

Remind of C/C++ basic elements

P. Ronchese

Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

The “main” function

The execution of a C++ program starts with the `main` function

- all C++ programs have one and only one `main` function
- it executes operations, calls other functions, creates objects...
- instructions are terminated by a semicolon ;
- it returns an integer (typically 0 to indicate no errors)

```
...
int main() {
    ...
    return 0;
}
```

Programs are (usually) splitted in several files:
“translation units”

```
c++ -Wall -o exec_name file_list
```

Data types

- signed integers: `int`, `short`, `long`, `long long`
- unsigned integers: `unsigned int`, ...
- enumerators: `enum`
- floating point: `float`, `double`, `long double`
- characters: `char`
- C-strings: `char*/char[]` (terminated by `'\0'`)
- logicals: `bool` (or also `int`)

All variables must be “declared” before their usage

- names are case-sensitive: `Energy` is not the same as `energy`
- they can be initialized: `int i=3;`
- they can be made unmodifiable: `const float x=3.14;`
- they are (usually) visible inside the “scope” (`{}`) where they're declared

Operations

- “unary” operators: $-x$, $j--$, $++l$, ...
- “binary” operators: $a+b$, $x-y$, $i*j$, $p<q$, ...
- “ternary” operators: $x?a:b$
- other operators:
 - `()` : function call
 - `new`, `delete` : create and destroy
 - ...
 - `sizeof(...)` : variable size (in bytes)

There's a defined precedence and associativity table:

- e.g. in $a+b*c$ the multiplications is executed before the sum
- but it can be overridden by using parentheses: $(a+b)*c$.

Flux control

- Conditional statements:
 - `if (expr) stat:`
it evaluates `expr`, if it's true it executes `stat`.
- Loop statements:
 - `for (exp1;exp2;exp3) stat :`
it evaluates `exp1`,
then it evaluates `exp2`,
if it's true it executes `stat` and then `exp3`,
then it evaluates `exp2` again and so on.
 - `while (expr) stat:`
it evaluates `expr`, if it's true it executes `stat`,
it evaluates `expr` again and so on.
 - `do stat while (expr):`
it executes `stat` and then it evaluates `expr`,
if it's true it executes `stat` again and so on.
 - `continue` and `break` instructions to alter the cycle
- Choice statements:
 - `switch (int_expr) { list_of_cases }`

Comments

Comments can (must) be included in programs!

The compiler ignores anything that:

- follows a `//` until the end of the line,
- is comprised between a `/*` and a `*/`

```
...
int main() {
    ... // an one-line comment
    /* a comment
       written over
       several lines */
    ...
    return 0;
}
```

Mathematical operators

Decreasing priority:

- ++ , -- : pre/post increment and decrement
- * , / , % : multiplication, division, modulus
- + , - : addition and subtraction
- < , > , <= , >= , == , != : relations
- = , *= , /= , %= , += , -= : assignment

Care needed!

- The result of the division between integers is an integer
- Equality and assignment operators are similar but different

```
int i=4, j=5;
float x=i/j; // x=0
float y=i*1.0/j;
           // y=0.8
```

```
int i=7, j=9, k=0;
if(i=j)k+=1; // true
if(i==j)k+=2; // false
           // k=1
```

Logical and bitwise operators

Logical - Decreasing priority:

- `&` : bitwise and
- `^` : bitwise exclusive or
- `|` : bitwise or
- `&&` : logical and
- `||` : logical or
- `&=` , `^=` , `|=` : bitwise assignment

Bitwise - Decreasing priority:

- `<<` , `>>` : bitwise shift left/right
- `<<=` , `>>=` : bitwise shift assignment

Assignment operators

Assignment operators are also expressions

- The value of the expression is given by the left-side after the assignment
- Assignments can be used inside complex operations

```
int i=3;
int j;
float x=5.7/(j=i);
// now both "i and "j" are 3
// and "x" is 5.7/3=1.9
```

Type conversions

Variables are converted to other types implicitly when needed, but some control is sometime necessary (e.g. `x=i*1.0/j`):
“type cast”

