13-06-2017

Base Grammar: Pointers & Arrays

- An array name can be used a pointer to its initial element:

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

- So an element can be accessed either with a pointer or with the subscription operator *[]* :

  ```
  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);
  ```

- char* arrays can be initialized to a string: *char *p="Test";*
- Be careful: *char *p="Testo"; p[4]='i';* is an error because the first statement assigns a constant to p; as a result the second statement behavior could be undefined. Explicit declaration of array size must be used:

  char p[]="Testo"; p[4]='i';

Base Grammar: Casting

- Casting, that is: *ExplicitType Conversion*
  - Once declares, a name can be used only for the set of operations defined for that given type.
  - What about using it in a set of operations defined for another type or assigning is to a name of different type?
  - A casting operation can be used, instructing the compiler to "view" in memory, the name as a different type.
  - Different type of casting are supported: *static_cast*, *reinterpret_cast*, *dynamic_cast* e *const_cast* .
  - Casting syntax is common:

```
kind_of_cast< type_to_ whom_convert >(type_to_be_converted)
int b;
long a;
a= static_cast<long>(b);
```

Base Grammar: Casting

- Casting, that is: *Explicit Type Conversion*
  - *static_cast*: converts related types like pointers in the same class hierarchy, or integer in enumerators, or floating in integer, or floating to double.
  - *reinterpret_cast*: converts not related types like an integer to a pointer, or pointers not in the same class hierarchy.
  - *dynamic_cast*: implements a *run-time* check of the conversion.
  - *const_cast*: removes qualifies *const* and *volatile.*
  - Style "C" casting is always allowed. Compiler will chose the best between *static*, *reinterpret* and *const_cast*s:

```
Type_di_b b;
Type_di_a a;
a= (Type_di_a) b; // C style
```

Base Grammar: Operators

- **Arithmetical operators (+, -, *, /, %)**
  - *+* sum
  - *-* subtraction
  - *** multiplication
  - */* division
  - *%* modulo
- *=*, is the assignment operator.
- *>>*, “stream to”.
- *<<*, “stream from”.

Base Grammar: Operators

- Logical operators (&, |, ^, ~,<<,>>)
  - *&* logical AND,
    - *int a=0x12,b=0xf8; int c=a&b; → c equals to 0x10*
  - */* logical OR,
    - *int a=0x12,b=0xf8; int c=a|b; → c equals to 0xfa*
  - *^* logical XOR,
    - *int a=0x12,b=0xf8; int c=a^b; → c equals to 0xea*
  - *~* logical NOT,
  - *>>* right bit shift (bit counting from 0)
    - *int a=2; int c=a>>1; → c equals to 1*
  - *<<* left bit shift (bit counting from 0)
    - *int a=1; int c=a<<1; → c equals to 2*

Base Grammar: Operators

- **Comparison operators (==, !=, >, <, >=, <=)**
  - *==* equal to;
  - *!=* different from;
  - *>* greater than;
  - *<* less than;
  - *>=* greater equal than;
  - *<=* less equal than;
- They returns a boolean once operation is performed:

bool test=(a != b);

Base Grammar: Operators

- Arithmetical and binary operators can be combined into an unary one:
  - *++, --, +=, -=, *=, %=, /=, &=, |=, ^=, <<=, >>=*
  - Preincrement: *++a;*
    a=a+1;
  - Postincrement: *a++;*
    (type_of_a t=a, a+=1, return t)
    int a=2,b=0; b=a++;

