Environment and scope in C++

Florido Paganelli
Lund University
florido.paganelli@hep.lu.se
Fysikum, Hus A, Room 403
Visiting time: 11:00-12:00 Every day

Outline

- Little theory about C++ :
  - Variable
  - Environment
  - Binding
  - Scope
- Geany extensions
- Alternatives to Geany

Variables, types in C++

- A **variable** is an identifier, a name, for a memory location.
- To **define** a variable is to give a **name** and a **type** to it. This tells the compiler to find a free memory space for that variable.

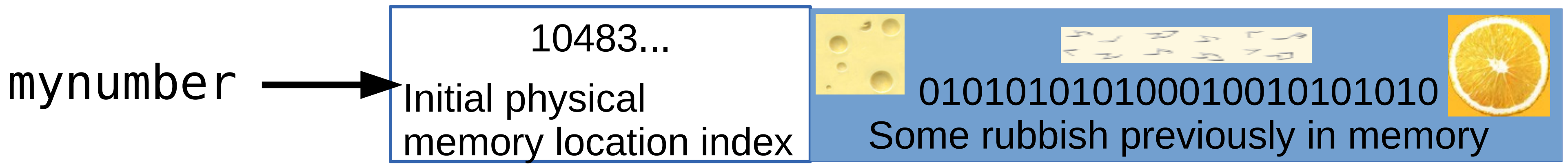
```
int number;
```

- The **type** indicates the kind of information stored inside the variable. In languages like C++ it must be declared explicitly; such languages are also called **typed languages**.
  - The type also defines **the size of the allocated memory**.
  - As the compiler reads your code (*compilation time*), it internally creates table of names of variables with their types, size, tentative memory pointers (**static allocation**).

Var name	**Var type**	**Associated size**	**Initial tentative logical memory location pointer**
`mynumber`	`int`	`sizeof(int)` `e.g. 2bytes`	10483392805

Variables, types in C++

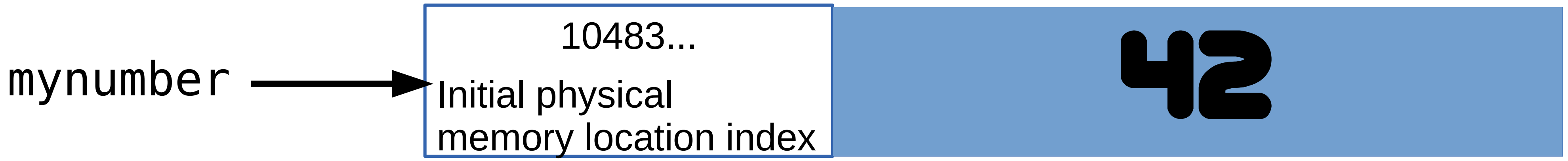
- If the variable is not **initialized**, it can contain anything. It means that at *runtime*, when the pointer actually will point to a real memory location, whatever is already there will represent the variable **value**.
  - If we were to run the code immediately **without initializing the variable**, we're not sure of what the content of the memory is:


mynumber
10483...
Initial physical memory location index
0101010101000100101010
Some rubbish previously in memory

- By **assigning** a **value** to a variable, we tell the compiler what to write in the memory.

```
number = 42;
```

Var name	Var type	Associated size	Initial tentative logical memory location pointer	value
`mynumber`	`int`	`sizeof(int)` `e.g. 2bytes`	10483392805	**42**



Environment, binding

- All the variable and function names "live" in a space called **environment**. You can think of it **as a table** in the compiler containing all variable names and their associations with memory chunks.
- A name is said to be **bound** to that environment when its value is associated to a memory index in that environment. In the table on the left we can see some bindings.
- When we **define** a variable, the variable name is added to the **environment**
- In languages like C++ we can see them in the form of **pointers**.
- Binding can be:
  - **Static**, that is, decided at **compile time**
  - **Dynamic**, that is, decided at **runtime**
    (yes one can change where in the memory that variable is pointing)

Environment	**Variable or function name**	**Starting virtual memory index assigned by compiler (at compile time)**	**Starting virtual memory index assigned by operating system (runtime)**
std	cout	Virt(#200), defined in std	physical(#ABBC)
global			
`global`	foo()	Virt(#1), defined in global	physical(#ABCC)
`foo()`	fooScope	Virt(#2), defined in foo->virt(#1)	physical(#7945)
`foo()`	Anonymous block#1	Virt(#3), defined in foo->virt(#1)	physical(#ABCC)
Anonymous block#1	blockScope	Virt(#4), defined in Anonymous block #1->virt(#3)	physical(#ABCC)

Visibility, scope

- A variable is **visible** in an environment when its binding is present in that environment, that is:
  - There **exists** a variable **name** in the environment
  - That variable name is **associated to a memory location** (this depends on languages)
- Usually a function has its own environment, that is, a *set* of variables in its own environment, and can *see* the variables in other environments according to some rules.
  These rules define the **scope**, or **visibility**, of a variable.
- In the case of C++, **blocks of code** (the curly brackets {}) are used to define new environments and scopes.
  - A variable **defined** in a block is always added to that block environment and **visible** in that block's environment. For ease of use, we say is visible in that block.
    - **Q: What happens if one uses the same names in two blocks???**
    - **A:** The memory to which that name is pointing is overridden by the last block that could change the environment.
      If you don't understand environments and scopes, you will only be able to verify this at runtime.

Functions and scopes in C++

- In C++, the environment and scopes are managed by the use of **blocks of code**.
- The general inheritance rules are as follows:
  - A block **inherits the environment from its parent block**, that is, all the variable and function names existing at the moment of opening the block are **imported** in the block environment.
  - Every variable name **defined** in a block is **added** in the environment of that block.
  - If a variable with the same name is present in the environment, the last defined variable **overrides** any other variable with the same name within that block.
    - That is, it is **not possible anymore to use the value contained in variables with the same name defined outside that block**.

