C++ Software Engineering

for engineers of other disciplines

Module 1

"C++ Syntax"

2nd Lecture: **`hello_world();`**

ALTEN

Autumn 2021

Gothenburg, Sweden

petter.lerenius@alten.se

rashid.zamani@alten.se

Proper Hello World!

- **`main`** indicates the program is an executable and it is also the entry point for the program execution.
- **`main`** always takes two inputs:
  - **`int argc`**: the number of arguments passed to the program upon execution
  - **`char* argv[]`:** an array filled with the actual arguments in string format, the size of the array is **`argc`**
- **`main`** is always passed the name of the program, meaning **`argc >= 1`**

- **`main`**'s return value could be captured from terminal when executing the program – zero usually indicates normal termination.
- **`argc`** (argument count) and **`argv`** (argument values) are conventional names which are used universally for better readability, otherwise they could be named anything.

helloworld.cpp > main()

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
}
```

```
mrz@vbubu:~/projects/HelloWorld$ ./hw
Proper Hellow World, from: ./hw
```

helloworld.cpp > helloworld

```cpp
#include <iostream>

int main(int argc,char* argv[]) {
    std::cout << "Proper Hellow World, from: " << argv[0] << std::endl;
    return 0;
}
```

- **`main`** inputs could be empty if they are not used, to avoid warnings compiler generates for unused variables, but the function shall always return an integer.

Pointers

- Address to a specific memory cell
- Declared using `*`:

```
SomeDatatype *PointerName;
```

- Address of *ordinary* variables could be fetched using `&`:

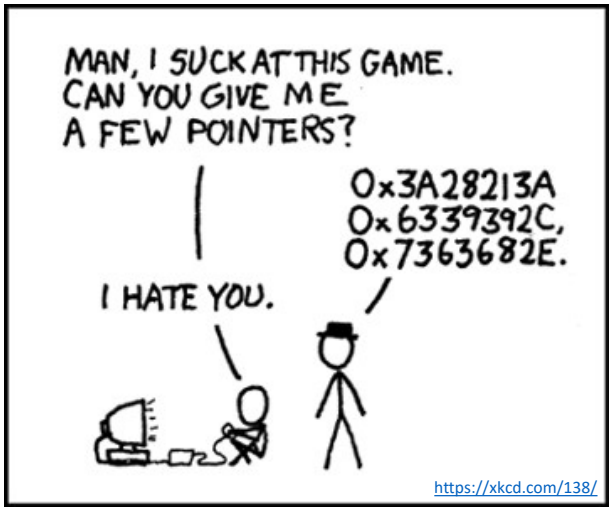
```
SomeDatatype *PointerName = &VariableName;
```

- Pointers can be ***initialized*** using `new` keyword

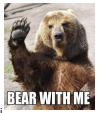
```
SomeDatatype *PointerName = new SomeDatatype;
```

0x7ffd8a26b400	0x48
0x7ffd8a26b401	0x69
0x7ffd8a26b402	0x00

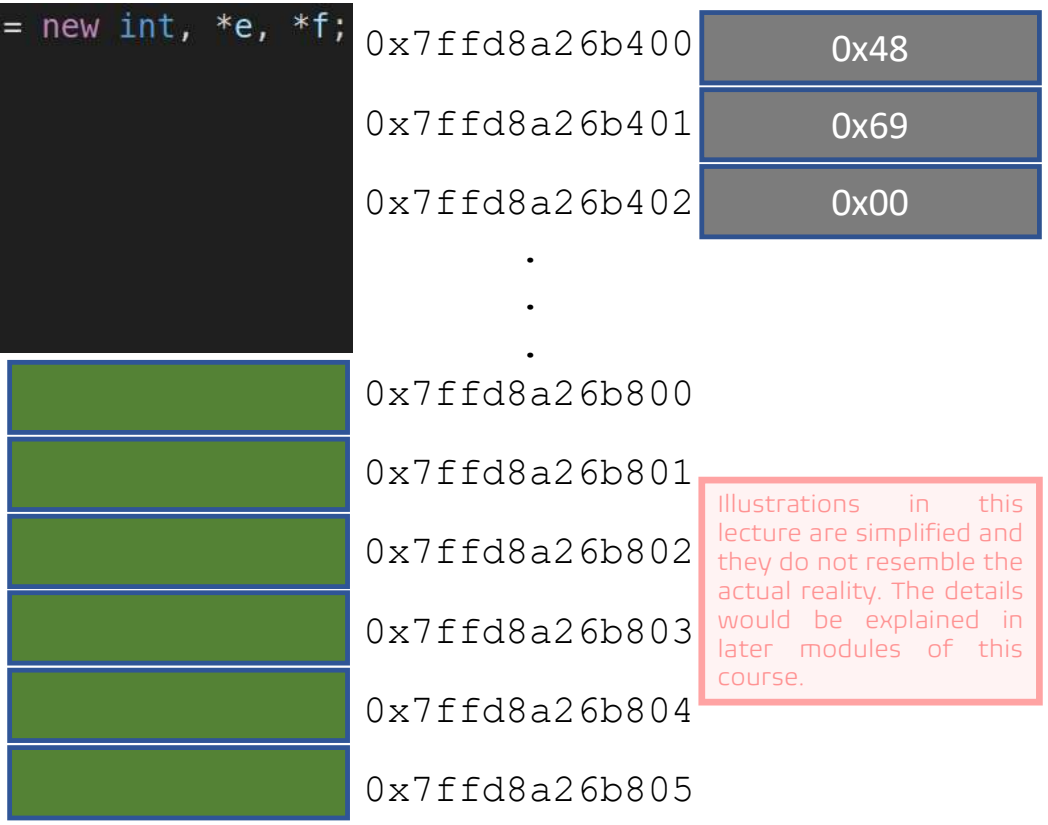
- `char*` is the address to a memory cell holding a character, yet since strings are null terminated, then string values could be stored there – basically the length is from the beginning address stored in `char*` and the end is when the memory cell holds a value of null (0x00) e.g. if the `char*` pointer variable points to a cell holding 0x48, followed by 0x69 and 0x00, then it is actually holding the value for string *"Hi"*.



Pointers In Action




```
1   int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
2   *c = 22;
3   c  = &a;
4   *d =  b;
5   b  = *c;
6   d  =  c;
7   e  =  d;
8   *f = 22;
```



- Compilers convert the source code into machine codes a.k.a. binaries. In GNU/Linux, binaries are represented in Executable and Linkable Format (ELF), and when loaded (executed), there are instructions on how reserve space in different spaces of the memory for the variables in the source code. There are three types of memory available to a program: static, automatic a.k.a. call *stack* (shown in cells colored in green) and dynamic which includes bot *heap* and *free store* (grey cells). Differences between these types of memory would be covered throughout the course. Loader is provided with regions of memory from OS to follow the instruction in the ELF file.

Illustrations in this lecture are simplified and they do not resemble the actual reality. The details would be explained in later modules of this course.

```
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
 linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=4c1cc299c5
f4d54ab47e5dee5cf5088e7eafa4f7, for GNU/Linux 3.2.0, not stripped
```

Pointers In Action


```
1  int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
2  *c = 22;
3  c  = &a;
4  *d =  b;
5  b  = *c;
6  d  =  c;
7  e  =  d;
8  *f = 22;
```

- Automatic memory, or *call stack* is where the local variables are stored.
- Dynamic memory is used for memory allocated explicitly by invoking allocation functions like **new**.
- Memory addresses here are just for the purpose of illustration – dynamic memories are usually allocated on higher address compared to static memories.
- Memory cells could vary in size depending on the type they want to store the data for. For simplifications, everything is depicted in the same size.
- Null pointer (**nullptr**) is a distinct type that is not itself a pointer type or a pointer to member type.

- Dynamic memory allocated using **new** is instantized in *free store* region and initialized in the default value of the type – for integers the default value is zero.

Pointers In Action


```
1   int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
2   *c = 22;
3   c  = &a;
4   *d =  b;
5   b  = *c;
6   d  =  c;
7   e  =  d;
8   *f = 22;
```

- In order to access the memory space a pointer is pointing to, asterisk (*) is used – the procedure is called dereferencing a pointer.

	0x7ffd8a26b400	22
	0x7ffd8a26b401	0x69
	0x7ffd8a26b402	0
	.	
	.	
	.	
a	14	0x7ffd8a26b800
b	41	0x7ffd8a26b801
***c**	0x7ffd8a26b400	0x7ffd8a26b802
***d**	0x7ffd8a26b402	0x7ffd8a26b803
***e**	nullptr	0x7ffd8a26b804
***f**	nullptr	0x7ffd8a26b805

Pointers In Action

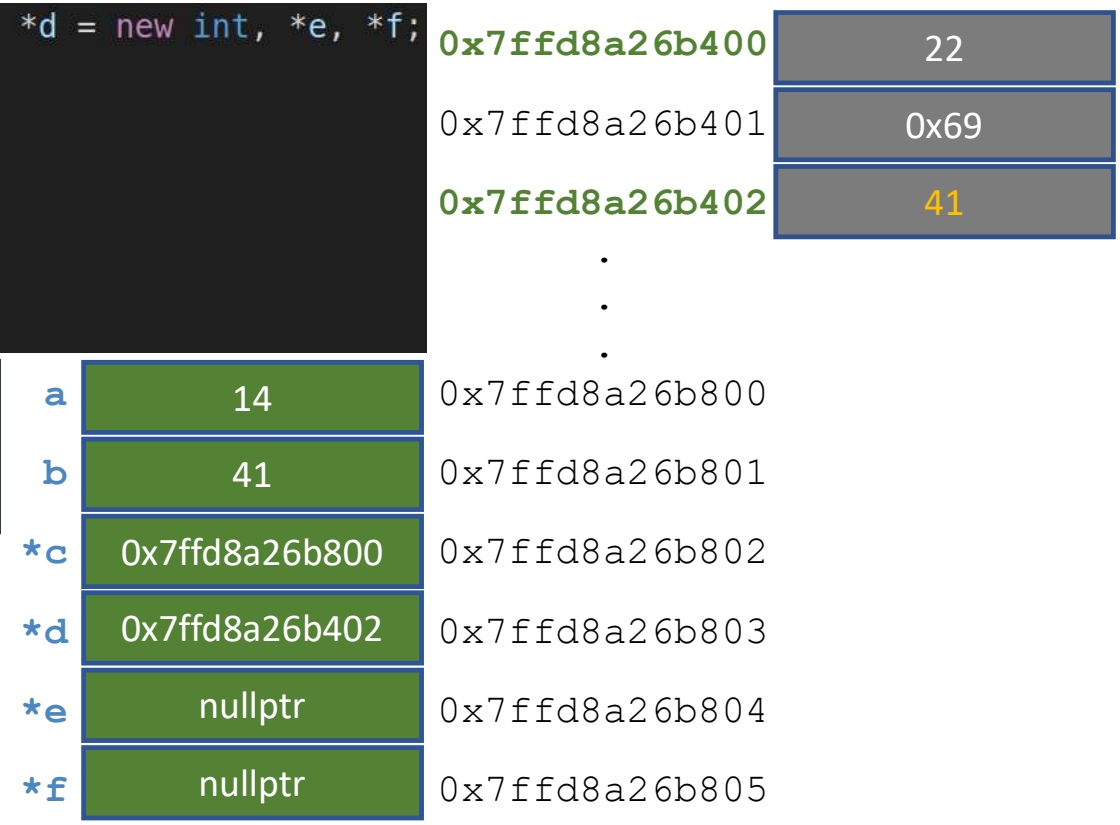