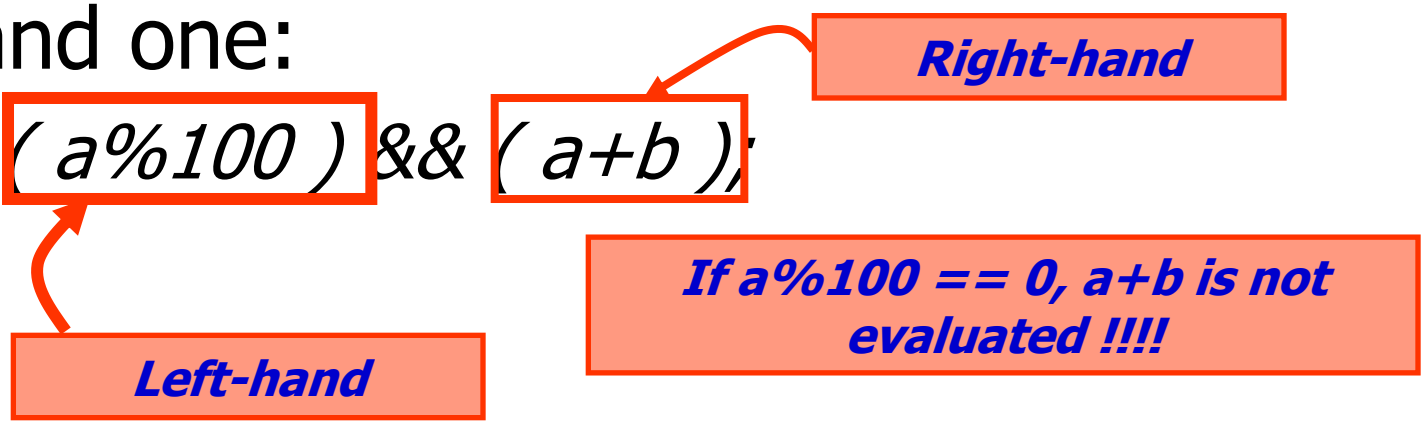
How much is b value?

- All operators but unary ones and assignment are left associative:
  - a=b=c → a=(b=c)
  - a+b+c → (a+b)+c
- Unary operators act on a single element.

Base Grammar: Operators

- There are no rules to the evaluation order of an expression.
- Operators: *, () && // ,* have their left-hand-operator evaluate before than the right-hand one:

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



Base Grammar: Pointers

- **Before we start, a step back: how are floating point number stored into a computer memory?**
  - The IEEE standard requires, for a single precision floating point value, a 32 bit word. From left to the right, the first bit, bit 0, is the sign bit: S; the 8 following bits, are the exponent, E, normalized to 128, the rest of the bits represent the so called fraction F (every bit,n, represents the power of 2^(-n) must be added to 1.)
  - Value stored, is calculated according to the value of its above mentioned components:
  - E=255 && F !=0: V=NaN ("Not a number");
  - E=255 && F =0 && S=1: V=-∞;
  - E=255 && F =0 && S=0: V=∞;
  - (0<E<255): V=(-1)**S * 2 ** (E-127) * (1.F); [Normalized values]
  - E=0 && F!=0: V=(-1)**S * 2 ** (-126) * (0.F); [Not normalized values]
  - E=0 && F=0 && S=1: V=-0;
  - E=0 && F=0 && S=0: V=0;
  - Esempi:
    - 0 00000000 00000000000000000000000 = 0
    - 1 00000000 00000000000000000000000 = -0
    - 0 11111111 00000000000000000000000 = ∞
    - 1 11111111 00000000000000000000000 = -∞
    - 0 11111111 00000100000000000000000 = NaN
    - 1 11111111 00100010001001010101010 = NaN
    - 0 10000000 00000000000000000000000 = +1 * 2**(128-127) * 1.0 = 2
    - 0 10000001 10100000000000000000000 = +1 * 2**(129-127) * 1.101 =(2**2)*(1+1/2+1/8)=6.5
    - 1 10000001 10100000000000000000000 = -1 * 2**(129-127) * 1.101 = -6.5
    - 0 00000001 00000000000000000000000 = +1 * 2**(1-127) * 1.0 = 2**(-126)
    - 0 00000000 10000000000000000000000 = +1 * 2**(-126) * 0.1 = 2**(-127)
    - 0 00000000 00000000000000000000001 = +1 * 2**(-126) * 0.00000000000000000000001 = 2**(-149) (Numero più piccolo rappresentabile)

