C++ 02 - Variables, Scopes & Types

Feb’22

EPITA Research & Development Laboratory (LRDE)

Variable & Symbol Visibility

The power of `{}`

Block scope

Each name that appears in a C++ program is only visible in some possibly discontinuous portion of the source code called its scope.

Block scope: From its declaration to the end of the block (next `}`).

```
// Code that does not use x
int x = 0; // Outer scope x of type int
// Code using outer scope integer x
{
  // Code using outer scope integer x
  float x = 1.1f; // Inner scope x, a float, begins
  // Code using the inner scope float x
}
```

Function parameter scope

Entity type	Scope
Variable	From declaration to the block’s end (`}`)
Function parameter	From function start `{` to the function’s end (`}`)

The compiler “searches’’ for names from the innermost to the outermost block.

Function parameter scope

Entity type	Scope
Variable	From declaration to the block's end (})
Function parameter	From function start { to the function's end (})

The compiler "searches" for names from the innermost to the outermost block.

Find the bug:

```
int f(int i, // Scope of "parameter i" starts
      int j  // Scope of "parameter j" starts
      ){
  // Code using "parameter i" and ``parameter j''
  if (j > 10){
    int i = i + j; // Scope of ``variable i'' starts,
                   // Scope of ``parameter i'' is paused
    // Code using ``variable i''
  }
  // Scope of ``parameter i'' resumes
} // Scope of ``parameter i'' and ``parameter j'' ends
```

Namespace scope - 1

Namespaces are a way to *modularize* your code by preventing naming conflicts.

One of them is `std`, which contains all types and functions defined in the C++ standard library.

In order to use a name from a namespace, it needs to be prefixed:

```
#include<iostream> // Contains cout, cin, endl etc
                   //inside of std
std::cout << "abc" << std::endl;
```

All names inside a namespace can be made available without the prefix with `using namespace std;` (do not do this 👎).

Namespace scope - 2

Namespace definition

```
namespace mymodule {
  int f(int i);
  namespace submodule { int g(int i); }
}
void h();
void foo();
```

- `h` and `foo` are in the *global* namespace
- Namespaces can be nested and spread over multiple files

Namespace scope - 2

Namespace definition

```
namespace mymodule {
  int f(int i);
  namespace submodule { int g(int i); }
}
void h();
void foo();
```

- `h` and `foo` are in the *global* namespace
- Namespaces can be nested and spread over multiple files

Symbol lookup

- Symbols have a *path*, the path separator is `::` (instead of `/`)
  The root path (global namespace) is `::` (instead of `/`)
- *Qualified*-lookup: `mymodule::submodule::g(2)`
  Look for `g` in the `mymodule:submodule` namespace (relative to the current one)
- *Unqualified*-lookup: `g(2)`
  Look for `g` in current *scope*, and goes upward if it is not found

foo.hpp

```cpp
namespace mymodule {
  int f(int i);
  namespace sub {
    int g(int i);
  }
}
void g();
void foo();
```

foo.cpp

```cpp
namespace mymodule {
  int f(int i) {
    g();                     // -> ::g()
    return sub::g(i);  // -> ::mymodule::sub::g
  }
}
namespace mymodule::sub {
    int g(int i) {
      f(i);    // -> ::mymodule::f
      ::g();   // -> ::g
      g(i-1);  // -> ::mymodule::sub::g
    }
}
void g() {  mymodule::f(2); } // -> ::mymodule::f
void foo() { g(); }           // -> ::g
```

Lifetime and symbol visibility

Storage classes - Storage Duration

The storage class specifiers control the *storage duration* and the *linkage*.

Storage duration

Automatic: The objects are allocated at the beginning of the scope and deallocated at scope’s end.

Static: The objects are allocated when the program starts and deallocated when the program ends.

Storage classes - Storage Duration

The storage class specifiers control the *storage duration* and the *linkage*.

Storage duration

Automatic: The objects are allocated at the beginning of the scope and deallocated at scope’s end.

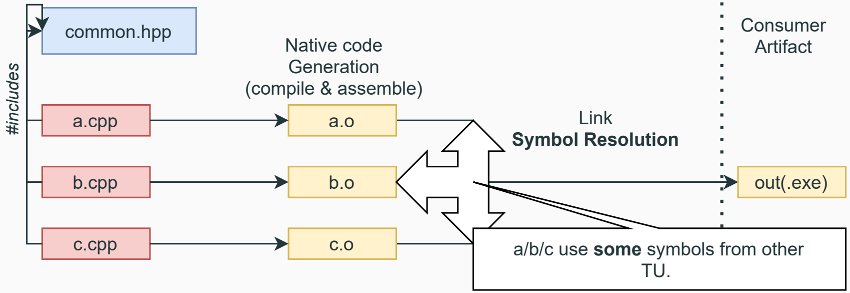
Static: The objects are allocated when the program starts and deallocated when the program ends.