Functions and scopes in C++

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl;

   { //Any block declares a scope, even this useless one
      int localScope = 3;
      cout << "localScope: " << localScope << endl;
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very bad practice!
      cout << "globalScope: " << globalScope << endl;
   }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << endl;
}
```

Functions and scopes in C++

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl;

   { //Any block declares a scope, even this useless one
      int localScope = 3;
      cout << "localScope: " << localScope << endl;
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very bad practice!
      cout << "globalScope: " << globalS    cope << endl;
   }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << endl;
}
```

Variables in the
global scope
and visible to everyone

Environment	Variable or function name	Parent environment	Val
global	globalScope		0

Functions and scopes in C++

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl;

   { //Any block declares a scope, even this useless one
      int localScope = 3;
      cout << "localScope: " << localScope << endl;
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very bad practice
      cout << "globalScope: " << globalS    cope << endl;
   }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << endl;
}
```

Variables in the
global scope
and visible to everyone

Variables
visible by **foo()**

Environment	**Variable or function name**	**Parent environment**	**Val**
global	globalScope		0
global	foo()		
global	main()		
foo()	fooScope	global	1

Functions and scopes in C++

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables not present in any environment no scope (**compile time error**!)

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl;

   { //Any block declares a scope, even this useless one
      int localScope = 3;
      cout << "localScope: " << localScope << endl;
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very bad practice
      cout << "globalScope: " << globalS    cope << endl;
   }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << endl;
}
```

Environment	Variable or function name	Parent environment	Val
global	globalScope		0
global	foo()		
global	main()		
foo()	fooScope	global	1

Functions and scopes in C++

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables not present in any environment no scope (**compile time error**!)

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl;

   { //Any block declares a scope, even this useless one
      int localScope = 3;
      cout << "localScope: " << localScope << endl;
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very bad practice!
      cout << "globalScope: " << globalS    cope << endl;
   }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << endl;
}
```

Environment	Variable or function name	Parent environment	Val
global	globalScope		0
global	foo()		
global	main()		
foo()	fooScope	global	1
main()		global	

Functions and scopes in C++

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl;

   { //Any block declares a scope, even this useless one
      int localScope = 3;
      cout << "localScope: " << localScope << endl;
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very bad practice!
      cout << "globalScope: " << globalScope << endl;
   }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << endl;
}
```

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables not present in any environment no scope (**compile time error**!)

Variables visible in the **useless block**

Environment	**Variable or function name**	**Parent environment**
global	globalScope	
global	`foo()`	
global	`main()`	
`foo()`	fooScope	global
`main()`		global
`Useless block`	localScope	`main()`
`Useless block`	globalScope	`main()`

Functions an C++

Hidden variable!

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl;

   { //Any block declares a scope, even this useless one
      int localScope = 3;
      cout << "localScope: " << localScope << endl;
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very bad practice!
      cout << "globalScop    " << globalScope << endl;
   }
   cout <<
   cout <<
}
```

Overridden variable name!

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables not present in any environment no scope (**compile time error**!)

Variables visible in the **useless block**

Environment	Variable or function name	Parent environment	Val
global	globalScope		0
global	foo()		
global	main()		
foo()	fooScope	global	1
main()		global	
Useless block	localScope	main()	3
Useless block	globalScope	main()	100

Functions and scopes in C++

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables not present in any environment no scope (**compile time error**!)

Variables visible in the **useless block**

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl;

   { //Any block declares a scope, even this useless one
      int localScope = 3;
      cout << "localScope: " << localScope << endl;
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very bad practice!
      cout << "globalScope: " << globalScope << endl;
   }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << endl;
}
```

Environment	Variable or function name	Parent environment	Val
global	globalScope		0
global	foo()		
global	main()		
foo()	fooScope	global	1
main()		global	
Useless block	localScope	main()	3
Useless block	globalScope	main()	100

Advanced Geany configuration

- These settings will help you while coding in C++.
- Find the Tools→Plugins Manager menu in Geany
- Activate the following plugins by ticking the boxes:
  - **Auto-close** (autocloses parentheses and blocks)
  - **Auto-mark** (highlights keywords you’re pointing at)
  - **Code navigation** (to switch between header and implementation)
  - **File Browser** (you can open files directly from Geany)
  - **GeanyCtags** (autocomplete of some C++ common keywords and library)
  - **Split Window** (you can divide the screen in multiple windows)
  - **TreeBrowser** (Allows you to navigate the filesystem as a tree)
- Autocomplete: while writing a function or a library name, press ALT + SPACEBAR to see possible options

Alternatives to Geany

- **Emacs / xemacs**
  - For **hardcore developers** who like to memorize a vast number of shortcuts
  - It does almost everything other IDEs do except the nice graphics.
  - Found on most Linux clusters around the world
  - available on the official Ubuntu repository, install with
    ```
    sudo apt-get install emacs xemacs21
    ```
- Any text editor you like will do. It’s just text at the end of the day. But...

IDEs

- Most coders use an **Integrated Development Environment**, a text editor with several useful tools. Here is a selection of them.
- **CodeBlocks**
  - available on the official Ubuntu repository, install with `sudo apt-get install codeblocks`
- **Codelite**
  - available on the official Ubuntu repository, install with `sudo apt-get install codeblocks`
- **Eclipse (DO NOT USE ON LUBUNTUVM!)**
  - Java-based (make it slow on machines with low memory)
  - Widely used, but not for C++
  - Can only be downloaded from their website: http://www.eclipse.org/downloads/packages/release/luna/r/eclipse-ide-cc-developers
- Many more, see https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#C/C++