```
1   int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
2   *c = 22;
3   c  = &a;
4   *d =  b;
5   b  = *c;
6   d  =  c;
7   e  =  d;
8   *f = 22;
```

- Address of ordinary values could be assigned to pointers using reference operator (`&`).
- Notice the dynamic memory cell address `c` used to point to, is not known to the program anymore, after assigning the address of `a` into `c`.

Pointers In Action


```
1   int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
2   *c = 22;
3   c  = &a;
4   *d =  b;
5   b  = *c;
6   d  =  c;
7   e  =  d;
8   *f = 22;
```

- Pointers could be dereferenced and assigned a value of a same type from a variable. The cell the pointers point to is obviously of the same type of the pointer.



Pointers In Action

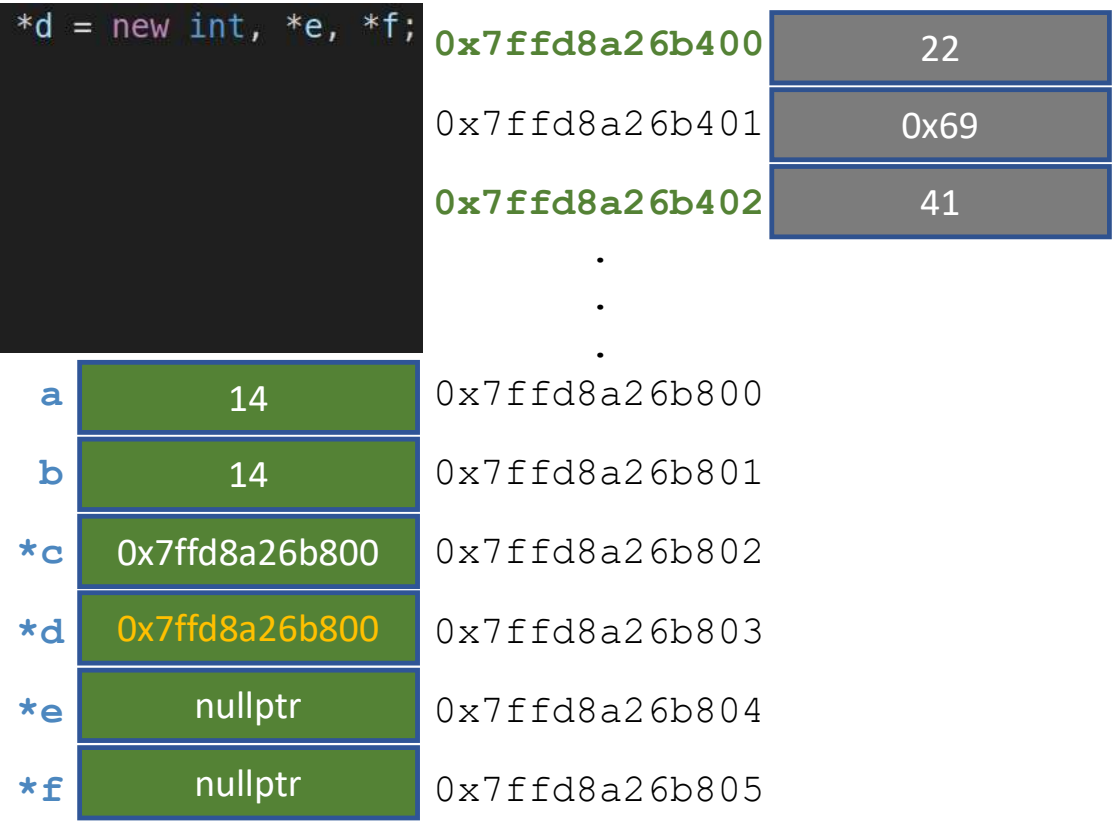

```
1   int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
2   *c = 22;
3   c  = &a;
4   *d =  b;
5   b  = *c;
6   d  =  c;
7   e  =  d;
8   *f = 22;
```

- Variables could be assigned to the value a pointer points to, once the pointer is dereferenced.

Pointers In Action


```
1   int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
2   *c = 22;
3   c  = &a;
4   *d =  b;
5   b  = *c;
6   d  =  c;
7   e  =  d;
8   *f = 22;
```

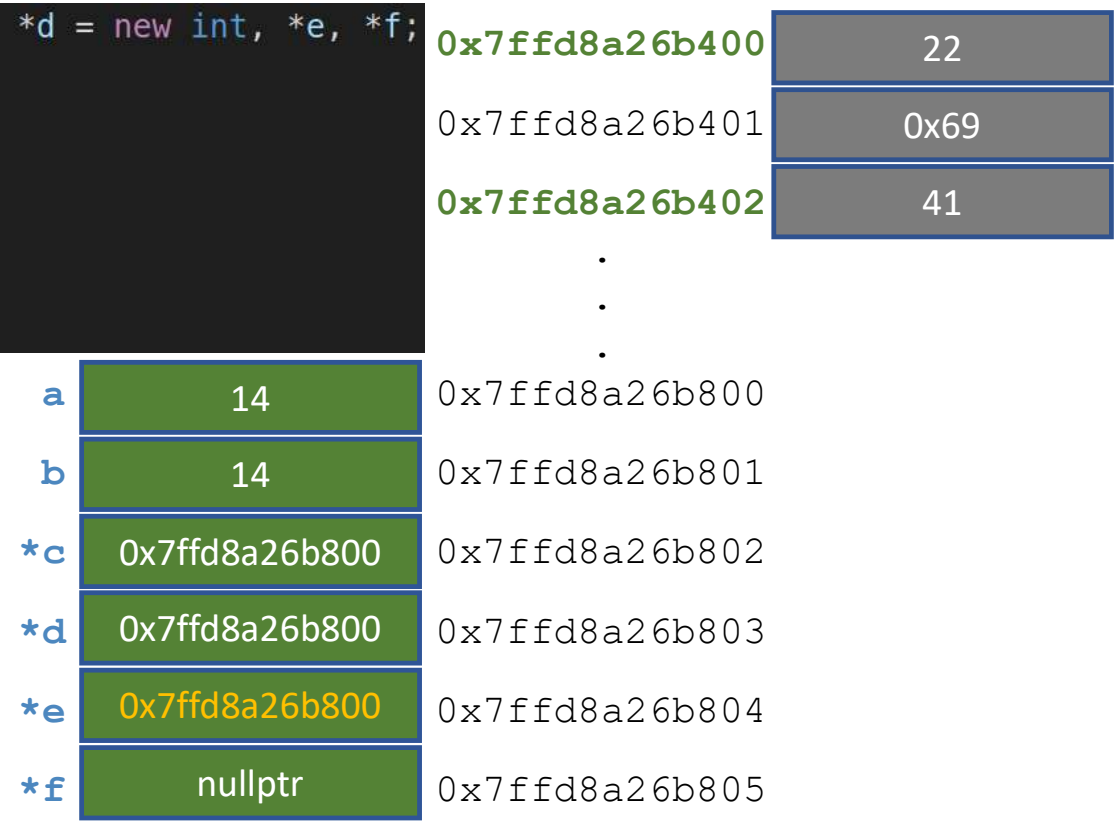
- The address a pointer points to, could be assigned to another pointer – both variables are pointing to the same location after the assignment operation, naturally.
- Notice the dynamic memory cell address `d` used to point to, is not known to the program anymore, after assigning the address of `c` into `d`.



Pointers In Action


```
1   int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
2   *c = 22;
3   c  = &a;
4   *d =  b;
5   b  = *c;
6   d  =  c;
7   e  =  d;
8   *f = 22;
```

- A pointer which is not locating to any available memory is known as uninitialized pointer.
- Uninitialized pointers could be assigned the value of other initialized pointers – both variables would then point to the same location of the memory.



Pointers In Action

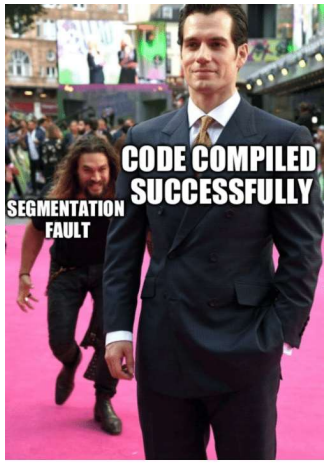

```
1   int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
2   *c = 22;
3   c  = &a;
4   *d =  b;
5   b  = *c;
6   d  =  c;
7   e  =  d;
8   *f = 22;
```

Segmentation fault (core dumped)

- Any operation including accessing the location of an uninitialized pointer such as dereferencing, would result in segmentation fault error.

"In computing, a segmentation fault (often shortened to segfault) or access violation is a fault, or failure condition, raised by hardware with memory protection, notifying an operating system (OS) the software has attempted to access a restricted area of memory (a memory access violation)." https://en.wikipedia.org/wiki/Segmentation_fault

	0x7ffd8a26b400	22
	0x7ffd8a26b401	0x69
	0x7ffd8a26b402	41
a	0x7ffd8a26b800	14
b	0x7ffd8a26b801	14
*c	0x7ffd8a26b802	0x7ffd8a26b800
*d	0x7ffd8a26b803	0x7ffd8a26b800
*e	0x7ffd8a26b804	0x7ffd8a26b800
*f	0x7ffd8a26b805	nullptr



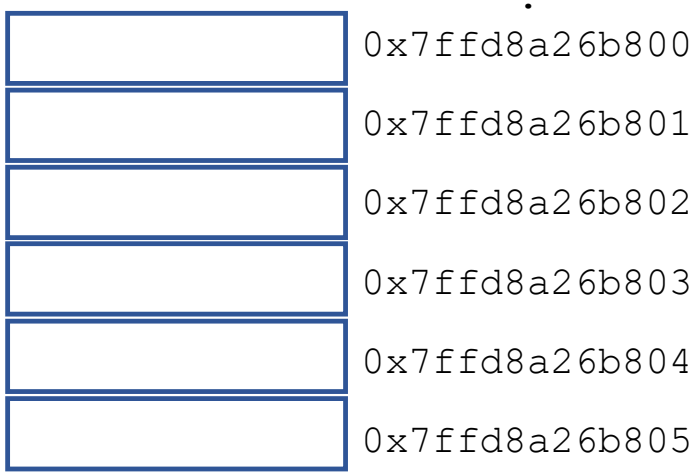
It happens every time...

Memory Leakage

ALTEN

- Initialized memory using new should be deleted to not leak!