☞ There is also **thread-local**, the objects have a per-thread copy (let us forget this for now)

☞ There is no garbage collector in C++. The lifetime of each object is *known*.

Storage classes - Storage Duration

The storage class specifiers control the *storage duration* and the *linkage* of a name.



Linkage

- **external linkage**: Names that can be referred by other TU.
- **internal linkage**: Names that can only be referred to in the same TU.

`static`, `extern` keywords

Declaration of variables in a function

	Duration	Linkage
`int i`	Automatic	No
`static int i`	Static	No

Declaration in a namespace or global scope

	Duration	Int. Link.	Ext. Link.
Variable (Global)	Static	`[static] int i;`	`extern int i;`
Function/Enum	n/a	`static void foo()`	`[extern] void foo()`

☞ You should never use the keyword `static`, use anonymous namespaces 👍

One-time initialization of a module

a.hpp

```cpp
#pragma once
enum Color { RED, GREEN, BLUE};
Color foo();
Color bar();
extern bool verbose; //<-Declaration
```

main.cpp

```cpp
#include "a.hpp"
int main()
{
  Color c = foo();
  if (verbose)
    stc::cout << "Foo result="
              << int(c) << "\n";
}
```

a.cpp

```cpp
#include "a.hpp"
namespace { // <- Anonymous namespace
  bool is_initialized = false;
  void load_config() {
    if (!is_initialized) {
      // Load configuration from a file
      // and set verbose
      is_initialized = true;
} } }
bool verbose; //<-Definition

Color foo() { load_config();
  return GREEN;
}
Color bar() { load_config();
  return RED;
}
```

C++ Basic Types

Fundamental types

C++ is *strongly* **statically** typed !

- Variables have a type, each function has a signature and the compiler ensures coherence.
- More type safety than C (`void*` is barely used)

Integers	bool	Floating Points	Pointer	Arrays
`[unsigned] short`	`bool`	`float`	`T*`[1]	`T[]`
`[unsigned] int`		`double`	`nullptr_t`[2]	`std::array<T>`
`[u]int8_t, [u]int16_t...`		`long double`		
`size_t, ptrdiff_t`				

Char Types	Text
`char`	~~`char*`~~
`wchar_t`[3]	`std::string`
	`std::wstring`

[1]:`T` is a placeholder for any type (pointer included)
[2]: C: `NULL`, C++: `nullptr` [3]: Support unicode codepoints

Conversion rules

Like in C, you can safely convert to a wider type, issues a warning when narrowing.
Beware of what is considered widening!

```
float       get_float() {return 5.f; };
int         get_int()   {return 6i; };
std::string s = "C++ is 👍";

// Ok, same type
float  f = get_float();
// Ok, implicit widening conversion
double d = get_float();
// Not Ok, implicit narrowing conversion (size_t vs int) (warning)
int    i = s.size();
// Ok, explicit cast (However we don't do those normally)
int    j = static_cast<int>(s.size());
// Ok for the compiler (widening) but beware of rounding
float x = get_int();
```

Foundamental types (cont.)

…	Pointer	Arrays	**Reference**
	`T*`	`T[]`	`T&`

Pointer and Addresses

- A *pointer object* holds the address of another object
- The *pointer type* includes the type (`T` placeholder) of the pointee.

```
int*   iptr;      // Pointer to a int
char** cstr_ptr;  // Pointer to a raw c-string
```

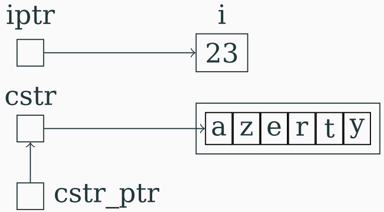
Pointer and Addresses

- A *pointer object* holds the address of another object
- The *pointer type* includes the type (T placeholder) of the pointee.

```
int*   iptr;      // Pointer to a int
char** cstr_ptr;  // Pointer to a raw c-string
```

- The expression &x is the address of x.