S	EEEEEEEE	FFFFFFFFFFFFFFFFFFFFFFF
0	1 8	9 31

Base Grammar: *Pointers*

```
int main(){
  int a,b[2];
  float c,d;
  int *pi,*pii;
  int ** pb;
  int *pa[2];
  float *pf;
  a=2;
  pi=&a;
  pii=pi;
  pi++;
  c=static_cast<float>(*pi);
  d=*reinterpret_cast<float *>(pi);
  c=static_cast<float>(*pii);
  d=*reinterpret_cast<float *>(pii);
  b[0]=5;
  b[1]=7;
  pi=b;
  pii=&b[1];
  pb=&pi;
  pa[0]=&a;
  pa[1]=&b[1];
  b[0]=5;
  a=b[0];
  pi=&a;
  pii=b;
  if(pi==pii) {
    c = 1.;
  } else {
    c = 2.;
  }
  if(*pi==*pii) {
    d = 1.;
  } else {
    d = 2.;
  }
  return 0;
}
```

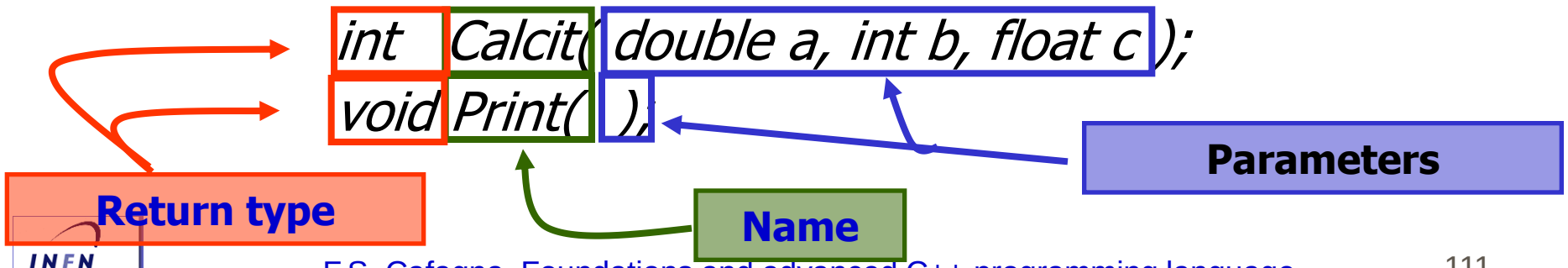
test_pointer.cpp

- An example: test_pointer.cpp

https://owncloud.ba.infn.it/public.php?service=files&t=9253dcd8a30a1b4331d6bc9023f6d598

Functions

- **Functions**
  - The best way to let your code execute a calculation, is to "call" a module that does it. This module is called function.
  - A function has a name that must be declared before using it.
  - A function name can be used as soon as it is declared.
  - In a function declaration must contain:
    - A type assigned to the value returned, if not use void.
    - The function name.
    - A list of formal arguments, often called parameters, needed for the calculation; arguments are declared in a comma separated list.
    - For every argument its type must be specified.



Functions

Functions

- In a function declaration the parameter names can be missing.

int test(double , int , float);

