C++ Software Engineering

for engineers of other disciplines

RECAP 00

ALTEN

Autumn 2021

Gothenburg, Sweden

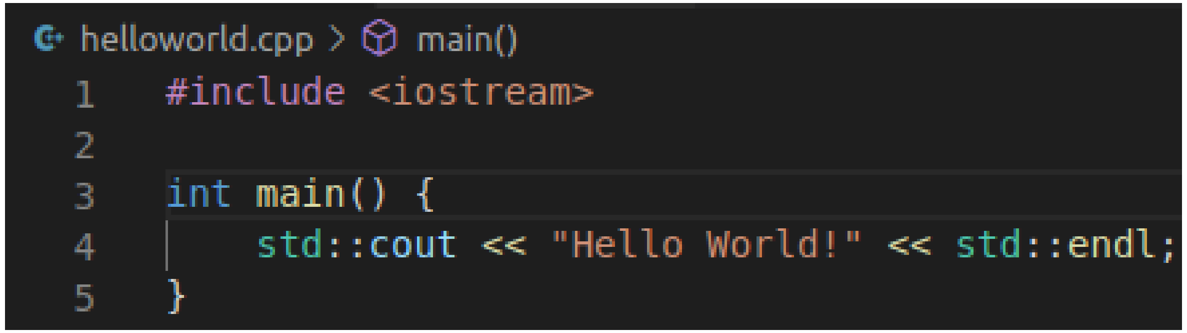
petter.lerenius@alten.se

rashid.zamani@alten.se

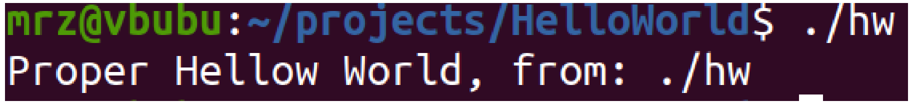
Proper Hello World!

- **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**

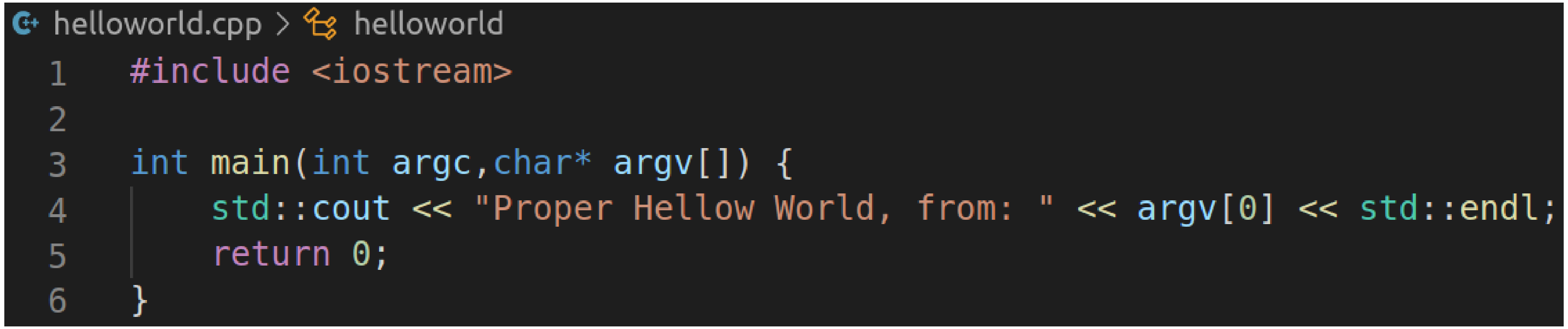
- **main**'s return value could be captured from terminal when executing the program – zero usually indicates normal termination.
- **argc** (argument count) and **argv** (argument values) are conventional names which are used universally for better readability, otherwise they could be named anything.



```cpp
#include <iostream>

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



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



```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.

