Lezione del 13/09/16

Exercise

- Using the quadratic equation solution write a list of statements that can implement:
  - Procedural programming;
  - Modular programming;
  - Object oriented programming;

Just focus on the expression and statements, write it on simple plain English, but try to implement the above mentioned programming style.

Exercise

- Let's write a program to solve the quadratic equation:

$$ax^2 + bx + c = 0, a \neq 0.$$

- We already know the solutions:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Exercise: Procedural Progr.

- So we can write our first procedure:
  1. Introduce the variable a;
  2. Introduce the variable b;
  3. Introduce the variable c;
  4. Introduce the variable x_1;
  5. Assign a value to a;
  6. Assign a value to b;
  7. Assign a value to c;
  8. Evaluate b^2 and subtract $4ac$;
  9. Evaluate square root of the result of the previous statement;
  10. Evaluate $-b$ and sum it to the result of the previous statement;
  11. Evaluate $2a$ and divide the result of the previous statement by it;
  12. Assign the result of the previous statement to x_1.

Will work?

I need to introduce some checks before using it!

Exercise: Procedural Progr.

So we can write our first procedure:

1. Introduce the variable a;
2. Introduce the variable b;
3. Introduce the variable c;
4. Introduce the variable x_1;
5. Assign a value to a;
6. Check that a contains a value greater than 0;
7. Assign a value to b;
8. Assign a value to c;
9. Evaluate b^2 and subtract $4ac$;
10. Check that the result of the previous statement is greater than 0;
11. Evaluate square root of the result of the previous statement;
12. Evaluate $-b$ and sum it to the result of the previous statement;
13. Evaluate $2a$, check that the result is greater than 0, and divide the result of the previous statement by it if it is greater than 0;
14. Assign the result of the previous statement to x_1.

Will work?

Are we sure a CPU knows how to calculate powers and square roots?

Exercise: Procedural Progr.

So we can write our first procedure:

1. Introduce the variable a;
2. Introduce the variable b;
3. Introduce the variable c;
4. Introduce the variable x_1;
5. Assign a value to a;
6. Check that a contains a value greater than 0;
7. Assign a value to b;
8. Assign a value to c;
9. Evaluate b^2 and subtract $4ac$;
10. Check that the result of the previous statement is greater than 0;
11. Evaluate square root of the result of the previous statement;
12. Evaluate $-b$ and sum it to the result of the previous statement;
13. Evaluate $2a$, check that the result is greater than 0, and divide the result of the previous statement by it if it is greater than 0;
14. Assign the result of the previous statement to x_1.

Will work?

What about digits? Are they variables or something else?

Exercise: Procedural Progr.

So we can write our first procedure:

1. Introduce the variable a;
2. Introduce the variable b;
3. Introduce the variable c;
4. Introduce the variable x_1;
5. Assign a value to a;
6. Check that a contains a value greater than 0;
7. Assign a value to b;
8. Assign a value to c;
9. Evaluate b^2 and subtract $4ac$;
10. Check that the result of the previous statement is greater than 0;
11. Evaluate square root of the result of the previous statement;
12. Evaluate $-b$ and sum it to the result of the previous statement;
13. Evaluate $2a$, check that the result is greater than 0, and divide the result of the previous statement by it if it is greater than 0;
14. Assign the result of the previous statement to x_1.

Will work?

What's about the operation on the objects? Where are results stored?

Exercise: Procedural Progr.

- So we can write our first procedure:

1. Introduce the variable a;
2. Introduce the variable b;
3. Introduce the variable c;
4. Introduce the variable x_1;
5. Assign a value to *a*;
6. Check that *a* contains a value greater than 0;
7. Assign a value to *b*;
8. Assign a value to *c*;
9. Introduce a temporary variable *t1*;
10. Assign the value contained in *b* to *t1*;
11. Assign the result of the multiplication of the value contained in *b* with the one in *t1* to *t1*;
12. Introduce a temporary variable *t2*;
13. Assign the result of the multiplication of the value contained in *a*, whit the one contained in *c* and with *4* to *t2*;
14. Introduce a temporary variable *t3*;
15. Assign the result of the subtraction of the value contained in *t1* with the one contained in *t2* to *t3*;
16. Check that *t3* contains a value greater than 0;
17. Introduce a temporary variable *t4*;
18. Evaluate the square root of the value contained in *t3* assigning the result to *t4*;
19. Introduce a temporary variable *t5*;
20. Evaluate the sign invertion of the value contained in *b* assigning the result to *t5*;
21. Assign the result of the sum of the value contained in *t5* with the one contained in *t4* assigning the result to *t5*;
22. Introduce a temporary variable *t6*;
23. Assign the result of the multiplication of the value contained in *a* with 2 assigning the result to *t6*;
24. Check that *t6* contains a value greater than 0;
25. Assign the result of the division of the value contained in *t5* with the value contained in *t6* assigning the result to *x1*;

Will work?

