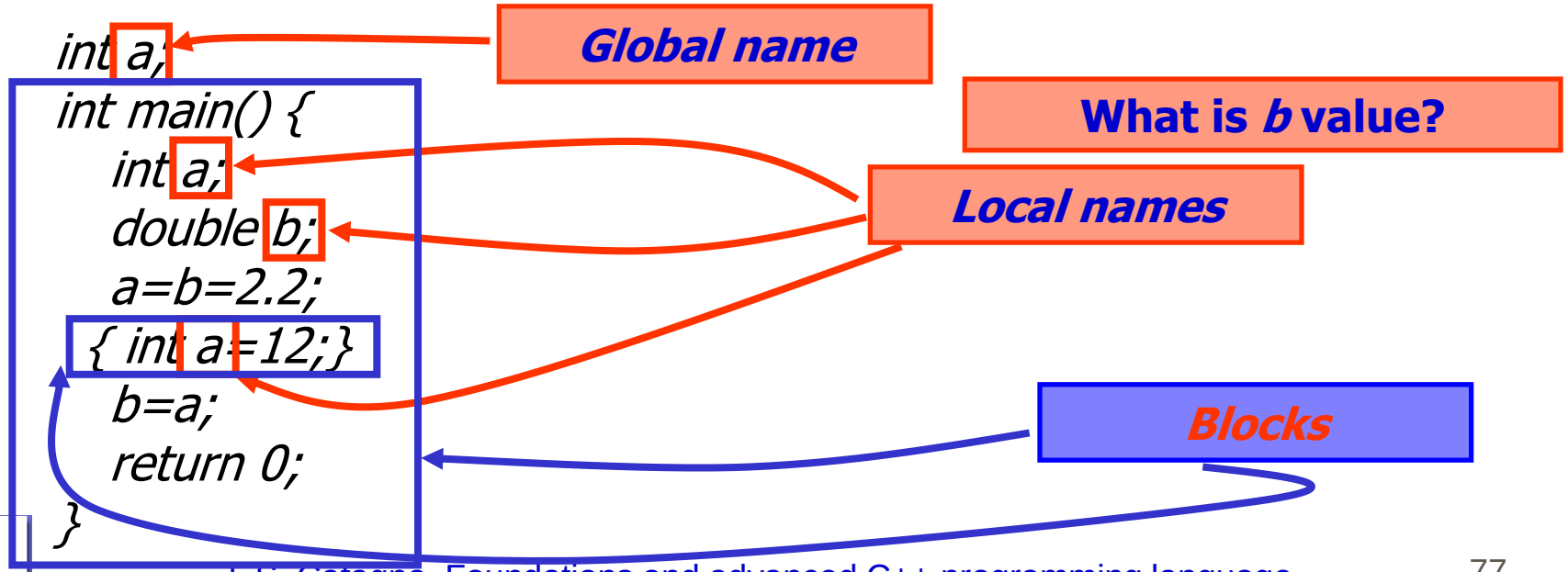
  - Another notation can be used, the functional notation:

int a(1);

  - The second case better show that a variable is created calling a specialized creator.

Base grammar: Declaration

- Declaration validity:
  - A declaration introduces a name into a scope, *i.e.* the name can be used only inside that specific program block.
  - When a name is declared into a scope it is called local. A local name is valid up to the end of the scopy into wich has been declared, *i.e.* the block where has been declared.
  - When a name is declared outside scopes, it is called global. The global name scope extends from the declaration up to the end of the file into wich has been declared.



Base grammar: Declaration

- Declaration validity
  - A name scope starts where the name as been declared, *i.e.* just after the declarator and before the initialization.

```
int a=a;
```

  - In case of name ambiguities, *i.e.* same local and global names, it is possible to use the scope resolution operator **::** to solve the ambiguities using the global name.

```
int a;
int main() {
    int a;
    double b;
    ::a=b=2;
  { int a=12;}
    b=a;
    return 0;
}
```

The global *a* is used

What is *b* value?