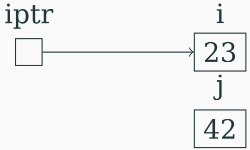
```
int i           = 23;
char  chr_arr[] = "azerty";
char* cstr      = chr_arr;


int* iptr = &i;
char** cstr_ptr = &cstr;
```



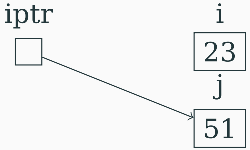
☞ Reminder: Arrays are not pointer but can decay as a pointer to the first element.

And pointer can be updated to point to another object.

```
int i = 23, j = 42;
int* iptr = &i;
```



```
iptr = &j;
*iptr = 51;
```



C++ References

References

References are an alternative to pointers. They:

- are **non-null** constant pointer with a non-pointer syntax
- are variables that *alias* other objects
- *cannot* be redirected once initialized !

```
int i;


int& iref = i;
int& jref;        // Error, a reference must be initilized
int& jref = iref;
jref = 3;
```

- `iref` is of type *reference to int*
- initialize `iref` to refer to i.
- `iref` behaves just like `i` (`jref` also refers to `i`)

Reference notation	Pointer notation
`int& iref = i`	`int* const iptr = &i` [1]
`iref = 51`	`*iptr = 51`
`int j = iref + 2`	`int j = *iptr + 2;`

[1] **const** keyword will be seen soon

Swap

```c
// C swap
void int_swap(int* pi1,
              int* pi2)
{
  int tmp = *pi1;
  *pi1 = *pi2;
  *pi2 = tmp;
}

void foo()
{
  int i = 5, j = 1;
  swap(&i, &j); // pointers
}
```

```cpp
// C++ swap
void swap(int& i1,
          int& i2)
{
  int tmp = i1;
  i1 = i2;
  i2 = tmp;
}

void foo()
{
  int i = 5, j = 1;
  swap(i, j); // references
}
```

References are non-optional in Modern C++

References are not just **sugar** for pointers, they are everywhere in modern C++.

Modifying elements with for-loop range (see later)

```cpp
int vals[] = {1, 2, 3, 4};

for (int& v : vals)
  v += 1;
```

Creating custom operators that look like native ones

```cpp
// Declaring an operator for a custom type (signature)
Matrix& operator+= (const Matrix& other);

Matrix a, b;
a += b;
```

That would be impossible without references !

C++ `const` types

`Const`-ness in C++

`const` keyword stands for constant, it is used[2] :

- To define **symbolic constants** `const float PI = 3.14`
- To define **immutable data** [3]

```
const int SOBEL[][] = {{-1, 0, 1},
                       {-2, 0, 2},
                       {-1, 0, 1}};
```

- To says that a **mutable** data is not going to be modified (especially function parameters).

[2] Back to the basics: `Const` as promise
[3] `constexpr` is now mostly used for this

Reading the `const` types

- `const` is part of the type:
  `const int` ≠ `int` and `const char*` ≠`char*`
- `*` and `&` are separators in the type:
  `int* const p` ≠ `const int* p` but “`const int* p` = `int const* p`

Reference notation	Pointer notation
`int& iref = i`	`int* const iptr = &i`
`const int& iref = i`	`const int* const iptr = &i`

`Const` in C++ interface (1/2)

`const` types are part of the idiomatic C++ interfaces that makes them:

- easier to use correctly, harder to use incorrectly

This is a promise that says *This **mutable** input is **read-only***
→ Use `const` only with pointers and references in interfaces

`Const` in C++ interface (1/2)

`const` types are part of the idiomatic C++ interfaces that makes them:

- easier to use correctly, harder to use incorrectly

This is a promise that says *This **mutable** input is **read-only***
→ Use `const` only with pointers and references in interfaces

	input param (r)	*output* (w) or *inout* (rw)
Small object	`foo(int)`	`foo(int&)`
Large object	`foo(const FILE&)`	`foo(FILE&)`
Optional Small object	`foo(std::optional<int>)`	`foo(int*)`
Optional Large object	`foo(const FILE*)`	`foo(FILE*)`

☞ The usage of *pointer vs reference* in interfaces is mostly about nullable.

☞ Passing by *value vs reference* are just for *read-only* parameters and depends on the object size

`Const` in C++ interface (2/2)

(const) reference/raw pointer as the return type is rare.
Mostly it is when forwarding a parameter.

```
Matrix& add(Matrix&, const Matrix&);
```

`Const` in C++ interface (2/2)

(const) reference/raw pointer as the return type is rare.
Mostly it is when forwarding a parameter.

```
Matrix& add(Matrix&, const Matrix&);

Matrix a,b,c;
add(add(a,b), c);
```

☞ Note that types in the signature tells the usage (no doc, no name required)
☞ Never return a reference/pointer to a local variables
    The object will expire → dangling reference/pointer!