What's about the order of the operations?

Exercise: Procedural Progr.

- So we can write our first procedure:

1. Introduce the variable a;
2. Introduce the variable b;
3. Introduce the variable c;
4. Introduce the variable x_1;
5. Assign a value to *a*;
6. Check that *a* contains a value greater than 0;
7. Assign a value to *b*;
8. Assign a value to *c*;
9. Introduce a temporary variable *t1*;
10. Assign the value contained in *b* to *t1*;
11. Assign the result of the multiplication of the value contained in *b* with the one in *t1* to *t1*;
12. Introduce a temporary variable *t2*;
13. Assign the result of the multiplication of the value contained in *a*, whit the one contained in *c* and with *4* to *t2*;
14. Introduce a temporary variable *t3*;
15. Assign the result of the subtraction of the value contained in *t1* with the one contained in *t2* to *t3*;
16. Check that *t3* contains a value greater than 0;
17. Introduce a temporary variable *t4*;
18. Evaluate the square root of the value contained in *t3* assigning the result to *t4*;
19. Introduce a temporary variable *t5*;
20. Evaluate the sign invertion of the value contained in *b* assigning the result to *t5*;
21. Assign the result of the sum of the value contained in *t5* with the one contained in *t4* assigning the result to *t5*;
22. Introduce a temporary variable *t6*;
23. Assign the result of the multiplication of the value contained in *a* with 2 assigning the result to *t6*;
24. Check that *t6* contains a value greater than 0;
25. Assign the result of the division of the value contained in *t5* with the value contained in *t6* assigning the result to *x1*;

Will work?

What's about the exception handling?

The sqrt?

- How we can calculate the square root of a number?
  - There are several methods. Let's start evaluating roughly a seed approximating the positive real number S we want to calculate the square root.
    - If S ≥ 1, let D be the number of digits to the left of the decimal point;
    - If S < 1, let D be the negative of the number of zeros to the immediate right of the decimal point.
  - Then the rough estimation, being $n = \log_{100} S$, is:
    - D is odd -> D = 2n + 1, then use $\sqrt{S} \approx 2 * 10^n$;
    - D is even -> D = 2n + 2, then use $\sqrt{S} \approx 6 * 10^n$;

    (2 and 6 are used because they approximate the geometric means of the lowest and highest possible values with the given number of digits)
- Heron's method
  - If x is out initial rough guess of $\sqrt{S}$ and e is the error in our estimate than $S = (x + e)^2$ then (assuming e small):

$$e = \frac{S - x^2}{2x + e} \approx \frac{S - x^2}{2x} \rightarrow x \approx x + e = \frac{S + x^2}{2x} = \frac{x + \frac{S}{x}}{2}.$$

  - This became the new guess and we can iteratively update the value until the desired accuracy is obtained (hits is a quadratycaly convergent algorithm).
- Taylor series
- Root-finding algorithm
  - $\sqrt{S}, f(x) = x^2 - S = 0$
- … etc. etc …
- Almost all are iterative procedures, i.e. we need iteration statement or the possibility to jump back into our algorithm.

Exercise: Modular Progr.

- So we can write our first procedure:

1. Introduce the variable a;
2. Introduce the variable b;
3. Introduce the variable c;
4. Introduce the variable x_1;
5. Assign a value to a;
6. Check that a contains a value greater than 0;
7. Assign a value to b;
8. Assign a value to c;
9. Introduce a temporary variable $t1$;
10. Assign the value contained in b to $t1$;
11. Assign the result of the multiplication of the value contained in b with the one in $t1$ to $t1$;
12. Introduce a temporary variable $t2$;
13. Assign the result of the multiplication of the value contained in a, whit the one contained in c and with 4 to $t2$;
14. Introduce a temporary variable $t3$;
15. Assign the result of the subtraction of the value contained in $t1$ with the one contained in $t2$ to $t3$;
16. Check that $t3$ contains a value greater than 0;
17. Introduce a temporary variable $t4$;
18. Execute the square root algorithm using the value contained in $t3$ assigning the result to $t4$;
19. Introduce a temporary variable $t5$;
20. Evaluate the sign invertion of the value contained in b assigning the result to $t5$;
21. Assign the result of the sum of the value contained in $t5$ with the one contained in $t4$ assigning the result to $t5$;
22. Introduce a temporary variable $t6$;
23. Assign the result of the multiplication of the value contained in a with 2 assigning the result to $t6$;
24. Check that $t6$ contains a value greater than 0;
25. Assign the result of the division of the value contained in $t5$ with the value contained in $t6$ assigning the result to $x1$;

What is C++?