```
SomeDatatype *PointerName = new SomeDatatype;
delete PointerName;
```



- Static and automatic memories are cleaned up by the loader automatically, once the execution exits the function's body, or the application execution terminates.
- Instructions on cleaning up the dynamic memory should be provided by the programmer. Dynamic memory which is not cleaned up would be still accessible after execution of the program even terminates. This imposes both security concerns and resource exhaustion worries.

"In computer science, a memory leak is […] (the) memory which is no longer needed (but) is not released […] they can exhaust available system memory […] (and) […] cause […] software aging." https://en.wikipedia.org/wiki/Memory_leak

Dynamic Memory

- Size of the variables stored in static memory size should be known at compile time, however, dynamic memory is a memory acquired at runtime.
- Dynamic memory is created by invoking either **new** operator or C style allocation functions. The memory should be cleaned up using a call to **delete** or **free** respectively.
- In C++, **new**/**delete** are the favorable method to create and release dynamic memory.

```
SomeDataType *PointerVariable =
                reinterpret_cast<SomeDataType*>( malloc( sizeof(SomeDataType) ) );
free (PointerVariable);
PointerVariable = nullptr;

PointerVariable =
                reinterpret_cast<SomeDataType*>( calloc(10, sizeof(SomeDataType) ) );
free (PointerVariable);
PointerVariable = nullptr;

PointerVariable = new SomeDataType; // preferred in C++
delete PointerVariable;
PointerVariable = nullptr;
```

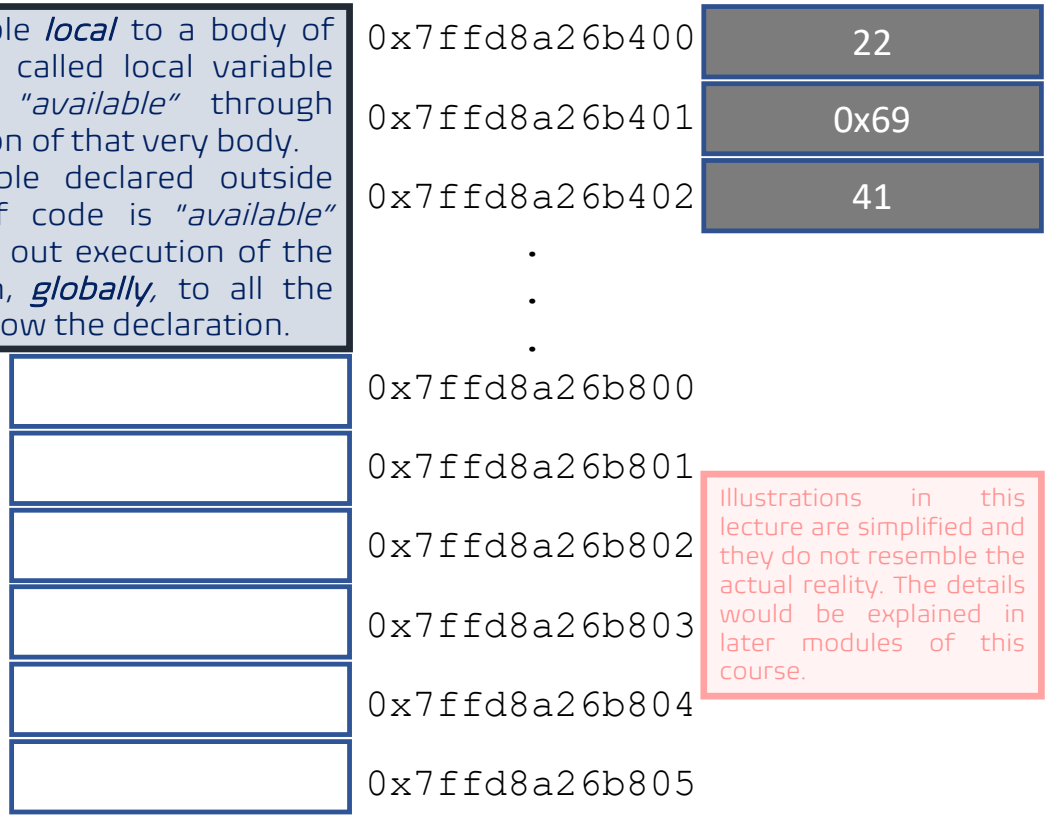
```
void *malloc(size_t __size)
Allocate SIZE bytes of memory.
```

- There are two different regions in dynamic memory at conceptual level; these are heap for C style allocations and free store for C++ **new**. Although these might be very similar regions in low levels, yet free store is more desirable in C++.

Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

- A variable ***local*** to a body of code, is called local variable and is *"available"* through execution of that very body.
- A variable declared outside body of code is *"available"* through out execution of the program, ***globally***, to all the code below the declaration.

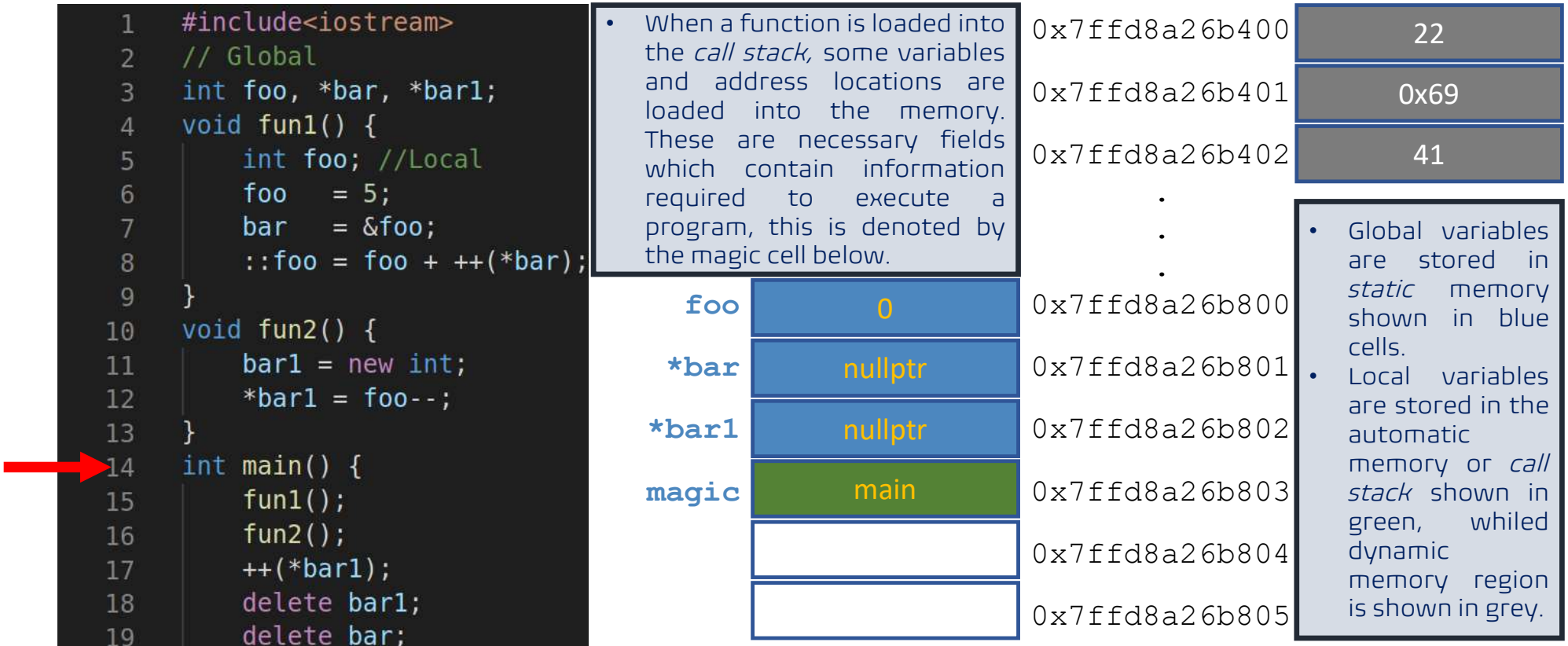


Illustrations in this lecture are simplified and they do not resemble the actual reality. The details would be explained in later modules of this course.

Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

- When a function is loaded into the *call stack,* some variables and address locations are loaded into the memory. These are necessary fields which contain information required to execute a program, this is denoted by the magic cell below.

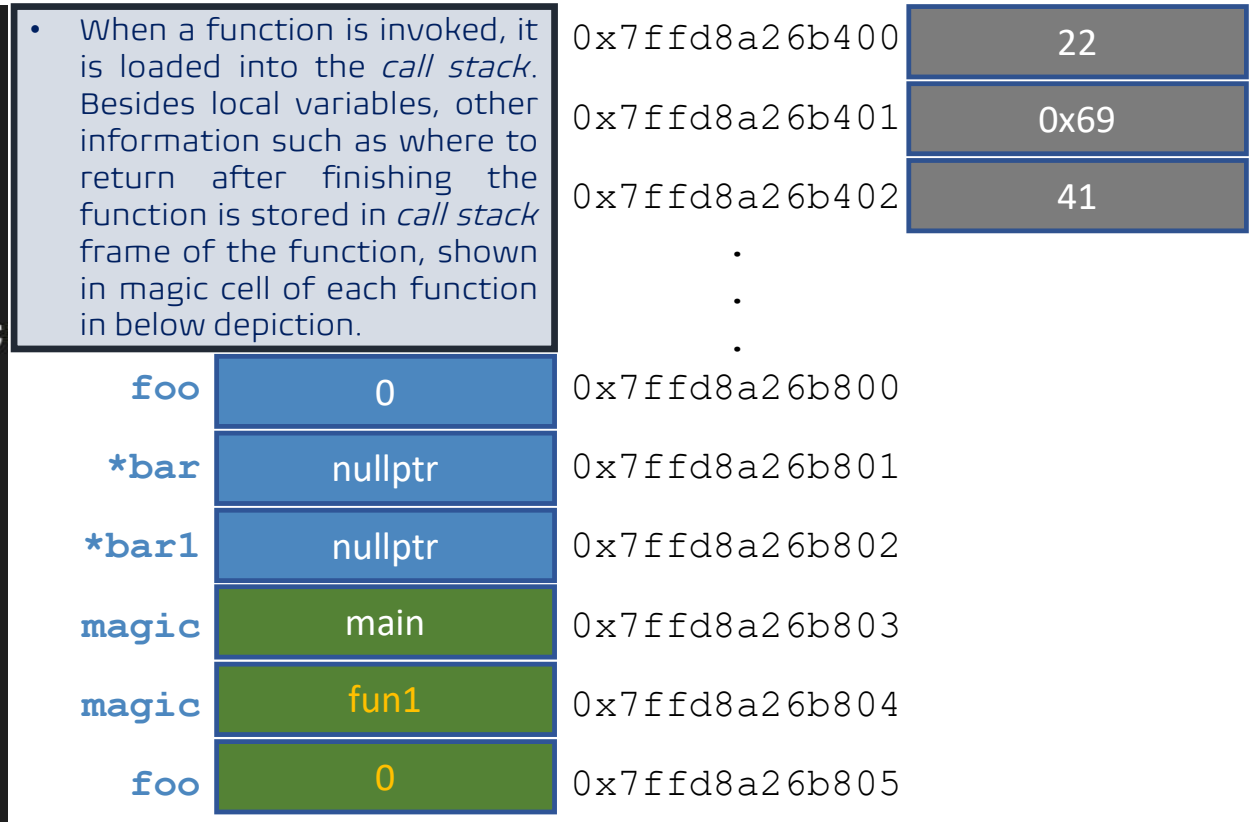


- Global variables are stored in *static* memory shown in blue cells.
- Local variables are stored in the automatic memory or *call stack* shown in green, whiled dynamic memory region is shown in grey.

Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

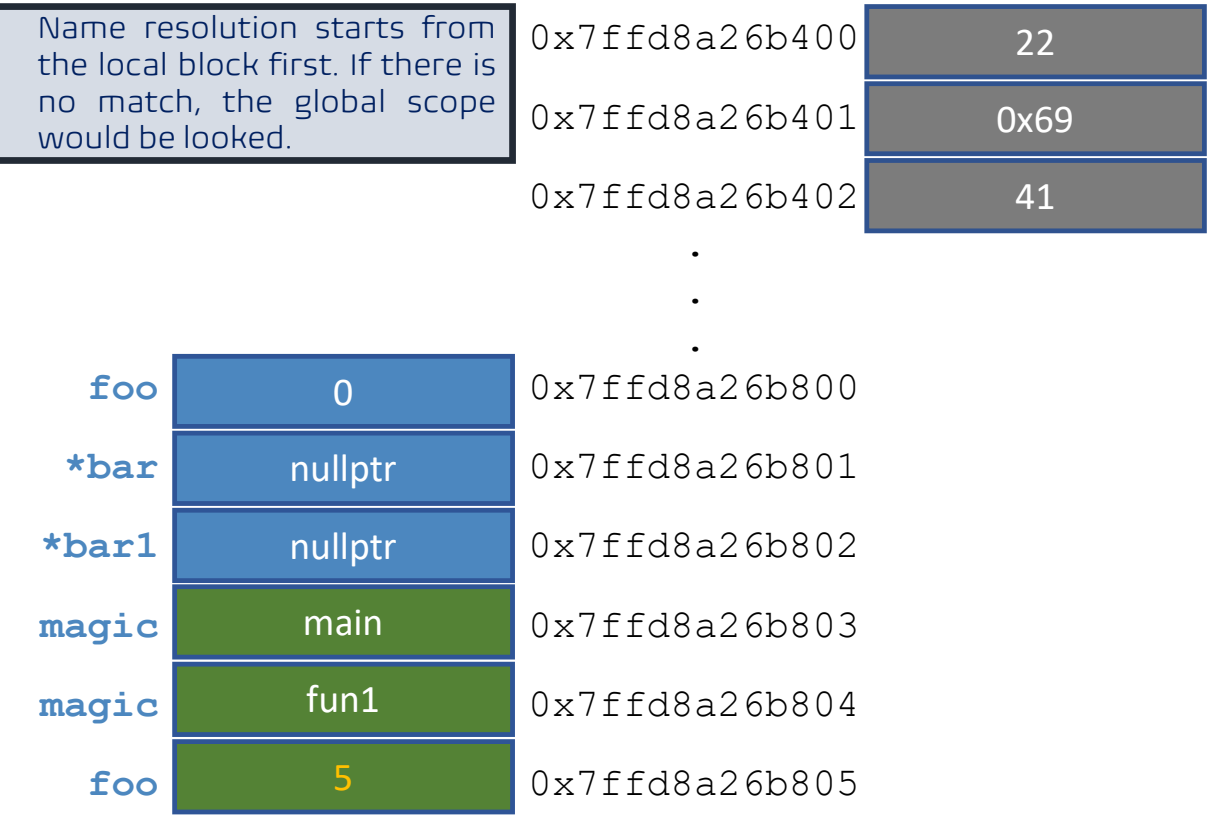
- When a function is invoked, it is loaded into the *call stack*. Besides local variables, other information such as where to return after finishing the function is stored in *call stack* frame of the function, shown in magic cell of each function in below depiction.



Scope

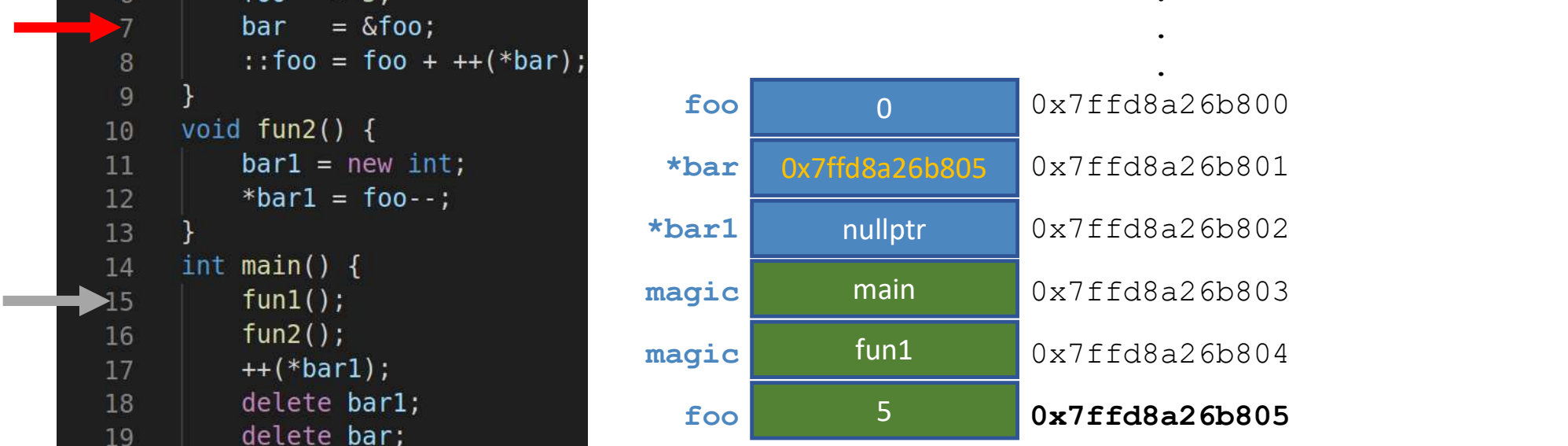

