

Storage and linkage specifiers

P. Ronchese

Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

Variables life cycle

C++ variables have a “life cycle”, i.e. they’re created at some point of the execution, and destroyed at another one

- Dynamic variables are created and destroyed when the corresponding “new” and “delete” operators are called
- Other variables are created and destroyed automatically:
 - for that reason they’re sometimes called “automatic variables”
 - by default they’re created when they’re declared,
 - by default they’re destroyed when they go “out of scope”,
 - this behaviour can be modified by adding a “storage specifier”.
- In standards c++98 and c++03 the default behaviour does correspond to adding an “auto” specifier.
- In standard C++11 the “auto” keyword has a different meaning.

Variables linkage

C++ variables are accessible in some parts of a program, and not in others: this is referred to as their “linkage”

- Dynamic variables are visible everywhere, provided a pointer to them is visible
- Other variables are usually visible in the scope where they're declared
 - usually they're declared in a function or a block ({ })
 - they can be declared outside all functions, making them “global”
 - their visibility can be modified by adding a “linkage specifier”

Static storage

- A variable declared in a block is destroyed when the block ends:
 - if the block is executed in a `for` loop, the variable is created and destroyed at each iteration,
 - the value stored in the previous execution is lost.
- A `static` variable is created once and never destroyed until the execution ends:
 - the initialization is performed only once
 - the value stored in the previous execution is preserved

```
int i;
for(i=0;i<10;++i){
    static int j=0;
    std::cout << ++j << std::endl;
}
```

A variable declared inside a function (or block)
has “function scope” (or “block scope”)

Internal linkage

- A variable declared outside all functions has “static storage” and, by default, “external linkage”:
 - it’s created at the execution start (unless it’s in dynamic libraries)
 - it’s by default visible by all the functions,
 - it can be hidden to functions in other translation units by adding a “static” specifier, giving it “internal linkage”

```
int i=0; // visible by all functions
static int j=0; // visible in this
                // translation unit only

void f() {
    std::cout << "f:i=" << ++i << std::endl;
    std::cout << "f:j=" << ++j << std::endl;
}
```

The `static` specifier has a different meaning when applied to variables declared outside all functions.

Extern linkage

A variable, declared outside all functions, with “external linkage” is visible by all functions:

- functions in other translation units can access it, too,
- a declaration is necessary in each translation unit, anyway,
- an "extern" specifier is added to the declaration, in a translation unit, of a variable defined in another translation unit: the name is declared but the variable is NOT created in the memory, because it's created elsewhere,
- a definition, i.e. a declaration without "extern", must be included in **one and only one** translation unit,
- otherwise an “undefined reference” or “multiple definition” error arises (as for functions).

Extern variables declaration and definition

```
int i; // global variable definition
...
int main() {
    i=12;
    g();
}
```

```
extern int i; // defined in another
               // translation unit
...
void g() {
    std::cout << "g:i=" << ++i << std::endl;
}
```

Multiple files

The program is (usually) scattered over multiple files.

- They can be compiled all together to produce an executable:

```
c++ -o exec file1.cc file2.cc
```

- They can be compiled one by one and linked into an executable only later:

```
c++ -c file1.cc
```

```
c++ -c file2.cc
```

```
c++ -o exec file1.o file2.o
```

- `c++ -c` : compile without link

The code contained in one file together with all the included ones is called “translation unit”.

Static libraries

- Source file(s) can be compiled to a “static” library:

```
c++ -c file2.cc
```

```
ar -r libTestS.a file2.o
```

- The library can be linked to the executable:

```
c++ -o exec file1.cc -L. -lTestS
```

- `ar -r` : create a library

- `c++ -Ldir` : look for libraries in directory *dir*

- `c++ -lname` : look for the library *libname.a*

The code is copied to the executable:

- the libraries are **only needed when compiling**,
- the libraries are **not needed at runtime**.

Dynamic libraries

- Source file(s) can be compiled to a “dynamic” library:

```
c++ -fPIC -shared -o libTestD.so file2.cc
c++ -o exec file1.cc -L. -lTestD
```

- `c++ -fPIC -shared` : produce a dynamic library
- `c++ -Ldir` : look for libraries in directory *dir*
- `c++ -lname` : look for the library `libname.{a,so}`

The code is only referred by the executable:

- the **libraries** are needed
both when compiling and at runtime,
- the **path to the library** must be defined
both when compiling and at runtime,
e.g. in the environment variable `LD_LIBRARY_PATH` ,
- the `main` itself can stay inside a library, provided it's unique.

Initialization order rules

The initialization order of global variables or objects is defined only inside the same translation unit:

- global objects/variables defined in the same translation unit are initialized following the definition order,
- global objects/variables defined in different translation units are initialized following in undefined order.

Initialization order pitfalls

Care needed!

- For native variables initialization order is mostly not important, unless the value of a variable is used to initialize another one.
- For more complex objects, to be seen later, any dependence in the initialization of a global object from another global object defined in another translation unit must be avoided. Otherwise, an error called “Static order initialization fiasco” occur.
- Some techniques to avoid the problem will be shown in the following parts of the course.