- What is C++ ?
  - **Modular Programming**, that is: *Decide which modules you want; partition the program so that data is hidden within modules.*
    - Increasing code complexity calls for the need of modularization. Algorithms can be subdivided into blocks implementing part of the procedure hiding data needed in this blocks.
    - Programmer must provide a module interface so the other blocks can use the module, via the interface, or the externally modifiable data, hidden in the module.

Exercise: Modular Progr.

- So we can modularize our procedure:

1. Define a module that stores the coefficient values and return it accordingly:
   value Coefficients(value, coef, action); where: *value* is the coeficient value, *coef* Is the cooefficient (*a=1, b=2, c=3*), *action* is (*read=1, write=2*).
2. Define a module that implement the square root algorithm:
   result Sqrt(value);
3. Define a module to calculate the discriminant:
   result Discriminant(bvalue, avalue,cvalue);
4. Define a module to calculate the first solution:
   result X1(avalue, svalue);
5. Define a module to calculate the second solution:
   result X2(avalue, svalue);
6. Store the value contained in a : *Coefficients(value, 1, 2)*;
7. Store the value contained in b : *Coefficients(value, 2, 2)*;
8. Store the value contained in c : *Coefficients(value, 3 2)*;
9. Introduce the variable x_1;
10. Introduce the variable x_2;
11. Assign the first result to x_1 : x_1 = *X1(Coefficients(0, 1, 1), Sqrt(Discriminant(Coefficients(0,2,1), Coefficients(0,1,1), Coefficients(0,3,1)))* ;
12. Assign the second result to x_2 : x_2 = *X2(Coefficients(0, 1, 1), Sqrt(Discriminant(Coefficients(0,2,1), Coefficients(0,1,1), Coefficients(0,3,1)))* ;

Very poor interfaces: we need documentation, prone to errors!

What about if I want to use different sqrt algorithms?

What is C++?

- What is C++ ?
  - **Modular Programming**, that is: *Decide which modules you want; partition the program so that data is hidden within modules.*
    - In C++ modules and data can be grouped into namespaces, implementing interfaces and distinguish module of data with identical names but caming from different interfaces.
    - Often modules became so complex that are difficult to maintain. An interface using simply a namespace is not enough.
    - C++ provides facilities to implement a module as an user defined type.

Exercise: Modular Progr.

- So we can modularize our procedure:

1. Define a module that stores the coefficient values and return it accordingly:
   value Coefficients(value, coef, action); where: *value* is the coeficient value, *coef* Is the cooefficient (*a=1, b=2, c=3*), *action* is (*read=1, write=2*).
2. Define a namespace Haron and the module that implement the square root algorithm according to Haron's method:
   result Haron::Sqrt(value);
3. Define a namespace Taylor and the module that implement the square root algorithm according Taylor's series method:
   result Taylor::Sqrt(value);
4. Define a module to calculate the discriminant:
   result Discriminant(bvalue, avalue,cvalue);
5. Define a module to calculate the first solution:
   result X1(avalue, svalue);
6. Define a module to calculate the second solution:
   result X2(avalue, svalue);
7. Store the value contained in a : *Coefficients(value, 1, 2)*;
8. Store the value contained in b : *Coefficients(value, 2, 2)*;
9. Store the value contained in c : *Coefficients(value, 3 2)*;
10. Introduce the variable x_1;
11. Introduce the variable x_2;
12. Assign the first result to x_1 : x_1 = *X1(Coefficients(0, 1, 1), Haron::Sqrt(Discriminant(Coefficients(0,2,1), Coefficients(0,1,1), Coefficients(0,3,1)))* ;
13. Assign the second result to x_2 : x_2 = *X2(Coefficients(0, 1, 1), Haron::Sqrt(Discriminant(Coefficients(0,2,1), Coefficients(0,1,1), Coefficients(0,3,1)))* ;

What is C++?

- What is C++ ?
  - **User-Definited Types**, that is: *Decide which types you want; provide a full set of operations for each type.*
    - The increase of a type complexity increase the need to safe data and hide procedure details to the user. We want the user focus to be on interfaces, not on the procedure details. Interface must be kept as stable and generic as possible so to avoid changes also if the internal module structure is changing.
    - C++, exploiting the usage of *data abstraction, provides* facilities to create generic interfaces that can have actual different implementations according to the type used.

Exercise: User type Progr.