```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

- Name resolution starts from the local block first. If there is no match, the global scope would be looked.

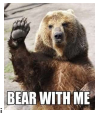


Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```



Scope



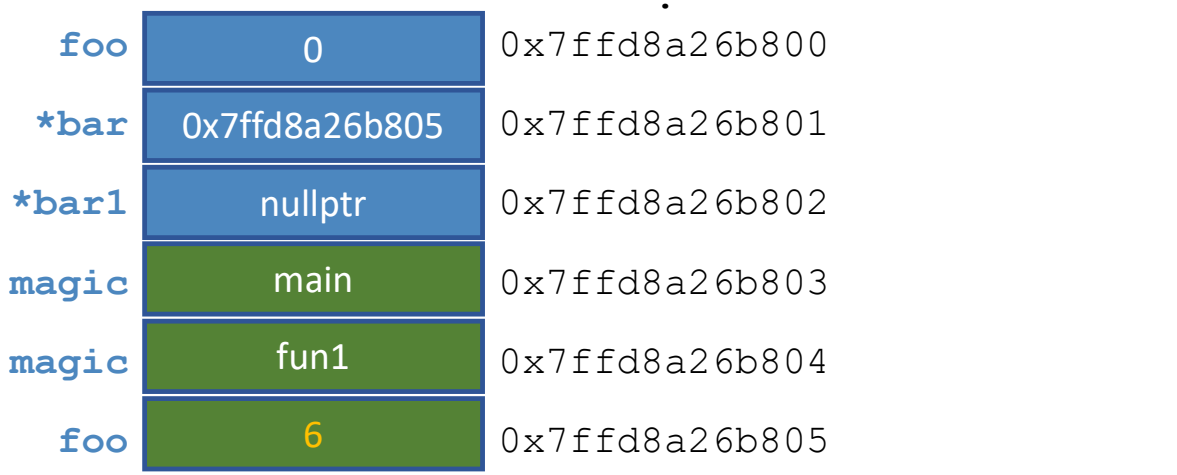

```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }                       1st
10  void fun2() {           2nd
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

Level	Precedence group	Operator	Description	Grouping
1	Scope	`::`	scope qualifier	Left-to-right
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right
		`()`	functional forms	
		`[]`	subscript	
		`. ->`	member access	
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left
		`~ !`	bitwise NOT / logical NOT	
		`+ -`	unary prefix	
		`& *`	reference / dereference	
		`new delete`	allocation / deallocation	
		`sizeof`	parameter pack	
		`(type)`	C-style type-casting	
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right
8	Relational	`< > <= >=`	comparison operators	Left-to-right
9	Equality	`== !=`	equality / inequality	Left-to-right
10	And	`&`	bitwise AND	Left-to-right
11	Exclusive or	`^`	bitwise XOR	Left-to-right
12	Inclusive or	`|`	bitwise OR	Left-to-right
13	Conjunction	`&&`	logical AND	Left-to-right
14	Disjunction	`||`	logical OR	Left-to-right
15	Assignment-level expressions	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment / compound assignment	Right-to-left
		`?:`	conditional operator	
16	Sequencing	`,`	comma separator	Left-to-right

Scope

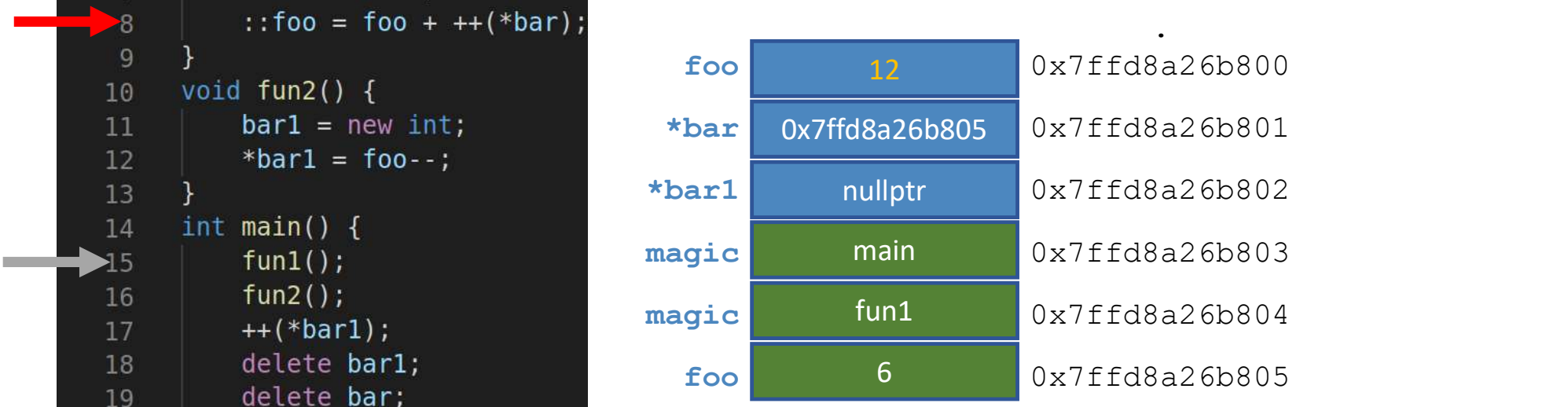

```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }                        1st
10  void fun2() {           2nd
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

- To access global variables, empty scope qualifier (::) prior to the variable name could be used.
- Prefix operators execute the arithmetic on the actual object and returns a reference of the object.