Base grammar: Types

- Base types:
  - char, wchar_t
  - int, short int, long int, unsigned int
  - bool
  - float
  - double, long double

Name	Description	Size*	Range*
`char`	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
`int`	Integer.	1word	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`short int` `short`	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`long int` `long`	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean value. It can take one of two values: true or false.	1byte	true or false
`float`	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
`double`	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
`long double`	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
`wchar_t`	Wide character.	2bytes	1 wide character

Base grammar: Literals

- Literals, that is how to use a fixed value, *i.e.* not a variable, in a code.
  - ***char***: a character prime delimited: *'a', '1'*
  - ***int***eger: they can be decimals, octals, hexadecimals and characters.
    - Decimals: *1 12345 555444333*
    - Octal (must begin with zero): ***0 0****2* ***0****1234567*
    - Hexadecimal (must start with a zero followed by and *x*): ***0x****abc* ***0x****Bac1Eabb5acc1* ***0x****ff*
    - Character: *a,b,c,d,e,f → 10,11,12,13,14,15* in hexadecial, ***U*** and ***L*** are suffixes to specify an *unsigned integer* or a *long: 3****U****,4****L****,5****UL.***

Base grammar: Literals

- Literals:

Floating point: a floating point literal is always a double*:*

3.14 0.123 6.3e-15

If a float literal is needed, the postfix *f* or *F* can be used to specify it.

If a long double is needed, the postfix *l* or *L* can be used to specify it.

3.14F 5.123l

Please note, a space is not a legal character. So, for example: *6.3 e – 15*, wil be interpreted as four separated parts and considered as a syntax error.

Base grammar: Types

- Base Types:
  - There is also a special type: *void*
  - It is void (of course …), but there cannot be void types, so it is used to specify that a function doesn't return any value or can be used to store a pointer to an object of unknown type:

    void a,b,c; // Error! Objects cannot be void.

    void f(); // Correct. Is the declaration of a function that doesn't return any value

    *void *g(); // Correct. Declaration of a function returning a pointer to an object of unknown type.*

Base Grammar: Types

- User defined type:
  - A **declaration** preceded by the keyword: *typedef* , introduces (declares) a new name for a type, instead of a new name for an object of a given type:

```
typedef int Fscaf; // Introduce the name Fscaf for an int
typedef double My_double; // Introduce the name My_double
    for a double
```

  - You can think about *typedefs as synonymous* with other type, instead of actual new types.
  - Extremely handy when it comes to reduce typing of very long type names or to reference to a specific type in a given part of the code.

Base Grammar: Enum

- An user defined type is the enumerator: *enum.*
  - It define a range of user specified integer values:

```
enum {UNO, DUE, TRE}; // UNO==0, DUE==1, TRE==2
enum numeri_interi {UNO, DUE, TRE};
void f(numeri_interi a){
   switch (a){
    case UNO:
         cout << " 1: " << a << endl;
         break;
    case DUE:
        cout << " 2: " << a << endl;
        break;
    default:
      cout << " Ne uno ne due" << endl;
   }

}
```

Base Grammar: Enum

- An user defined type is the enumerator: *enum.*
  - If not specified, integer values are assigned in an increasing order, starting from 0:

    enum numbers {ZERO,ONE,TWO};
  - A range declaration is permitted:

    enum a1_5 {a=1,b=5};

    a1_5 quattro=4; a1_5 tre=a1_5(3);

Base Grammar: *Declaration*

- To summarize, a declaration is composed of:
  - *specifier base type declarator initizializer*;
    - *specifier*: is optional, it specifies type attributes. E.g.: *virtual, extern, const* .
    - *base type*: is the actual type. E.g.: int, long, unsigned, float, Pere.
    - *declarator*: is composed by a list of names and an operator on any name. Operators can be prefix or postfix:
      - *** prefix — **Operator: pointer to**
      - ** const* prefix — **Operator: const pointer to**
      - *[ ]* postfix — **Operator: array**
      - *( )* postfix — **Operator: function**
    - *initizializer* : it optional, it specialize the name (entity) to whom is referred to.
  - A declaration must byterminated by a*;* But for a *function* or *namespace definitions*.

Base grammar: const

- One of the most popular *specifier* is the keyword: *const* . This *specifier* introduces the attribute of not changeability for the type.
- In this way is possible to declare constant objects, that is not possible to modify:

  const double pi=3.14; // pi, cannot be changed

Base Grammar: Arrays

- Arrays are homogenous and ordered, i.e. aligned, aggregates of objects of the same type.
- To declare arrays of objects of a type T, the postfix operator [], in the *declarator*, must be used, and size must be specified:

  int a[5]; double a[5];

- Arrays can be initialized to a list of values, comma separated, enclosed between *{}*:

  int a[5]={1,2,3,4,5};
  double b[6]={1.,2.,3.,4.}
  (the latest definition is equivalent to: *double b[6]={1.,2.,3.,0.,0.,0.}*)

- In a definition, dimension can be omitted:

  int a[]={1,2,3,4,5};

Base Grammar: Arrays

- Array can be *randomly* accessed using the *subscription* operator *[]* .
- Warning!!! Elements are indexed starting from 0.