- So we can use user defined types:

1. Define a type that behaves like a floating number but raise an exception if is not possible to evaluate the square root and such a function is applied to it: *user defined type Squarable;*
2. Define a type that behaves like a floating number but raise an exception if is storing a zero and a division operation is applied to it: *user defined type Denominator*;
3. Define a namespace Haron and the module that implement the square root algorithm on the *Squarable type* according to Haron's method: *result Haron::Sqrt(Squarable value)*;
4. Define a namespace Taylor and the module that implement the square root algorithm on the *Squarable type* according Taylor's series method: *result Taylor::Sqrt(Squarable value)*;
5. Define a module to calculate the discriminant returning a *Squarable*: *Squarable result Discriminant(bvalue, avalue,cvalue)*;
6. Define a module to calculate the first solution using a *Denominator*: *result X1(Denominator avalue, svalue)*;
7. Define a module to calculate the second solution using a *Denominator*: *result X2(Denominator avalue, svalue)*;
8. Define the variable a of type *Denominator*;
9. Define the variable b ;
10. Define the variable c ;
11. Define the variable x_1;
12. Define the variable x_2;
13. Assign a value to a ;
14. Assign a value to b ;
15. Assign a value to c ;
16. Assign the first result to x_1 : x_1 = *X1(b , Haron::Sqrt(Discriminant(b , a , c))* ;
17. Assign the second result to x_2 : x_2 = *X2(b , Haron::Sqrt(Discriminant(b , a , c))* ;

What is C++?

- What is C++ ?
  - **Object-Oriented Programming**, that is: *Decide which classe you want; provide a full set of operations for each class; make commonality explicit by using inheritance.*
    - If abstract data can be defined also virtual class (abstract) can be. This class do not correspond to any actual class but define an interface that can be used by an user without specification of the actual type.
    - If I have to write code to manage *Inmate*, *Physician*, *Paramedic* and *Employees*, to count the accesses into a building, can I write a code that deals only with *Persons* ?

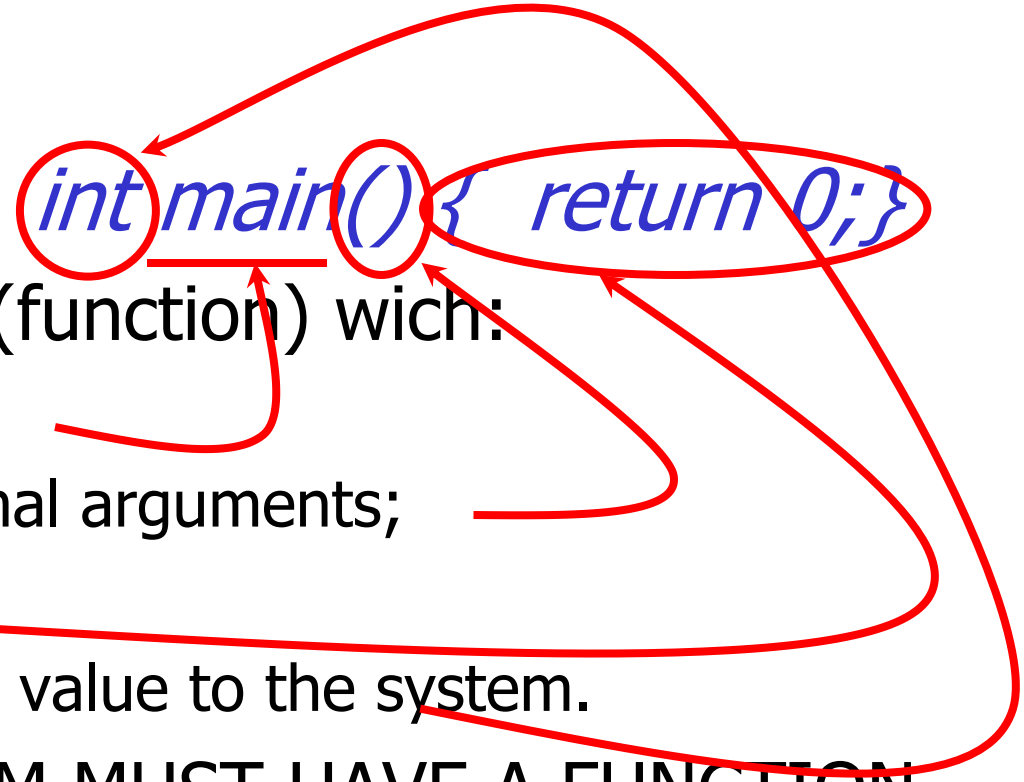
Exercise: OO Progr.