Explicit conversions between an `int i` and a `float x`:

- C-style casts: `i=(int)x` or `i=int(x)`
 - not always clear what they do
 - difficult to find across the code
- C++-style casts: `i=static_cast<int>(x)`

C++ has 3 other types of casts, they will be seen later

Type synonyms

An existing type (e.g. `float`) can be given an additional name with a `typedef` declaration

```
typedef float number;  
number x=5.1;  
number y=6.7;  
number z=x+y;  
std::cout << z << std::endl;
```

- A set of variables can be declared with a common type that can be changed by modifying just one line.
- Short names can be defined for complex types (to be seen later).

User-defined functions

Blocks of code can be isolated into “functions”:

- a function takes a list of “arguments”,
- a function returns one value, or none (“void”),
- a function must be declared before being used.

```
int f(int x, float y);  
int main() {  
    int i=2;  
    float z=3.4;  
    int j=f(i, z);  
    return 0;  
}
```

A function can be “defined” after being used, or even in another “translation unit” (i.e. another file): only the declaration must be present before the usage

Recursion

C/C++ allow a function to call itself: “recursive” calls

- At each call all the function local variables are created and initialized.
- Some condition must occur for the function to return without calling itself.
- Example: function to compute $n!$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)!n & \text{if } n > 0 \end{cases}$$

```
unsigned int fact(unsigned int n) {  
    if(n) return n*fact(n-1);  
    return 1;  
}
```

inline functions

By declaring a function `inline` the compiler is instructed to replicate the code across the program (if possible):

- there's no function call/return overhead,
- larger executables are produced,
- the function declaration is not sufficient.

```
inline int iabs(int i) {  
    return (i>0?i:-i);  
}
```

- Inlining is not possible for recursive functions.
- The program size increase could vanish the benefit.
- The compiler can ignore the indication.

Functions arguments

Function arguments are passed “by value”, i.e. each variable is copied to a local one, inside the function scope:

- the function can modify that copy,
- the function cannot modify the variable in the calling function,
- the copy is destroyed when the function ends.

The return value is copied back to the calling function.

C++ specific: function “overloading”

A function name is “overloaded” when several functions exist with the same name but different argument number and/or types

Default arguments

Default values can be provided:

- they're set in the function declaration,
- if an argument has a default value, all the following ones must have one.

```
int f(int i, int j=1, int k=2);
int main() {
    int n=12;
    int m=23;
    int l=f(n,m);
    // equivalent to
    // int l=f(n,m,2);
}
```

main function arguments

The “main” function has its arguments, too:

- the first one is an integer, equal to the number of “words” in the command line, i.e. the number of arguments plus one,
- the second is an array of C-strings, corresponding to those words.

```
int main(int argc, char* argv[]) {
    std::cout << argv[0]
               << " called with arguments:";

    int iarg;
    for(iarg=1; iarg<argc; ++iarg)
        std::cout << ( iarg > 1 ? ", " : " " )
                  << argv[iarg];

    std::cout << std::endl;
    return 0;
}
```

Predefined functions

Some functions are “predefined”, i.e. they’re already available

Mathematical functions (in `math.h`):

- `sqrt(double)` , `pow(double, double)`
- `sin(double)` , `acos(double)` , ...
- `atan2(double, double)`
- `exp(double)` , `log(double)`
- `fabs(double)` : **abs. value**
- `lround(double)` , `llround(double)` : **rounding**

Add a trailing “l” to use with long doubles

Utility functions (in `stdlib.h`):

- `random()` : **random int**
between 0 and $2^{31} - 1$ (`0x7fffffff`)
- `srand(unsigned int)` : **set the seed for the random generation**
- `exit(int)` : **stop the execution immediately**

Pointers

The “pointer” to a variable (or an object) is its memory address:

- it's declared by adding a “*” to the variable type,
- it can be obtained by mean of the operator “&” ,
- it can be changed to contain the address of another variable (of the same type),
- the variable or object content can be obtained back by mean of the operator “*” ,
- dereferencing an invalid pointer produces a fatal error,
- a null pointer (=0) is always invalid.

```
int i=12;
int j=23;
int* p=&i; // "p" is the address of "i"
std::cout << *p << std::endl;
*p=24; // "i" is now 24
p=&j; // "p" is now the address of "j"
std::cout << i << " " << *p << std::endl;
```

Arrays

Arrays are sets of variables of the same type:

- they're declared by adding a "[N]" (where "N" is an integer) to the variable name,
- their elements are accessed by adding a "[i]", where $0 < i < N-1$.

```
int i[12];
int j;
for(j=0; j<12; ++j) i[j]=2*j;
for(j=0; j<12; ++j) std::cout << j << " "
                        << i[j] << std::endl;
```

- Arrays are pointers.
- `int* p=i;` is the pointer to the first element:
`*p ≡ i[0]` , `*(p+n) ≡ i[n]`
- Strings are arrays of `chars`, with a `'\0'` as last element.

References

A “reference” can be seen as a new name for an existing variable or object:

- it's declared by adding a & to the variable type,
- the referred variable must be specified in the declaration,
- contrary to pointers, it cannot be changed to refer to a different variable.

```
int i=12;
int& j=i; // "j" is a reference to "i"
std::cout << j << std::endl;
j=24; // "i" is now 24
std::cout << i << std::endl;
```

- They're useful in passing or retrieving variables to/from functions.
- Actually they're pointers, with the “*” embedded.

References and pointers to const

A variable can be modified through a pointer or reference to it, unless a “pointer/reference to const” is used.

```
int i=12;
const int* p=&i; // "p" is the address of "i"
std::cout << *p << std::endl;
i=19; // allowed, "i" is not "const"
std::cout << *p << std::endl;
*p=26; // WRONG, "*p" is const
```

Only references to “const” and pointers to “const” can be defined for “const” variables (of course)

References, pointers and functions

Functions pass arguments by value, but:

- arguments and/or result can be pointers, or references,
- the pointers or references are copied, actually,
- the pointed/referred variable or object can be changed,
- arguments passed as const reference cannot be changed.

```
void f( int& i, float* x ) {  
    i*=2;  
    x*=3.4;  
    return;  
}
```

Functions result can also be a pointer or reference, but:

- memory used for local variables is deallocated when the function returns,
- when accessed by the calling function, garbage is found,
- returning pointers and/or references to local variables lead to unpredictable results.

Dynamic memory handling

Pointers are used to allocated/deallocated memory at run time (dynamically):

- variables are created/destroyed with the operators “new” and “delete”,
- dynamic variables are not bound to a scope.

```
int* i = new int(3);  
// "i" is a pointer to an int  
// whose value is "3"  
float* f = new float[12];  
// "f" is an array of 12 float  
...  
delete i;  
// "delete" destroys one single variable  
delete[] f;  
// "delete[]" destroys an array
```

Dynamic memory pitfalls

Special care is required in dynamic variables handling

- Dynamic variables are destroyed only by a “delete” operation, or at execution end:
 - they use unrecoverable memory when all the pointers to them go out of scope (“memory leak”),
 - they must be deleted when no more necessary.
- When a variable has been deleted, the pointer to its memory location is still existing:
 - it cannot be de-referenced (“dangling reference”),
 - a second “delete” operation cannot be performed,
 - care is required with multiple copies of a pointer.
- Unpredictable results are obtained when “delete” is used for arrays or “delete[]” is used for single variables.
- Applying a delete or delete[] to a null pointer (=0) has no effect; a fatal error is produced with any other invalid pointer.