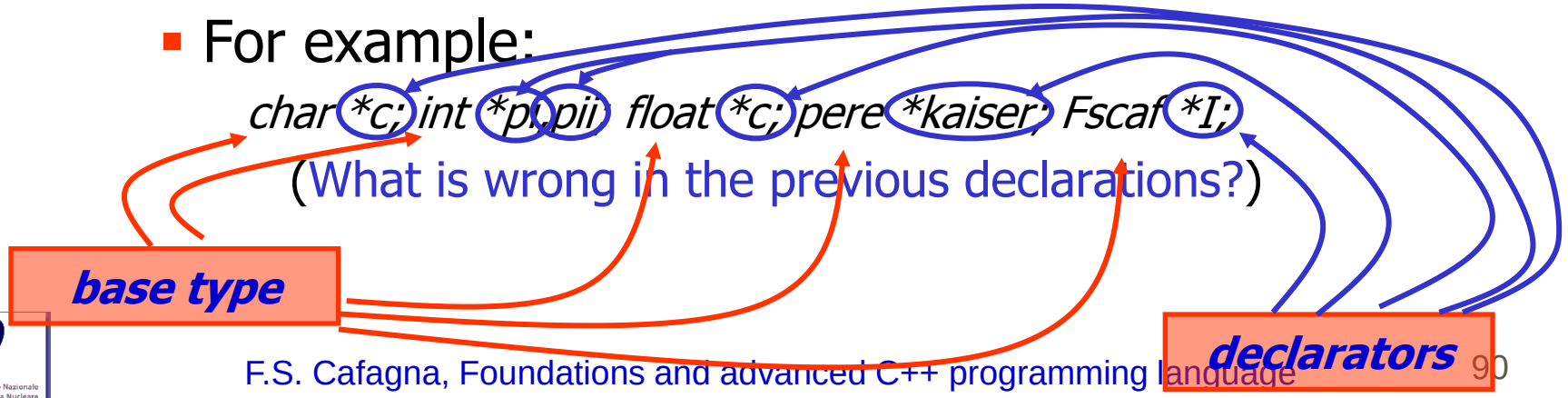
```
int a[5];
a[0]=1; // assign literal 1 to the first array element
a[4]=5; // assign literal 5 to the last array element
```

- Warning!!! There is no boundary check.

```
int a[5];
a[5]=6; // ERROR!!! Indexing more than the declared number of
    elements. It is conceptually wrong but legal!!!!
```

Base Grammar: Pointers

- Pointers:
  - For every type: T, a type T* does exist representing a pointer to a type T.
  - A variable of type T* contains the memory address of an object of type T.
  - To declare a name of type "pointer to …", the prefix * in declarator, must be used.
    - For example:



Base Grammar: Pointers

- A pointer *dereferences*: *i.e.* refers to an object pointed by the value stored in the pointer. This action is possible using the operator: *
- It maps the machine memory.
- The operator that applied to a name (varible), returns is memory address, is: &

  *int a=2; int *pa; pa=&a; *pa=3;*
- Operators * and &, are unary operators, that is they act on a single object.

Base Grammar: Pointers

- Algebraic operators can be applied on pointers:
  +, - , ++, -- .
- Warning!!! Sum is defined between a pointer and an integer constants, not between two pointers:
  *float c,d; float *pf=&c,*pf2=&d; pf=pf+3; pf2=pf2+pf; (error !!!!!)*
- Be careful with subtraction!!!!!
- Be careful with constants: **const*, *declarator operator* to define a const pointer, *const** is not a legal declarator because is considered part of the base type:

```
const int *pi; // pointer to an integer constant: pi+1 is permitted
int const *pi; // pointer to an integer constant: pi+1 is permitted
int *const pi; // constant pointer: pi+1 is not permitted
```

Base Grammar: Pointers

- Pointers to constant objects is extremely useful, for example, in function definition to avoid object modification in the function itself:
  - *char * strcpy(char *p, const char *q); // *q cannot be modified in the function strcpy*
- Pointer to *void:*
  - Any pointer can be assigned to a variable od type *void*:*

    *int *pi; void* pv=pi;*
  - *void** can be assigned to a *void** and can be compared to another *void**
  - But of course a *void** cannot be dereferenced: *void *pv; *pv=123; (not permitted!!!!)*
  - *It is used to pass or receive pointers from functions when no hypothesis can be done on the returned type.*