- So we can use user defined types:

1. Define a type that behaves like a floating number but raise an exception if is not possible to evaluate the square root and such a function is applied to it: *user defined type Squarable;*
2. Define a type that behaves like a floating number but raise an exception if is storing a zero and a division operation is applied to it: *user defined type Denominator*;
3. Define a namespace Haron and the module that implement the square root algorithm on the *Squarable type* according to Haron's method: *result Haron::Sqrt(Squarable value)*;
4. Define a namespace Taylor and the module that implement the square root algorithm on the *Squarable type* according Taylor's series method: *result Taylor::Sqrt(Squarable value)*;
5. Define a type that describe a quadratic equation: *user type Quadratic*
   - Stores a Denominator value
   - Stores two floating number values for the coeeficients
   - Implements the Discriminant calculation on the above listed values
   - Implements the caluclation of the two results
6. Define the variable QadEq of type *Quadratic*;
7. Assing value to *QuadEq*::a ;
8. Assing value to *QuadEq*::b;
9. Assing value to *QuadEq*::c;
10. Define the variable x_1;
11. Define the variable x_2;
12. Choose a the namespace for the square root algorithm;
13. Assign the first result to x_1 : x_1 = *QuadEq::X1*;
14. Assign the second result to x_2 : x_2 = *QuadEq::X2* ;

Base grammar

- The minimal code: *int main(){ return 0;}*
  - Defines a module (function) wich:
    - Is called: *main;*
    - Does have no formal arguments;
    - Does nothing;
    - Returns an integer value to the system.
  - ALL C++ PROGRAM MUST HAVE A FUNCTION CALLED: *main()*
  - The executable starts executing such a function.



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.

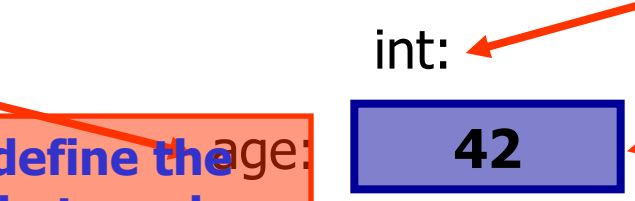
```
int main() {
  {
      // empty block
  }
  return 0;
  }
```

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*.

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

- 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.

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;

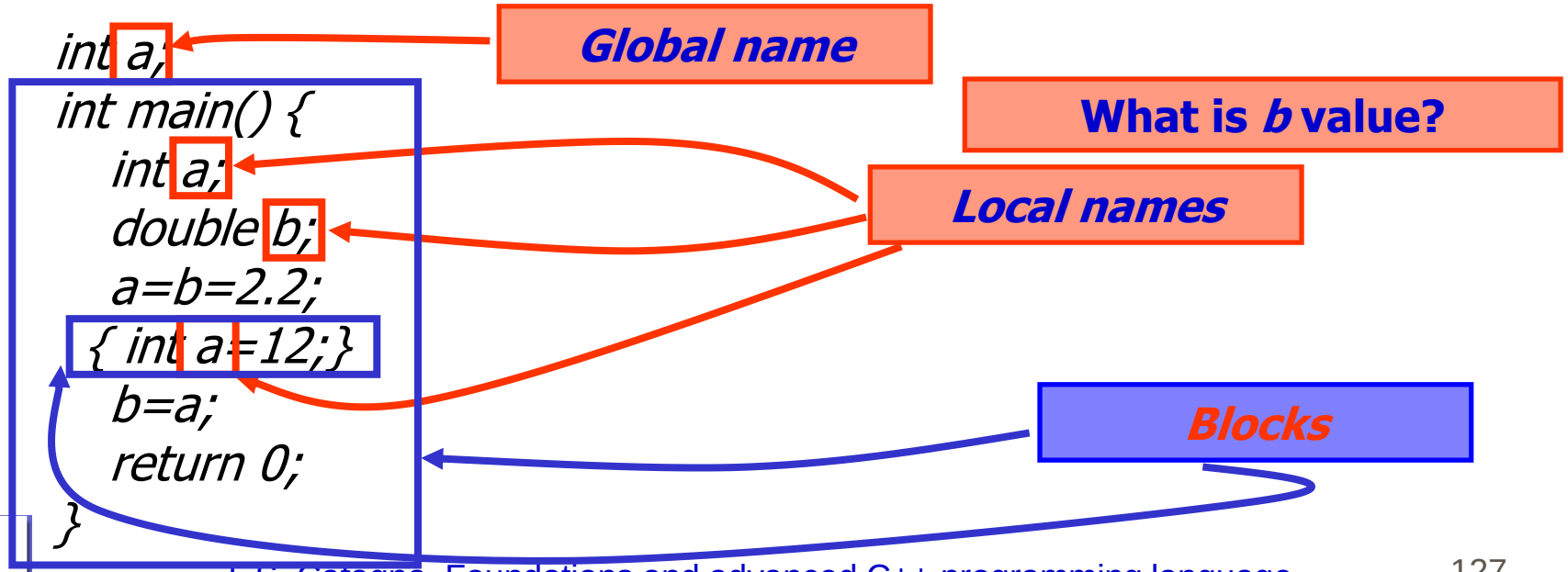
  - Another notation can be used, the functional notation:

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.