Parameters with no name, just the type

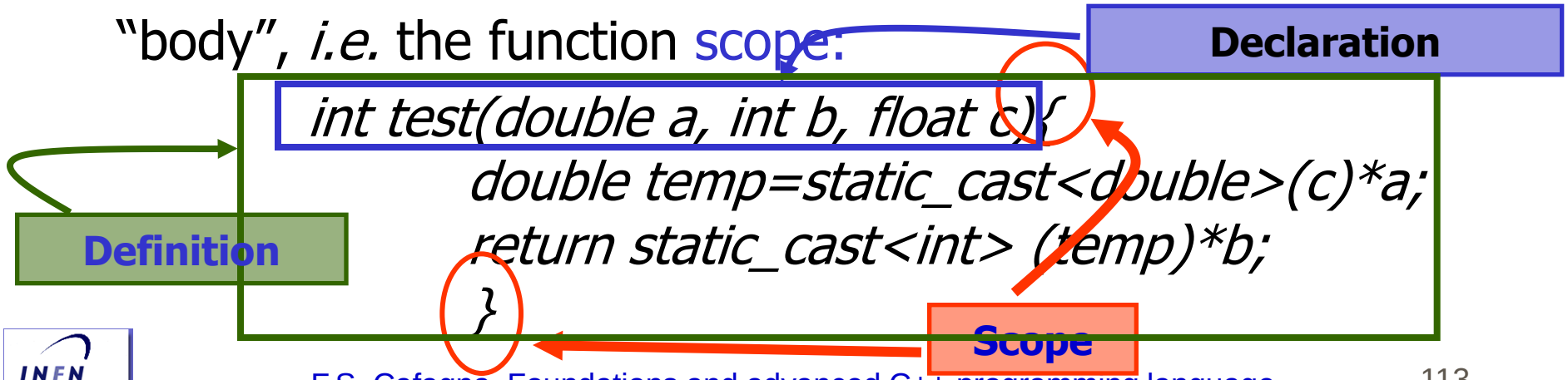
- A function returning: *void*, doesn't return any value.
- The statement: *return some_name;*, define an unmamed variable of the same type of the one used in the function declaration and "returns" it to the user.
- The semantics of argument passing are identical to the semantics of initialization. The type of an actual argument is checked against the one of the corresponding parameter and all standard an user-defined type conversions are performed.

```
// Test used with actual values
int a = test(2.5, 2, 3.0f );
```

Functions

Functions

- In a declaration we specify a function name and interface, *i.e.* return types and parameters.
- But we need functions doing something. How we specify this?
- We have to define the function. Somewhere we have to write the actual code that specify what it does.
- Definition can be done only once.
- A function definition is a declaration that include the function "body", *i.e.* the function scope:



Header files

- Declaration and definition of a function can reside in different files.

```
int test(double, int, float);

int main(){
...
        int bb=test(dd,ii,ff);
...
       return 0;
}
```

main.cpp

```
int test(double a, int b, float c){
 double temp=
     static_cast<double>(c)*a;
 return static_cast<int> (temp)*b;
}
```

test.cpp

Compilation

1. g++ -c test.cpp
2. g++ -c main.cpp
3. g++ -o my_prog main.o test.o

Link

Header files

- Usually declarations are grouped into files, called *header*. (They are usually placed at the very beginning of a code, but they can be placed in any place but before using the name …)
- The ususal extension for them is: .h, but can be any.

```
#include "my_decl.h"

int main(){
        int bb=3;
        return fac(3);
}
```

main.cpp

```
int test( double, int, float );
```

my_decl.h

```
int test(double a, int b, float c){
 double temp=
     static_cast<double>(c)*a;
 return static_cast<int> (temp)*b;
}
```

test.cpp

Header files

- *#include* is a preprocessor macro. It is not a C++ expression.
- Instruct the preprocessor to include the file: *my_decl.h*, content into the code *main.cpp*, starting from the line where the macro is placed.
- Included file name can be delimited by: *< … > , or " … "*
  - *< … >* , in this case included files are searched for in the "standard" system directories, i.e. those declared in the configuration (remember the *g++ -v* printout?).
  - *"…"*, in this case included files are searched for in the directory where the compiler has been launched or those specified using the option: *-I*

g++ -I/home/cafagna/includes –I../inc –I/my_header_dir …

Header files

- In general a header is a file that holds declarations of functions, types, constants, and other program components. A header gives you access to functions, types, etc. that you want to use in your programs.
- The actual functions, types, etc. are defined in other source code files
- They are often distributed as part of libraries or operating system.