Scope

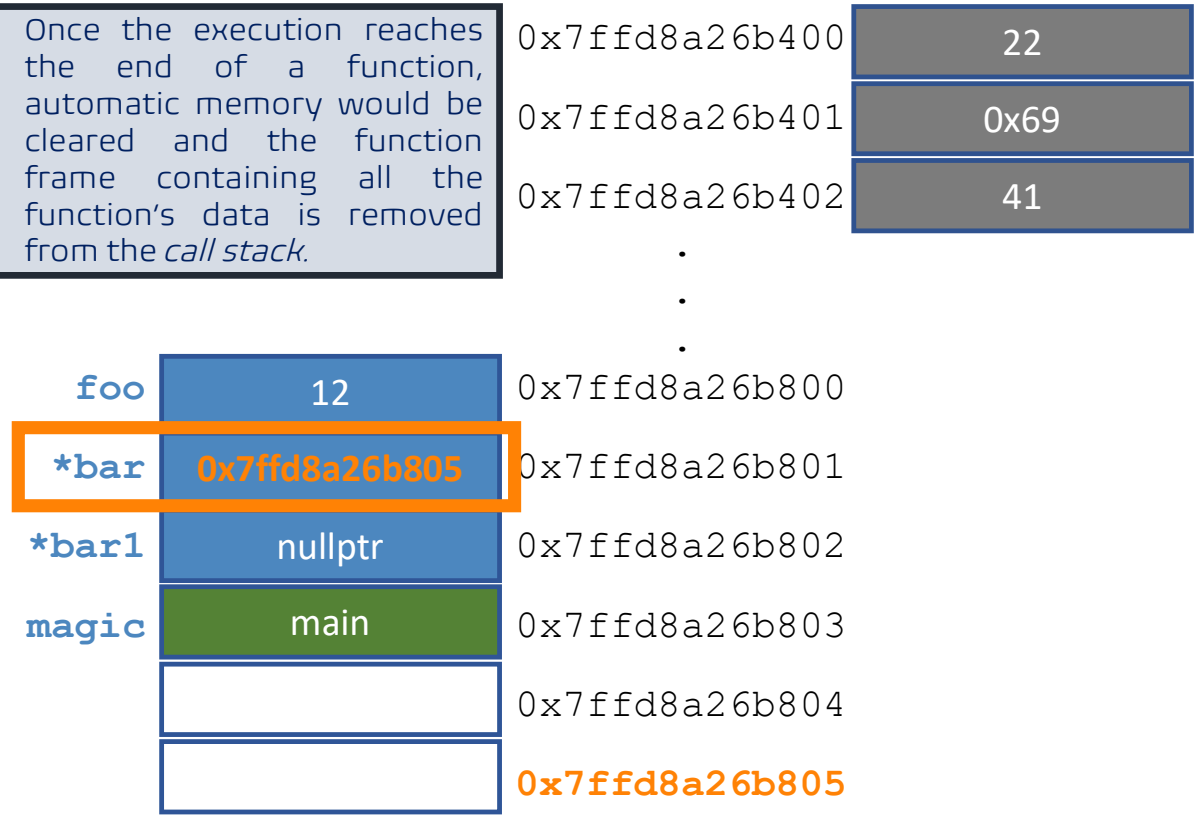

```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```



Scope

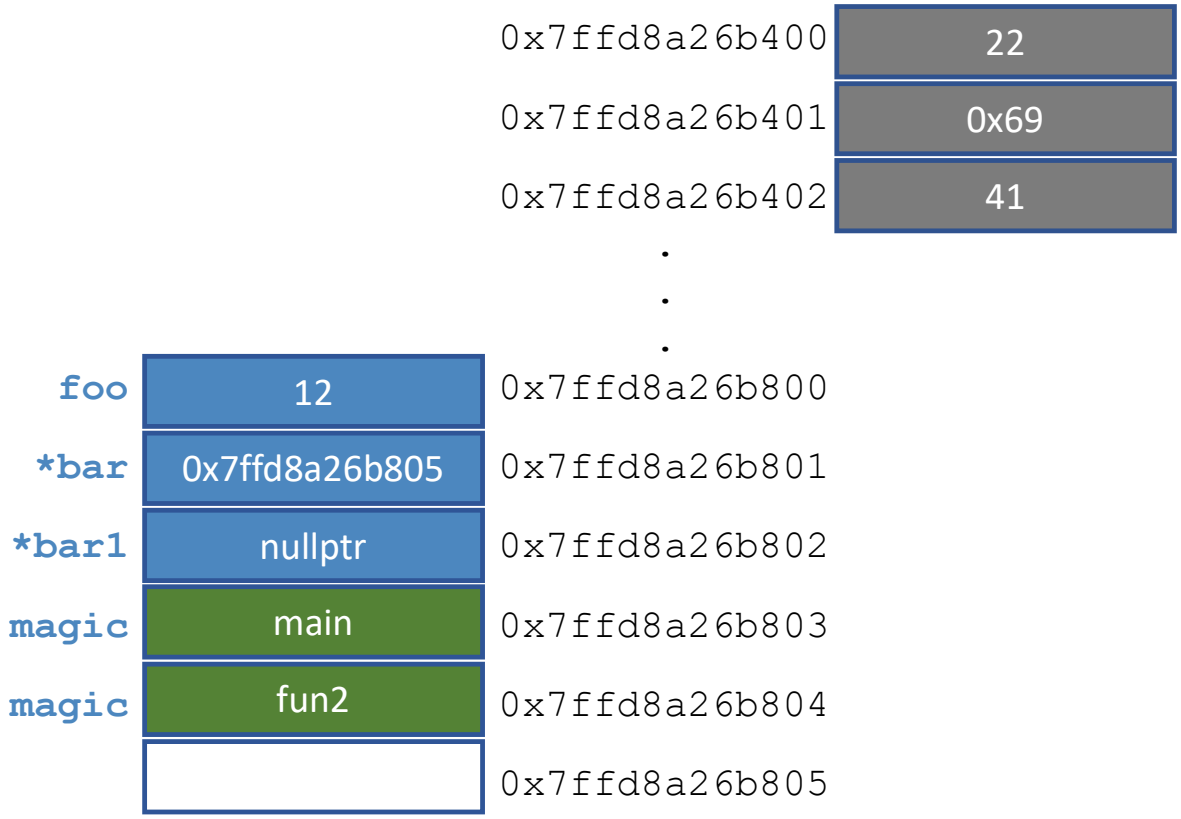

```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

- Once the execution reaches the end of a function, automatic memory would be cleared and the function frame containing all the function's data is removed from the *call stack*.



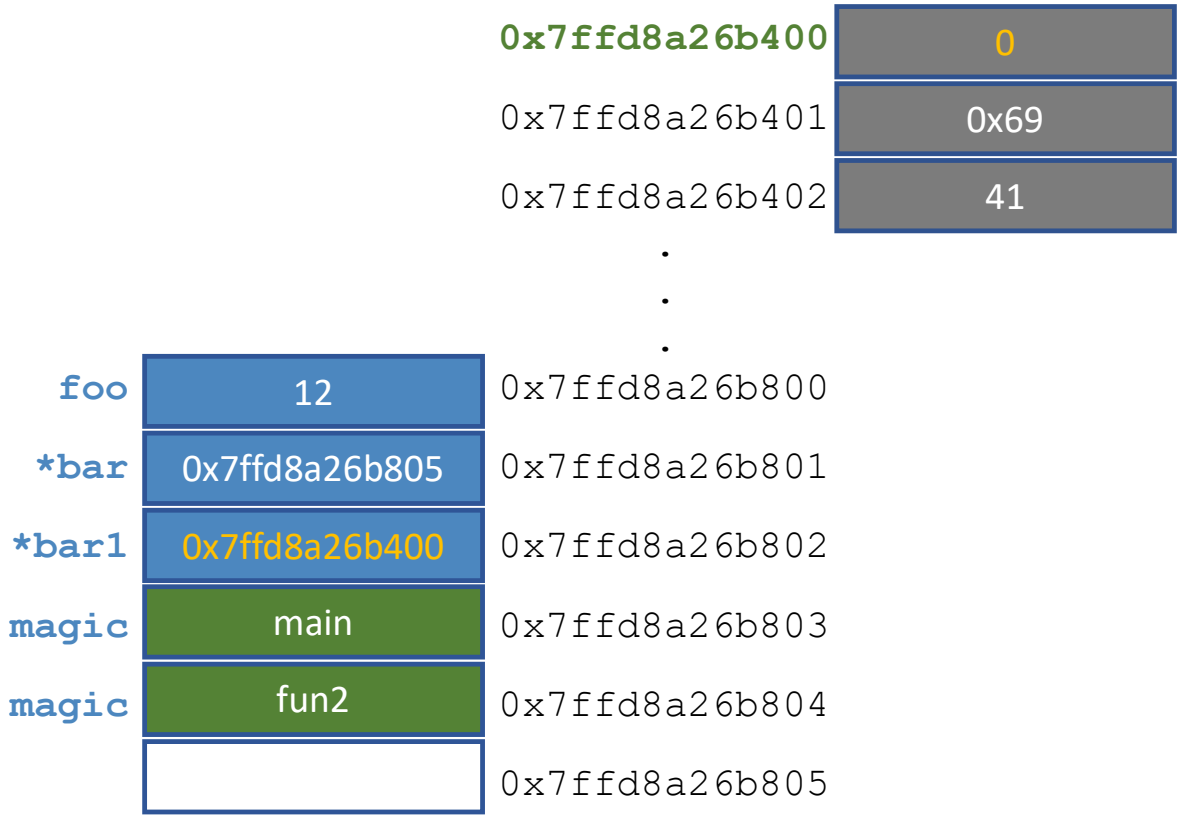
Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```



Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```



Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

2nd 1st

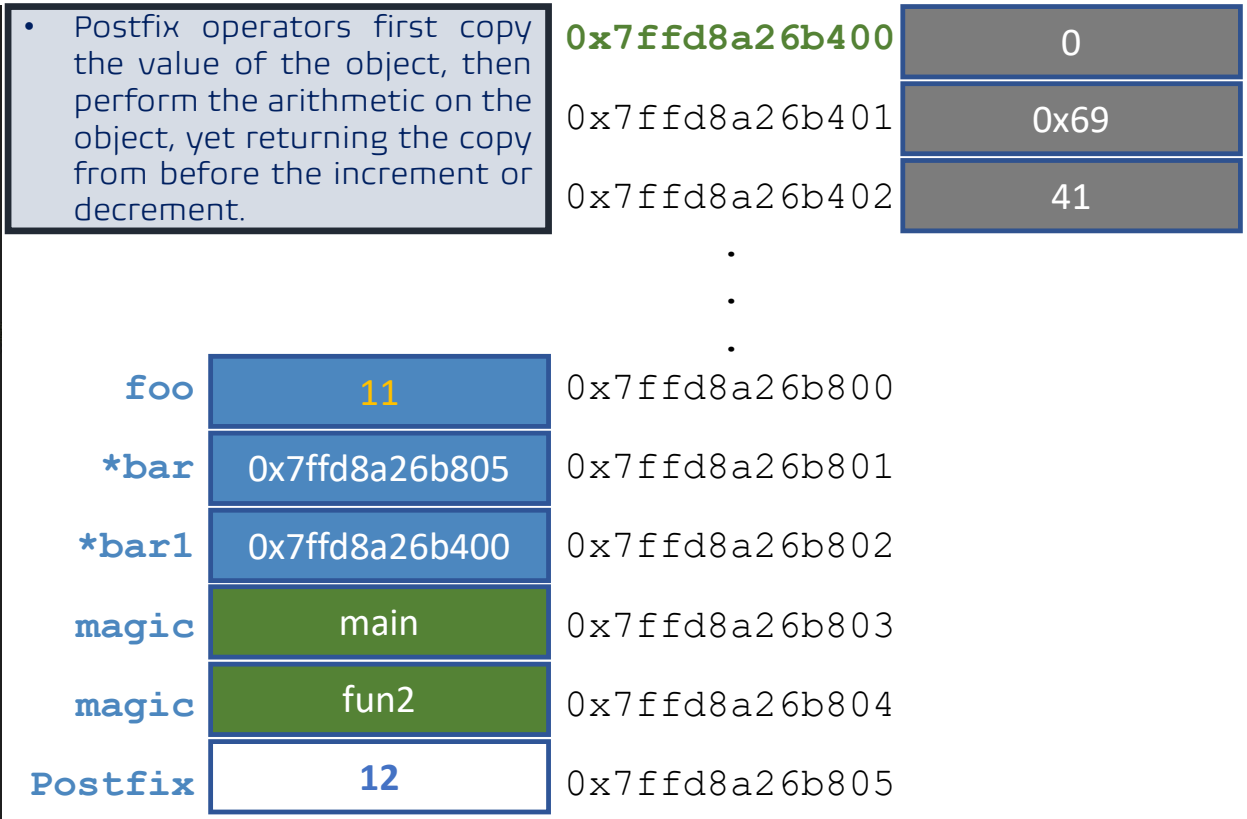
Level	Precedence group	Operator	Description	Grouping
1	Scope	`::`	scope qualifier	Left-to-right
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right
		`()`	functional forms	
		`[]`	subscript	
		`. ->`	member access	
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left
		`~ !`	bitwise NOT / logical NOT	
		`+ -`	unary prefix	
		`& *`	reference / dereference	
		`new delete`	allocation / deallocation	
		`sizeof`	parameter pack	
		`(type)`	C-style type-casting	
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right
8	Relational	`< > <= >=`	comparison operators	Left-to-right
9	Equality	`== !=`	equality / inequality	Left-to-right
10	And	`&`	bitwise AND	Left-to-right
11	Exclusive or	`^`	bitwise XOR	Left-to-right
12	Inclusive or	`|`	bitwise OR	Left-to-right
13	Conjunction	`&&`	logical AND	Left-to-right
14	Disjunction	`||`	logical OR	Left-to-right
15	Assignment-level expressions	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment / compound assignment	Right-to-left
		`?:`	conditional operator	
16	Sequencing	`,`	comma separator	Left-to-right

Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

1st

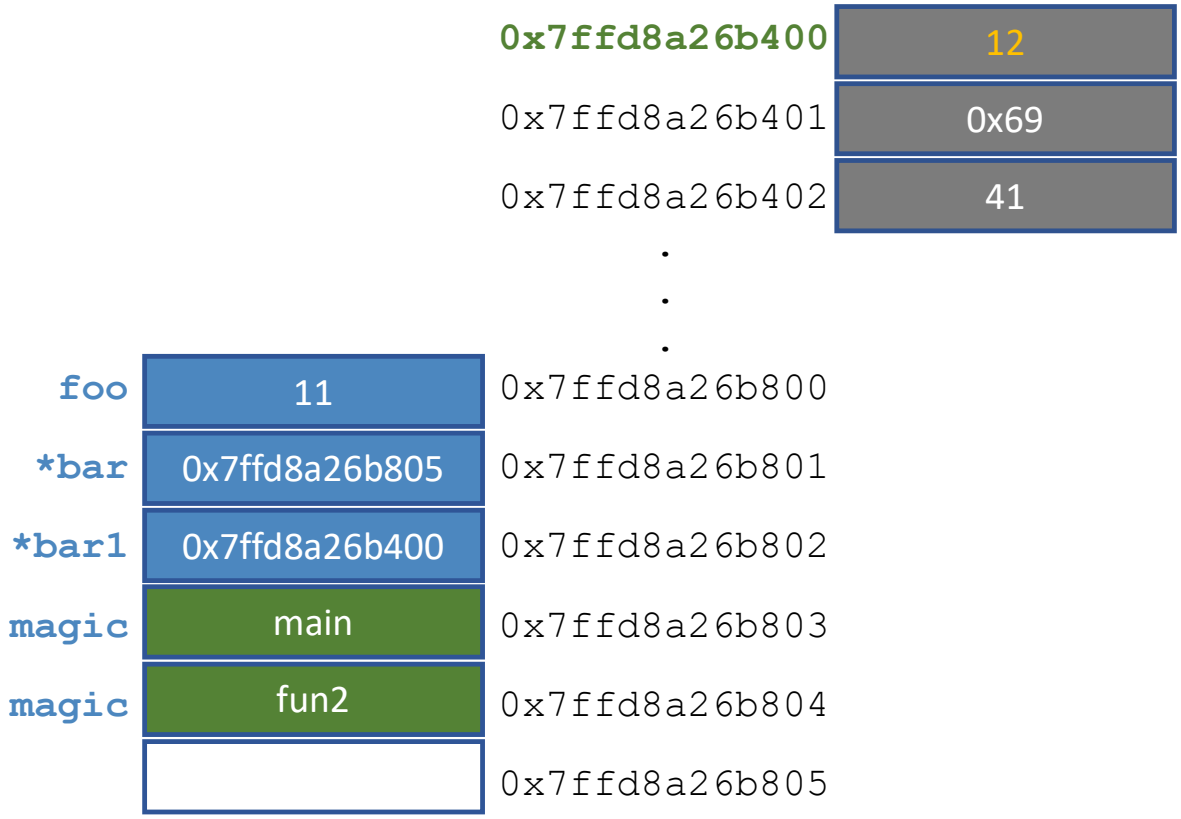
- Postfix operators first copy the value of the object, then perform the arithmetic on the object, yet returning the copy from before the increment or decrement.



Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

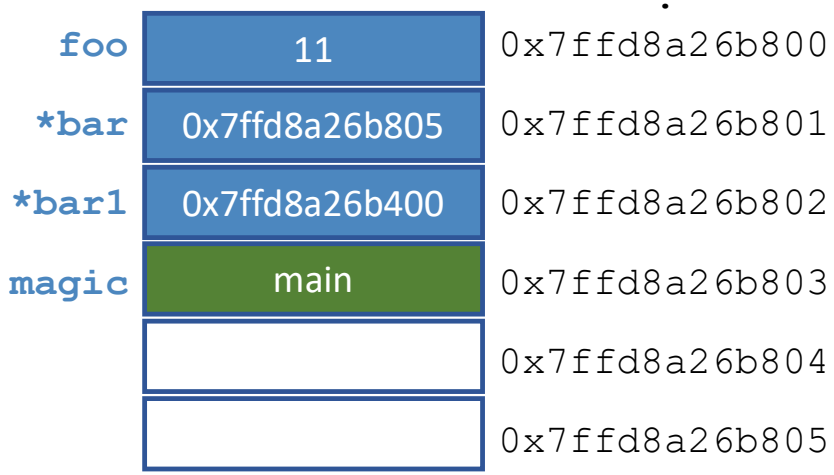
2nd



Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

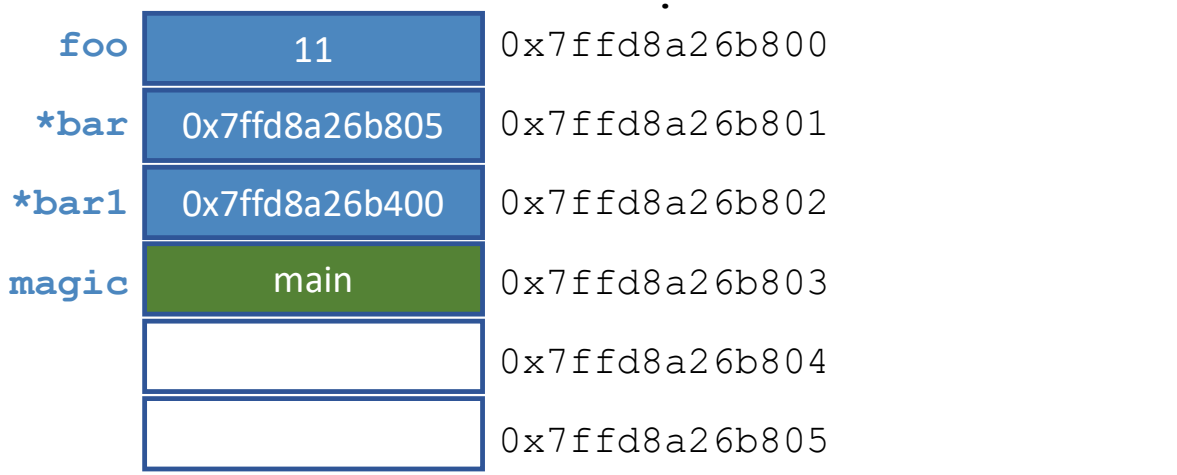
- Dynamic memory would not be cleaned up after execution leaves the function body and is still *"available"* for manipulation.



Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

- Dynamic memories would be available up until they are deleted. They are accessible from anywhere in the code where the address is known.



Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

- Deleting a pointer releases the memory cell the pointer value was pointing to; however, the pointer value would still be pointing to the same *"unavailable"* memory cell. Operations on *unavailable* regions which are uninitialized is not allowed.

foo	11	0x7ffd8a26b800
*bar	0x7ffd8a26b805	0x7ffd8a26b801
*bar1	0x7ffd8a26b400	0x7ffd8a26b802
magic	main	0x7ffd8a26b803
		0x7ffd8a26b804
		0x7ffd8a26b805

0x7ffd8a26b400	
0x7ffd8a26b401	0x69
0x7ffd8a26b402	41

- It is a common practice to assign value of `nullptr` to a deleted pointer to be able to recognized it is uninitialized if necessary, later in the code.