Grammatica di base: Tipi

- Tipi di Base:
  - Esiste anche un tipo particolare: *void*
  - E' vuoto, non possono esistere tipi vuoti, ma specifica che una funzione non ritorna un valore o ritorna un puntatore ad un oggetto di tipo sconosciuto:

```
void a,b,c; // Errore ! Non esistono oggetti di tipo
                //vuoto
void f();   // Funzione che non ritorna un valore
void *g(); // Funzione che ritorna un puntatore ad un
                // oggetto di tipi sconosciuto
```

Grammatica di base: Tipi

- Tipi definiti dall'utente:
  - Una dichiarazione preceduta dalla parola chiave: *typedef* , dichiara un nuovo nome per un tipo piuttosto che per una nuova variabile di un dato tipo:

    typedef int Fscaf;

    typedef double My_double;
  - I *typedefs* sono sinonimi di altri tipi piuttosto che nuovi tipi.
  - Vengono usati per ridurre la scrittura di tipi troppo lunghi o per limitare il riferimento a tipi specifici in un solo posto del codice

Grammatica di base: Tipi

- Tra I tipi definiti dall'utente ci sono gli enumeratori: *enum.*
  - Contiene un set di valori specificati dall'utente. Essenzialmente un intero in un range;

```
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;
   }

}
```

Grammatica di base: Enum

- Tra I tipi definiti dall'utente ci sono gli enumeratori: *enum.*
  - Se non specificati I valori vengono assegnati in ordine crescente partendo da 0:
    enum numbers {ZERO,ONE,TWO};
  - Posso definire un range:
    enum a1_5 {a=1,b=5};
    a1_5 quattro=4; a1_5 tre=a1_5(3);

Grammatica di Base: *Declaration*

- In conclusione una dichiarazione contiene:
  - *specifier base type declarator initizializer*;
    - *specifier*: è opzionale, specifica degli attributi per il tipo. Per es.: *virtual, extern, const* .
    - *base type*: è il tipo. Per es: int, long, unsigned, float, Pere.
    - *declarator*: è formato da una lista di nomi e da un operatore su ogni nome. Gli operatori possono essere prefissi o postfissi:
      - *** prefisso — **Operatore: puntatore a**
      - ** const* prefisso — **Operatore: puntatore costante a**
      - *[ ]* postfisso — **Operatore: array**
      - *( )* postfisso — **Operatore: funzione**
    - *initizializer* : è opzionale, specializzano il nome (entità) a cui si riferisce.
  - Una dichiarazione termina sempre con *;* a meno che non si tratti di definizione di *function* o *namespace*.

Grammatica di base: Costanti

- Tra gli *specifier* useremo spesso la parola chiave: *const* . Questo *specifier* specifica l'attributo di non modificabilità per il tipo.
- In questo modo è possibile dichiarare delle variabili costanti, ovvero non modificabili:

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

Grammatica di base: Arrays

- Array sono aggregati omogenei ed ordinati dello stesso tipo.
- Gli array di un tipo T si dichiarano usando l'operatore [] (postfisso) nel *declarator*, specificandone la dimensione:

  int a[5]; double a[5];
- Gli array possono essere inizializzati ad un elenco di valori , separati da virgole, ed inclusi tra *{}*:

  int a[5]={1,2,3,4,5};
  double b[6]={1.,2.,3.,4.}
  (equivalente a: *double b[6]={1.,2.,3.,0.,0.,0.}*)
- La dimensione può essere omessa se viene inizializzato al numero di elementi:

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

Grammatica di base: Arrays

- È possibile accedere in maniera casuale (*random access*) agli elementi di un array, usando l'operatore di accesso (*subscription*) *[]* .
- Attenzione!!! Gli elementi sono indicizati partendo da 0.

```
int a[5];
a[0]=1; // assegno il literal 1 al primo elemento dell'array
a[4]=5; // assegno il literal 5 all'ultimo elemento dell'array
```

- Attenzione!!! Non c'è controllo del limite (boundary check).