Pointer and reference based type casts

By using pointers and references,
other type casts become possible

Force the modification of a (non-const) variable
through a pointer to const
(unpredictable results for originally-const variables)

```
int i; // "i" is not "const"
const int* p = &i; // "*p" is "const"
*const_cast<int*>(p)=2;
```

Convert the pointer to a type to the pointer to another type,
with no checks

```
float x = 23.45;
float* pf = &x;
int* pi = reinterpret_cast<int*>(pf);
std::cout << *pi << std::endl;
// prints "1102813594"
```

Generic pointers

A “pointer to `void`” can contain the address of any variable or object:

- it's declared as `void*`,
- it cannot be de-referenced,
- it cannot be used as argument for `delete`,
- to be used a `reinterpret_cast` is needed.

```
int i;  
void* p=&i;  
...  
std::cout << *reinterpret_cast<int*>(p)  
           << std::endl;
```

C++-style input-output

- Input and output go through “streams”, `cin` and `cout` are the standard input and output streams.
- Input and output operators are `>>` and `<<` (“bit move”).
- Input and output from/to files go through file streams.

```
#include <iostream>
#include <fstream>
int main() {
    int i;
    std::ifstream file("inputfile");
    file >> i;
    std::cout << i << std::endl;
    ...
}
```

Loop input

- Input stream operator `<<` return value can be tested to be
 - `true` to check for successful reading,
 - `false` to check for end of file.
- End of input from keyboard can be sent with `ctrl-d`.
- To read again after and end-of-input the input stream must be reset by the function `clear()`.

```
#include <iostream>
int main() {
    int i;
    while(std::cin >> i)
        std::cout << i << std::endl;
    std::cin.clear();
    ...
}
```

Output formatting commands

A lot of additional commands to format the output are available, (e.g. to set the number of digits to write for numbers)

```
#include <iostream>
int main()
    float x;
    ...
    std::cout.width(12);
    std::cout.precision(5);
    std::cout << x << std::endl;
    return 0;
}
```

Output formatting objects

The same commands can be sent inside the output streaming

```
#include <iostream>
#include <iomanip>
int main()
    float x;
    ...
    std::cout << std::setw(12)
                << std::setprecision(5)
                << x << std::endl;
    return 0;
}
```

C-style input-output

C++ allows the use of plain-C I/O functions (in `stdio.h`):

- `scanf` and `printf` for input and output to/from standard; the first argument is a string setting the format
- `fscanf` and `fprintf` for input and output to/from file
- `sscanf` and `sprintf` for input and output to/from strings

```
#include <stdio.h>
int main() {
    int i;
    scanf("%d",&i); // pointer to "i" required
    printf("%d\n",i); // "\n" for new line
    return 0;
}
```

C-style formatting

Data type is to be specified

%N.Md	decimal int	%N.Pf	plain float
%N.Mo	octal int	%N.Pe	exponential float
%N.Mx	hexadecimal int	%N.Ls	char string
N	output width	M	number of digits
P	precision	L	string max. length
%ld	long	%qd	long long
%lf	double	negative N : left-justify	

```
printf("=%9.6d=\n", 123);  writes = 000123=
printf("=%9.3f=\n", 1.23); writes = 1.230=
```

I/O with strings

- Read input from strings
- Write output to strings

```
#include <stdio.h>
int main() {
    int i;
    char s[100];
    sprintf(s, "%d\n", i);
    ...
    return 0;
}
```

```
#include <iostream>
#include <sstream>
int main() {
    int i;
    std::stringstream s;
    s.clear();
    s.str("12");
    s >> i;
    ...
    return 0;
}
```

Input by line

Text input can be read “line by line”

A line of input is read by mean of the function “`getline`”, taking as arguments an array of `chars` and the max length (plus eventually the line-terminate character, by default ‘`\n`’)

```
#include <iostream>
int main() {
    int maxLength=1000;
    char* line=new char[maxLength];
    while(std::cin.getline(line,maxLength))
        std::cout << line << std::endl;
    return 0;
}
```

Binary input-output

Binary files contain the variables exactly as they're stored in memory.

Binary I/O is performed with the functions “read” and “write”, taking pointers to `char` (and number of bytes) as arguments

```
#include <iostream>
#include <fstream>
int main() {
    int i;
    std::ifstream file("inputfile",
                      std::ios_base::binary);
    file.read(reinterpret_cast<char*>(&i),
              sizeof(i));
    std::cout << i << std::endl;
    return 0;
}
```