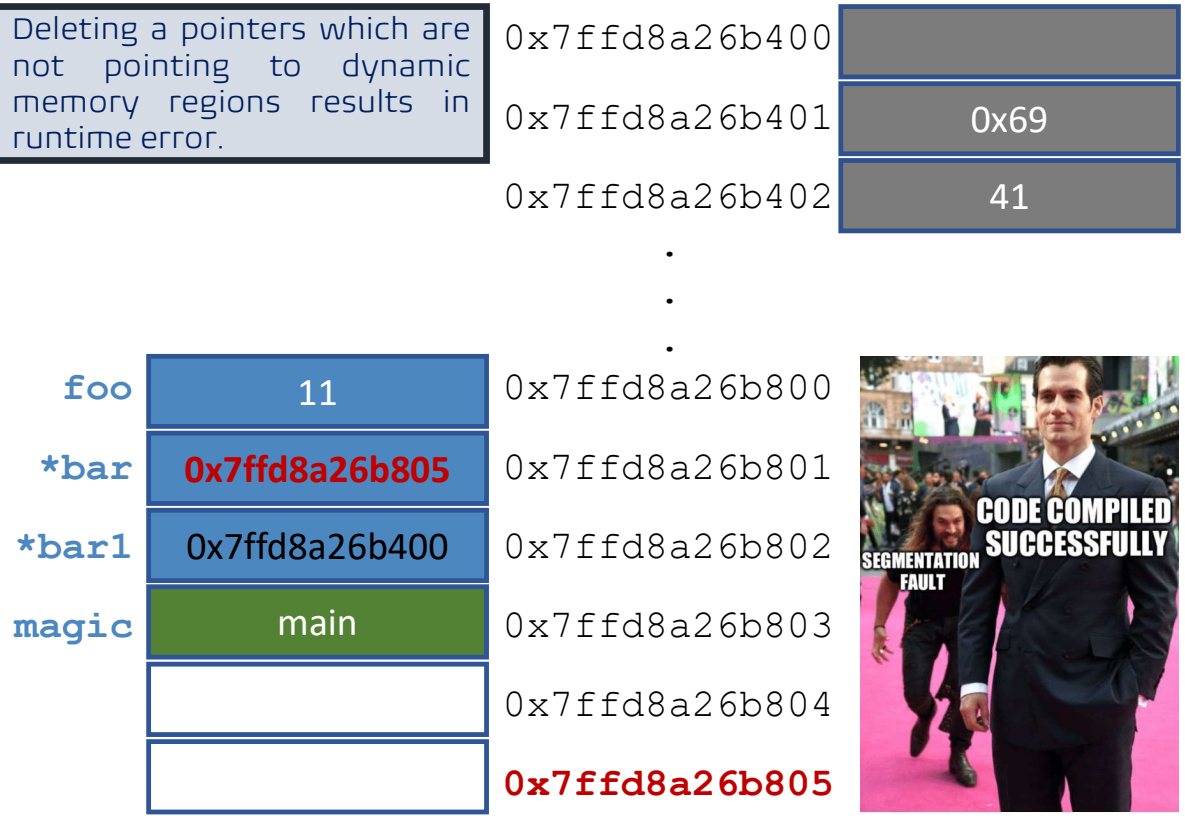
```
delete bar1;
bar1 = nullptr;
```

Scope


```
1   #include<iostream>
2   // Global
3   int foo, *bar, *bar1;
4   void fun1() {
5       int foo; //Local
6       foo   = 5;
7       bar   = &foo;
8       ::foo = foo + ++(*bar);
9   }
10  void fun2() {
11      bar1 = new int;
12      *bar1 = foo--;
13  }
14  int main() {
15      fun1();
16      fun2();
17      ++(*bar1);
18      delete bar1;
19      delete bar;
20      return 0;
21  }
```

free(): invalid pointer
Aborted (core dumped)

- Deleting a pointers which are not pointing to dynamic memory regions results in runtime error.



It happens every time...

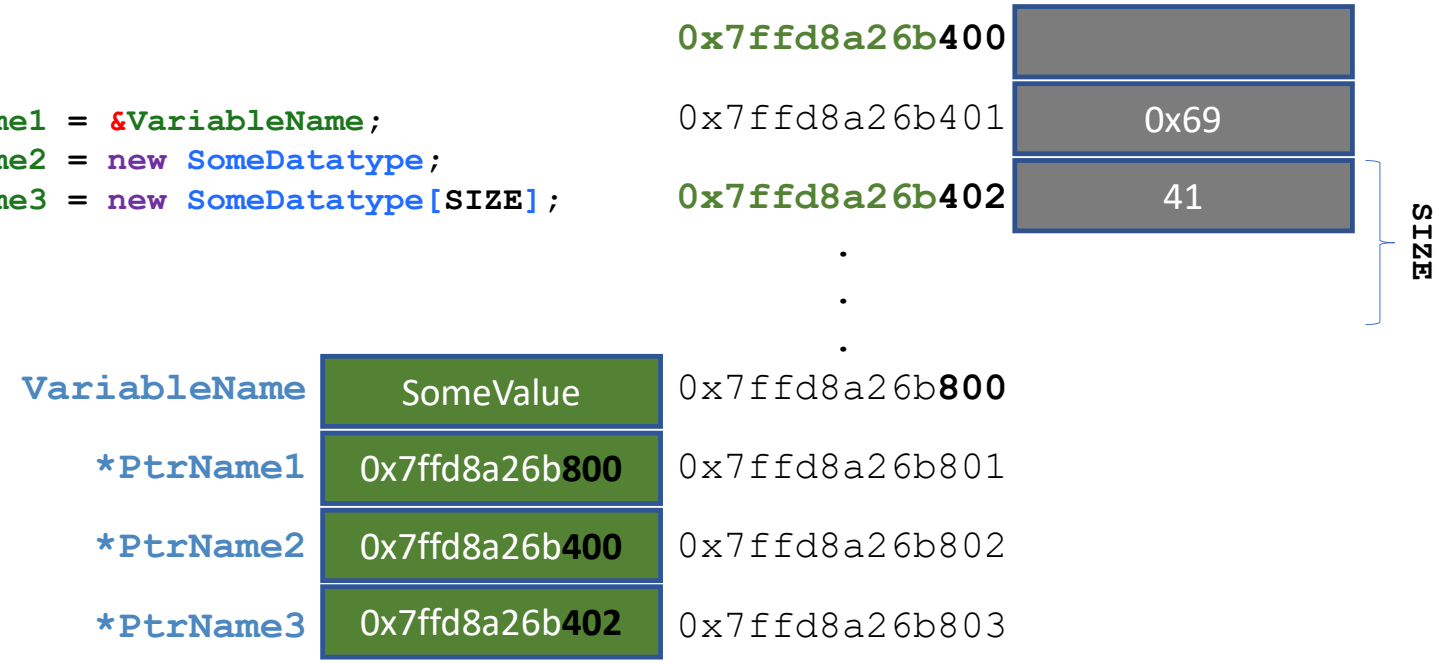
Pointers' Basics Summary

- More on pointers in future!

```
SomeDatatype *PtrName1 = &VariableName;
SomeDatatype *PtrName2 = new SomeDatatype;
SomeDatatype *PtrName3 = new SomeDatatype[SIZE];

.
.
.
.
.
.
.
.
.

delete[] PtrName3;
delete   PtrName2;
PtrName2 = PtrName3 = nullptr;
delete   PtrName1; // CRASHES!
```



- Pointers could point to an array of objects in dynamic memory.
- References could be used on any memory cells (static, automatic, and dynamic) to retrieve the address of the cell – the type is a pointer of the same datatype the memory cell holds.

struct

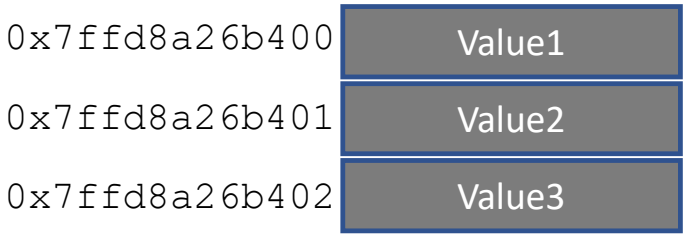
- **`struct`** is a type consisting of a sequence of *members* whose storage is allocated in an ***ordered sequence***.

```
struct helloworld_struct
{
    DataType1 Value1;
    DataType2 Value2;
    DataType3 Value3;
};
```

```
struct helloworld_struct HW_stu;

HW_stu.
        Value1                  DataType1 helloworl…
        Value2
        Value3
```

```
struct helloworld_struct *HW_stu_ptr = new struct helloworld_struct;
HW_stu_ptr->
            Value1                  DataType1 helloworl…
            Value2
            Value3
```

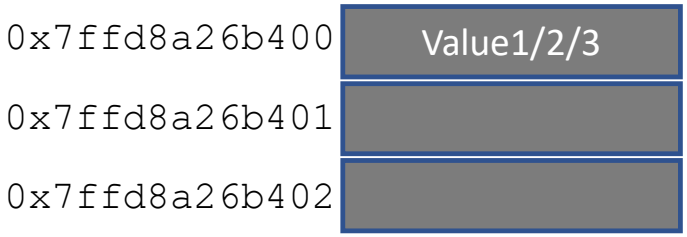


- **`struct`** when defined is considered a *compound* datatype and like other datatypes, **`struct`** could be allocated in *static*, *automatic* or *dynamic* memory.
- In case **`struct`** is defined in static or automatic memory, its *members* could be accessed using dot '**`.`**' access operator.
- In order to access *members* of **`struct`** declared in dynamic memory '**`->`**' access operator is used.

union

- **`union`** is a type consisting of a sequence of *members* whose storage ***overlaps***.

```
union helloworld_union
{
    DataType1 Value1;
    DataType2 Value2;
    DataType3 Value3;
};
```



- At most, only one of the *members* could be accessed/stored at any one time.
- Size of **`union`** is equal to the size of its biggest member, while size of **`struct`** is at least sum of all its member.
- A pointer to **`union`** could be *cast* to the datatype of any of its members, in oppose to a pointer of **`struct`** which could only be *cast* to its first member.

- Similar to **`struct`**s, **`union`**s are also considered a *compound* datatype upon definition and like other datatypes, they could be allocated in *static*, *automatic* or *dynamic* memory.
- Access operators are the same for **`union`** -- '**`.`**' access operator for objects in *static* and *automatic* memory, and '**`->`**' for objects stored in *dynamic* memory.

enum

ALTEN

- Enumeration is a type whose value is restricted to a range of values i.e. named constants called *enumerators*.

```
enum WeekDays {
  Sat, Sun, Mon, Tue, Wed, Thu, Fri
};

WeekDays today = WeekDays::Tue;
```

- The actual value of the named constants or *enumerators* are of an integral type.