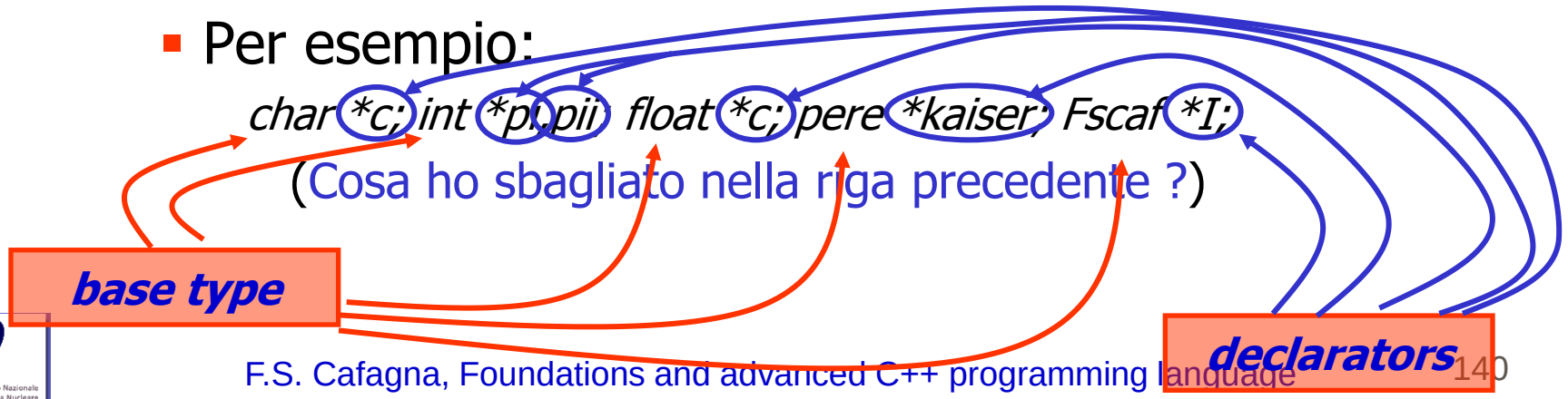
```
int a[5];
a[5]=6; // ERRORE!!! Ho superato il numero di elementi dichiarato
        // sbagliato ma legale !!!!
```

Grammatica di base: Pointers

- I puntatori:
  - Per ogni tipo T, esiste il tipo T* che rappresenta il puntatore ad un tipo T.
  - Una variabile di tipo T* può contenere l'address di un oggetto di tipo T.
  - Per dichiarare un nome di tipo "puntatore a …", usiamo il prefisso * nel declarator.
    - Per esempio:

*char *c; int *p, pii; float *c; pere *kaiser; Fscaf *I;*

(Cosa ho sbagliato nella riga precedente ?)



Grammatica di base: Pointers

- Un puntatore *dereferenzia (dereferencing)*: si riferisce ad un oggetto puntato dal valore che il puntatore contiene. L'operatore di dereferenziazione è l'operatore: *
- Mappa direttamente la memoria della macchina.
- L'operatore che dato un nome (variabile) ritorna il suo address di memoria è l'operatore: &

  *int a=2; int *pa; pa=&a; *pa=3;*
- Gli operatori * ed &, sono unari, ovvero agiscono su un solo oggetto.

Grammatica di base: Pointers

- Sui puntatori è possibile applicare gli operatori aritmetici: *+, - , ++, --* .
- Attenzione!!! La somma è definita tra un puntatore ed una costante intera, non tra puntatori:

  *float c,d; float *pf=&c,*pf2=&d; pf=pf+3; pf2=pf2+pf; (errore !!!!!)*
- Attenzione alla sottrazione!!!
- Attenzione alle costanti: **const*, *declarator operator* per puntatore costante, *const** non e' un declarator permesso viene considerato parte del base type:

  *const int *pi; // puntatore ad un intero costante: pi+1 permesso*

  *int const *pi; // puntatore ad un intero costante: pi+1permesso*

  *int *const pi; // puntatore costante: pi+1non permesso*

Grammatica di base : Pointers

- L'uso di puntatori ad oggetti costanti è utile nelle definizioni delle funzioni per evitare che le funzioni modifichino l'oggetto puntato:
  - *char * strcpy(char *p, const char *q); // *q non si può modificare*
- Puntatore a *void:*
  - Un puntatore di ogni tipo può essere assegnato ad una variabile di tipo *void*: int *pi; void* pv=pi;*
  - *void** può essere assegnato ad un *void** e lo si può comparare ad un altro *void**
  - Ovviamente non posso deferenziare un *void*: void *pv; *pv=123; (non permesso !!!!)*
  - *Si usa per passare puntatori o ricevere puntatori da funzioni non potendo fare assunzioni sul tipo passato o ritornato*