`auto` **everywhere ?**

`auto` is the right type

```
std::string s = "example";
int  szi = s.size(); // <- narrowing conversion
auto szs = s.size(); // Deduced std::size_t, no conversion
```

Placeholder type specifier

`auto` (C++-11) is a placeholder type specifier. For variables, the type that is being declared will be automatically deduced from its value see.

`auto` is the right type

```cpp
std::string s = "example";
int  szi = s.size(); // <- narrowing conversion
auto szs = s.size(); // Deduced std::size_t, no conversion
```

Placeholder type specifier

`auto` (C++-11) is a placeholder type specifier. For variables, the type that is being declared will be automatically deduced from its value see.

```cpp
auto i = 0; // i is an int.
auto u = 0u; // u is an unsigned int.
auto s = std::string{"foo"} // s is a string
auto it = std::begin(s); // it has a really ugly type.
```

Pros of auto (1/2)

- Increases performance as no implicit conversions are performed
- Prevents uninitialized variables (!)
- `auto` is robust to minor change
  - Makes code easier to maintain
  - We want to capture the *semantics* not the type

```
int complex_computation(int x) {
  int v = x + 43.;
  int w = v * M_PI;
  return v;
}
```

```
auto complex_computation(float x) {
  auto v = x + 43.;
  auto w = v * M_PI;
  return v;
}
```

Pros of auto (2/2)

- `auto` is essential when **types are unknown as in generic functions** (coming soon)
- `auto` is handy for long types
- `auto` avoids stuttering code

```cpp
std::vector<std::string>* v_ptr =
  new std::vector<std::string>();

typename std::vector<int>::const_iterator i
  = std::begin(v);
```

```cpp
auto  v_ptr = new std::vector<std::string>();
auto* v_ptr = new std::vector<std::string>();

auto i = std::begin(v); // shorter!

auto c = some_lib::get_container();
auto i = std::begin(c);
```

const and *references* with `auto`

Note that `auto` drops `const` and `&` from the type (it gives the “raw” type) by default.
But, you can force the type.

```
auto j = jumbo(10);  // A large object
auto& jr = j;        // A read-write alias, auto = jumbo
const auto& jcr = j; // A read-only alias,
                     // auto = jumbo
auto jptr = &j;      // A pointer to j,
                     // auto = jumbo*
auto j2 = jr;        // A copy, auto = jumbo
```

Annexes and reference

Reference

```
int i = 1;
int& j = i;
j = 2;
bool b = i == 2;
// b is true or false?
```

Reference

```
int i = 1;
int& j = i;
j = 2;
bool b = i == 2;
// b is true or false?
```

b is `true`

```
int i = 3, j = 4;
int& k = i;
k = j;
j = 5;
// i == ?  k == ?
```

Reference

```
int i = 1;
int& j = i;
j = 2;
bool b = i == 2;
// b is true or false?
```

b is `true`

```
int i = 3, j = 4;
int& k = i;
k = j;
j = 5;
// i == ?  k == ?
```

`i == 4  k == 4`

This is what *C* code looks like:

```
int i = 1;
int *const p_j = &i;
*p_j = 2;
bool b = i == 2; // true
```

```
int i = 3, j = 4;
int *const p_k = &i;
*p_k = j;
j = 5;
// i == 4  *p_k == 4
```

Reference in return types

```
std::string& foo()
{
  std::string x = "go";
  return x
}
```

Reference are like pointers. Returning a reference to local variables creates a dangling reference to an object whose lifetime as expired (at `}`).

Constness example

```cpp
int f1(int i){return ++i; } // Ok
int f2(const int i){return ++i; } // Does not compile
int f3(int& i){return ++i; } // Ok
int f4(const int& i){return ++i; } // Does not compile

int i = 1; // Ok
int& ir = i; // Ok
const int& icr = i; // Ok
const int j = 2; // Ok
int& jr = j; // Does not compile
const int& jcr = j; // Ok
```

Constness example

```
int f1(int i){return ++i; } // Ok
int f2(int& i){return ++i; } // Ok

int i = 1; // Ok
const int& icr = i; // Ok
const int j = 2; // Ok

f1(i); // Ok
f1(icr); // Ok
f2(i); // Ok, i is now 2
f2(icr); // Does not compile
f1(j); // Ok
f2(j); // Does not compile
```