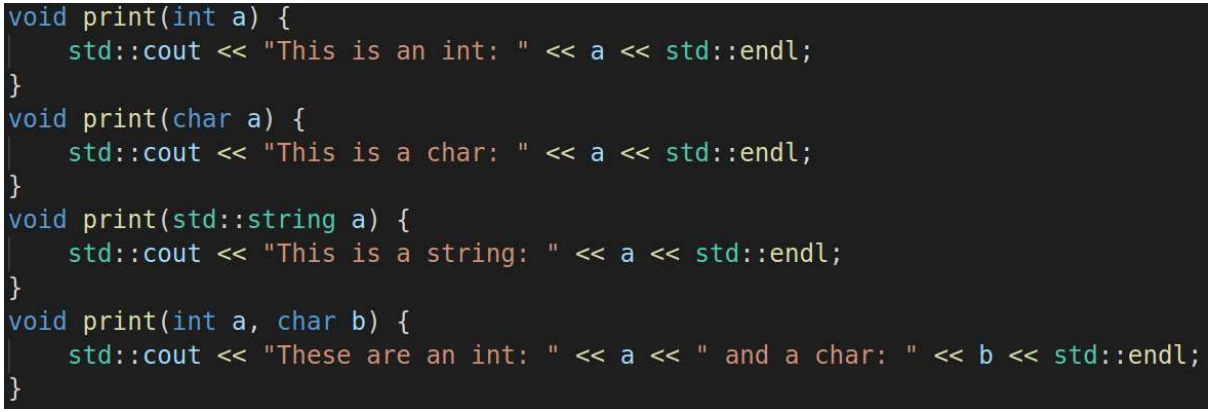
```
void celebrateTuesdays(WeekDays day) {
  switch (day)
  {
    case WeekDays::Tue: {
      partyHard();
      break;
    }
    default: {
      std::cout <<
      "It is not the time yet!"
      << std::endl;
      break;
    }
  }
};
```

- Enumeration is a datatype and variables of its type could be used and allocated similar to other datatypes.
- Enumeration values are accessed using `::`.

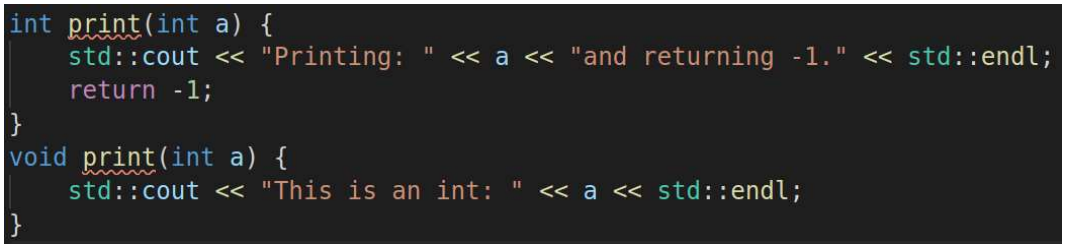
```
WeekDays *day = new WeekDays;
*day = WeekDays::Tue;
```

DEMO!

Function Overloading

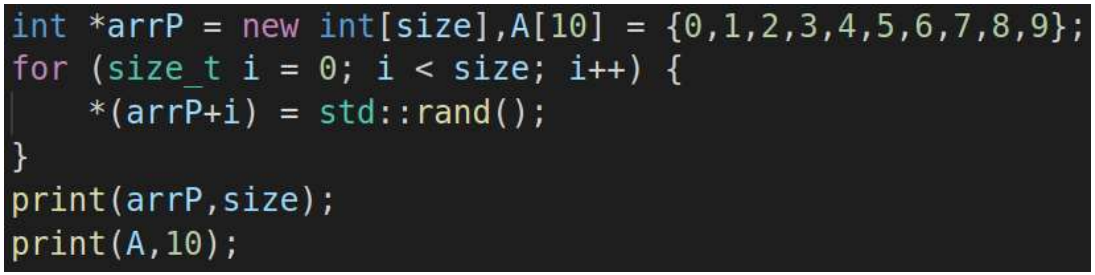


```
void print(int a) {
    std::cout << "This is an int: " << a << std::endl;
}
void print(char a) {
    std::cout << "This is a char: " << a << std::endl;
}
void print(std::string a) {
    std::cout << "This is a string: " << a << std::endl;
}
void print(int a, char b) {
    std::cout << "These are an int: " << a << " and a char: " << b << std::endl;
}
```



```
int print(int a) {
    std::cout << "Printing: " << a << "and returning -1." << std::endl;
    return -1;
}
void print(int a) {
    std::cout << "This is an int: " << a << std::endl;
}
```

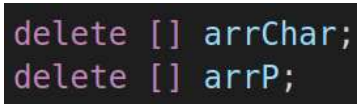
Pointers Are Arrays



```
int *arrP = new int[size],A[10] = {0,1,2,3,4,5,6,7,8,9};
for (size_t i = 0; i < size; i++) {
    *(arrP+i) = std::rand();
}
print(arrP,size);
print(A,10);
```

```
char *arrChar = reinterpret_cast<char*> (arrP);
```

```
void print (char *arr, size_t s)
```



```
delete [] arrChar;
delete [] arrP;
```

Reference vs Pointers

ALTEN

"reference ***is less powerful but safer*** *than the pointer"* https://en.wikipedia.org/wiki/Reference_(C%2B%2B)

```
int &f1() {
    int a = 2;
    return a;
}

int &f1(int &a) {
    return ++a;
}

int *f2() {
    int a = 2;
    return &a;
}

int *f3() {
    int *a = new int;
    *a = 3;
    return a;
}
```

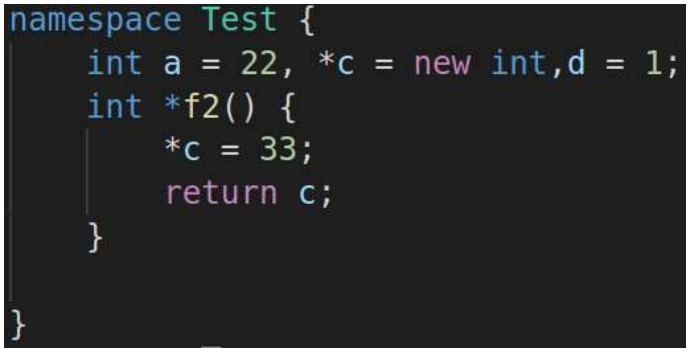
```
/*
int *&f4(int *a) {
    *a = 7;
    return a;
}
*/
int *&f4(int *&a) {
    *a = 7;
    return a;
}
/*
int &*f5(int &*a) {
    *a = 7;
    return a;
}
*/
```

```
void f4 (int *a) {
    (*a)++;
}
void f5 (int * const a) {
    (*a)++;
}
void f6 (int const * a) {
    (*a)++;
}
void f7 (const int * a) {
    (*a)++;
}
```

```
void f1 (int a) {
    a ++;
}
void f1 (int &a) {
    a ++;
}
void f2 (int &a) {
    a ++;
}
void f3 (const int &a) {
    a ++;
}
void f3 (int & const a) {
    a++;
}
```

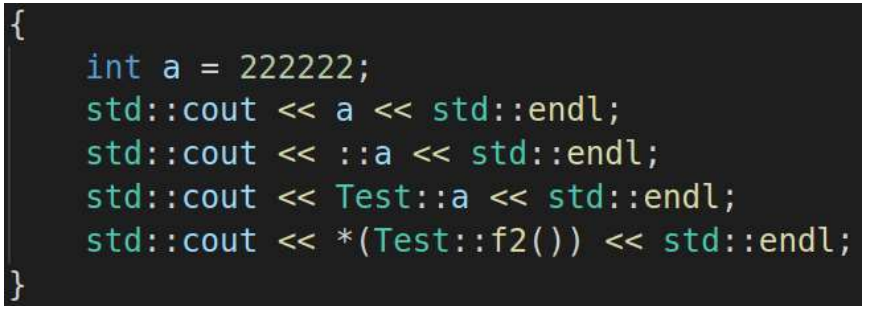
- The keyword const is read *clockwise:* http://c-faq.com/decl/spiral.anderson.html
- References cannot be constant: https://stackoverflow.com/questions/38044834/why-are-references-not-const-in-c/38044974

Scope

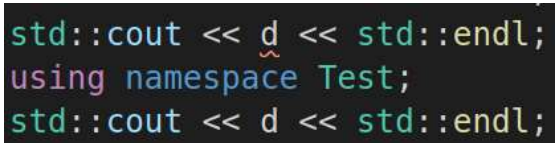


```
namespace Test {
    int a = 22, *c = new int,d = 1;
    int *f2() {
        *c = 33;
        return c;
    }

}
```



```
{
    int a = 222222;
    std::cout << a << std::endl;
    std::cout << ::a << std::endl;
    std::cout << Test::a << std::endl;
    std::cout << *(Test::f2()) << std::endl;
}
```



```
std::cout << d << std::endl;
using namespace Test;
std::cout << d << std::endl;
```

struct

ALTEN

```
typedef struct struct_c{
    void fun();
}C;

typedef struct
{
    void fun() {
        std::cout << "This is D!" << std::endl;
    }
}D;
```

```
void struct_c::fun() {
    std::cout << "This is C!" << std::endl;
}
```

```
C c;
D *d = new D;
c.fun();
d->fun();
```

```
struct struct_a {
    char a;
    char b;
    char c;
    char d = 5;
};

typedef struct_a A;
struct struct_b {
    int a = 0;
};
//typedef struct_b B;
void struct_b();
```

```
//struct_b b;
A a;
struct struct_b b;
struct_a a2;

//AA aaa;
```

```
struct struct_aa
{
    char a;
    char b;
    char c;
    char d = 50;
}AA,BB,CC;
```

```
f0(a);
f0(reinterpret_cast<A*>(&b));
```

```
}
        struct struct_a
void f0(struct_a a) {
    std::cout << a.a << "
}

void f0(struct_a *a) {
    std::cout << a->a << "
}
```

union

ALTEN

- **union**s are identical to **struct**s syntax wise; the only difference is the *storage* for union is *overlapping*!

```cpp
#include <iostream>
union uni {
    int a;
    char c[4];
};
int main() {
    std::cout << sizeof(union uni) << std::endl;
    uni a;
    a.a = -300;
    std::cout << "This is a: " << a.a  << std::endl;
    std::cout << "These are: ";
    for (size_t i = 0; i < 4; i++) {
        std::cout << "c[" << i << "]=" << static_cast<int>(a.c[i]) << " ";
    }
    std::cout << std::endl;
    return 0;
};
```

enum


```
enum Mode {INT, DBL};
enum Type {A, B};

typedef struct _a{
    Mode mode;
```

```
void PrintA(stuA *a);
void PrintB(stuB *b);
void print(void *_p, Type _t) {
    if (_t == Type::A) {
        PrintA(reinterpret_cast<stuA*>(_p));
    } else if (_t == Type::B) {
        PrintB(reinterpret_cast<stuB*>(_p));
    }
}
```