Grammatica di base: Pointers & Arrays

- Il nome di un array può essere usato come puntatore al suo elemento iniziale:

  int a[3]={1,2,3}; int pi=a; pi=&a[0];*

- Quindi è possibile accedere ad un elemento di un array sia con puntatori che con l'operatore *[]* :

  int v[]={1,2,3,4,5,6,0};
  for (int i=0; v[i]!=0; i++) do_nothing(v[i]);

  int v[]={1,2,3,4,5,6,0};
  *for (int *i=v; *i!=0; i++) do_nothing(*i);*

- Array di char* si possono inizializzare a stringhe di lettere: *char *p="Test";*
- Attenzione: *char *p="Testo"; p[4]='i';* e' un errore poichè il primo *statement* ha l'effetto di assegnare una costante a p; il risultato del secondo *statement*, potrebbe essere indefinito. Usare: *char p[]="Testo"; p[4]='i';*

Grammatica di base: Casting

- Casting, ovvero: *ExplicitType Conversion*
  - Dichiarato un nome potrò usarlo per operazioni definite sul tipo del nome.
  - Ma se io voglio usarlo per operazioni definite per un altro tipo o se voglio assegnarlo ad un nome di un tipo diverso?
  - Si usa il casting. Si dichiara che il nome va "visto" in memoria come un tipo diverso.
  - Esistono vari tipi di casting: *static_cast*, *reinterpret_cast*, *dynamic_cast* e *const_cast* .
  - La sintassi per l'utilizzo è la stessa:

categoria_di_cast< tipo_verso_cui_convertire >(tipo_da_convertire)

```
int b;
long a;
a= static_cast<long>(b);
```

Grammatica di base: Casting

- Casting, ovvero: *Explicit Type Conversion*
  - *static_cast*: converte tipi in relazione tra loro come puntatori nella stessa gerarchia di classe, interi in enumeratori, floating in interi.
  - *reinterpret_cast*:converte tipi non in relazione tra loro come un intero in un puntatore o puntatori non appartenenti alla stessa gerarchia di classe.
  - *dynamic_cast* : permette una conversione controllata *run-time.*
  - *const_cast*: rimove i qualificatori *const* e *volatile.*
  - Si può sempre usare il casting in stile "C". Che permette qualunque combinazione degli *static*, *reinterpret* e *const_cast*s:

```
Type_di_b b;
Type_di_a a;
a= (Type_di_a) b;
```

Grammatica di base: Operatori

- **Operatori aritmetici (+, -, *, /, %)**
  - *+* addizione
  - *-* sottrazione
  - *** moltiplicazione
  - */* divisione
  - *%* modulo
- *=*, è l'operatore di assegnazione.
- *>>*, "metti in" (se applicati a *stream*).
- *<<*, "ricevi da" (se applicato a *stream*).

Grammatica di base: Operatori

- Operatori logici(&, |, ^, ~,<<,>>)
  - *&* AND logico,
    - int a=0x12,b=0xf8; int c=a&b; → c vale 0x10
  - */* OR logico
    - int a=0x12,b=0xf8; int c=a|b; → c vale 0xfa
  - *^* XOR logico
    - int a=0x12,b=0xf8; int c=a^b; → c vale 0xea
  - *~* NOT logico
  - *>>* right bit shift (i bit si contano da 0)
    - int a=2; int c=a>>1; → c vale 1
  - *<<* left bit shift (i bit si contano da 0)
    - int a=1; int c=a<<1; → c vale 2

Grammatica di base: Operatori

- **Operatori di comparazione (==, !=, >, <, >=, <=)**
  - *==* Uguale a;
  - *!=* diverso da;
  - *>* maggiore di;
  - *<* minore di;
  - *>=* maggiore uguale a;
  - *<=* minore uguale a;
- Ritornano un booleano dopo l'operazione:

  bool test=(a != b);

Grammatica di base: Operatori

- Gli operatori aritmetici e binari si possono combinare:
  - *++, --, +=, -=, *=, %=, /=, &=, |=, ^=, <<=, >>=*
  - Preincremento: *++a;*

    a=a+1;
  - Postincremento: *a++;*

    (type_of_a t=a, a+=1, return t)

    int a=2,b=0; b=a++;

Quale è il valore di b ?

- Tutti gli operatori tranne gli unari e l'assegnazione sono associativi a sinistra:
  - a=b=c → a=(b=c)
  - a+b+c → (a+b)+c
- Gli operatori unari agiscono su un solo elemento.

Grammatica di base: Operatori

- Non ci sono regole all'ordine di valutazione delle espressioni
- Gli operatori: *, () && // ,* hanno il loro left-hand-operand valutato prima del right-hand:

```
bool t= ( a%100 ) && ( a+b );
```