Selection statements

- What is the bool type needed for?
- And the comparison operators?
- I can use them to test an expression and select between alternatives using a *selection statement: if, switch, case, ?*
- An *if* statement select between two alternatives:
  - *if (condition) statement*
  - *if (condition) statement else statement*

Selection statements

```
int a=2;
int b=3;
if(a==b) {
      int c=a;
      c++;
      Print(c);
          }
if(c!=b){
    a=b;
} else {
    a=c;
}
```

Where is the error?

Selection statements

- if *statement* tests if the condition is true or different from zero. So any expression (also using pointers) can be used in the condition:

```
int x=2;
if(x) {
//  X diverso da zero
} else {
//  X uguale a zero
 }
```

- if *statement* verifies if a pointer is different from NULL pointer:

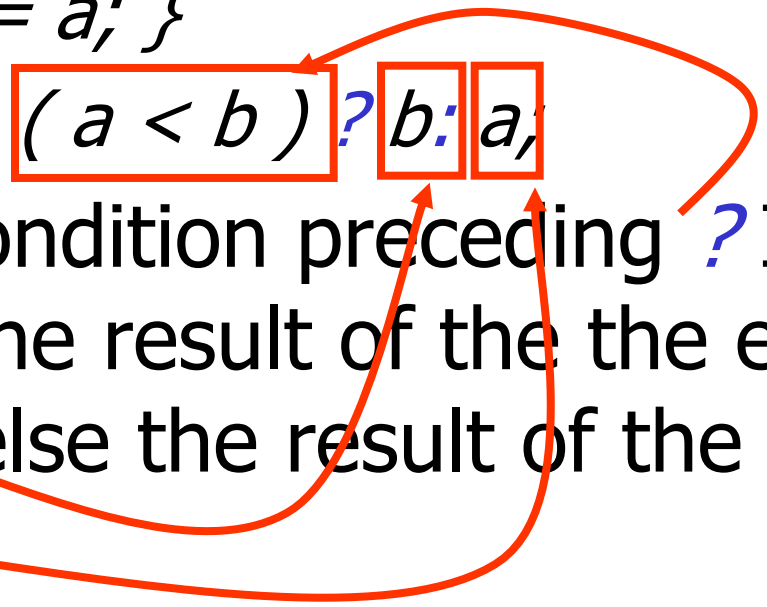
```
int *p;
if(p) {
// … true
} else {
// … false
 }
```

Which statement will be executed ?

Selection statements

- Assignment in a if statement can be substituted by: ? :

```
if (a <b)
{max = b; }
else
{max = a; }
max = ( a < b ) ? b: a;
```



- If the condition preceding ? Is true then assign the result of the the expression after ? else the result of the expression after :

Selection statements

- Often you have to choose between several alternatives not just two. You can do that using the statement: *switch*
- The *switch* statement selects along several alternatives comparing a value against integer constants:

```
If (val==1)
{ f();
}else {
   if (val==2){
     g();
   }else {
     h();
}
}
switch(val){
case 1:
   f();
   break;
case 2:
   g();
   break;
default:
   h();
}
```

Selection statements

- The *switch* statement argument must be of an int type.
- The argument value is compared to each constant declared with a *case* label.
- A label is terminated by *:* .
- A *case* label must contain the integer constant:

  case int_constant_name_or_litteral:
- If the value matches the constant in a *case* label , the statement(s) for that case is chosen.
- If the value doesn't match any of *case* labels, the statement identified by the default label is chosen.
- Note that no other comparison is done.

```
int a=0, b=0;
switch (a) {
  case 0:
          b++;
  case 1:
          b++;
  case 2:
          b++;
}
```

What is the value of b?

Selection statements

- There is the break statement. This statement stop a scope execution.
- Each case should be terminated by a break. If not, you will drop through the next case.
- A break statement is also useful to terminate a loop.
- *Loop* ?