C++ Keywords

- Fundamental Datatypes
- Control Flows: Selections, Iteration Statements & Jump Statements

alignas, alignof, and, and_eq, asm, auto, bitand,
bitor, **bool**, **break**, **case**, catch, **char**, **char16_t**,
char32_t, class, compl, **const**, constexpr,
const_cast, **continue**, **decltype**, **default**, delete, **do**,
double, dynamic_cast, **else**, enum, explicit, export,
extern, **false**, **float**, **for**, friend, **goto**, **if**, inline,
int, **long**, mutable, namespace, new, noexcept, not,
not_eq, **nullptr**, operator, or, or_eq, private,
protected, public, register, reinterpret_cast,
return, **short**, **signed**, **sizeof**, static,
static_assert, static_cast, struct, **switch**,
template, this, thread_local, throw, **true**, try,
typedef, typeid, typename, union, **unsigned**, **using**,
virtual, **void**, volatile, **wchar_t**, **while**, xor, xor_eq

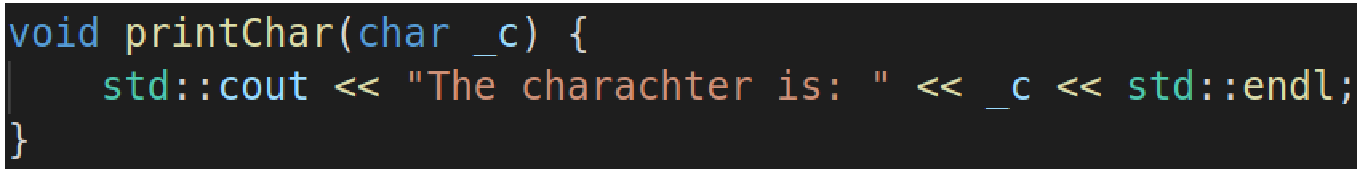
- Identifiers created by developers shall not match these keywords.
- Different compilers might add specific keywords.
- C++ is **case-sensitive**.

Functions

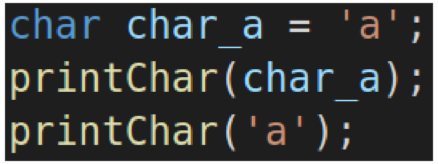
- Functions are used to structure the code.
- Basic function declaration is as follows:

```
ReturnDatatype FunName(InputDatatype Input1_Name …){
    // FUNCTION BODY
}
```

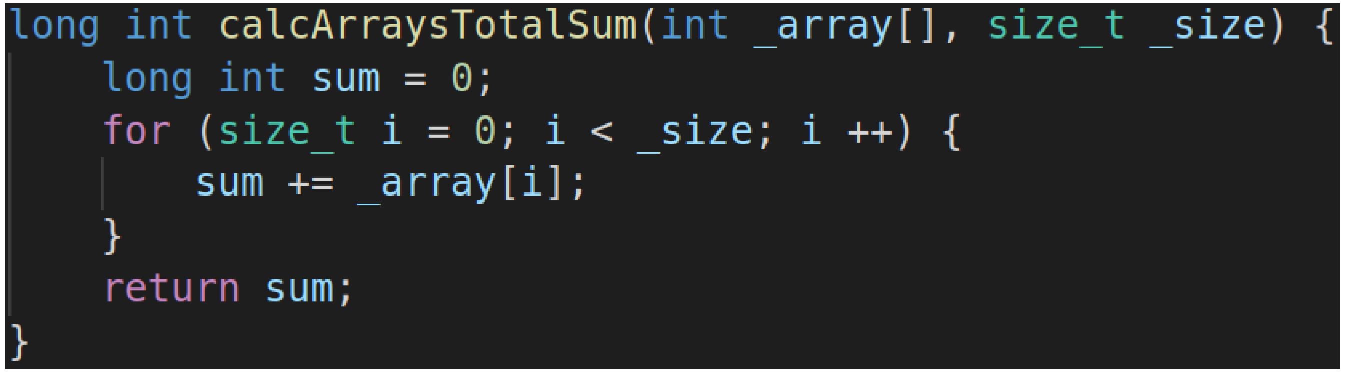
- **void** is used as a return datatype for functions without a return value.
- **main** if used as function name, define the entry point of the program i.e. is the first (only) function being invoked when program executed. If a branch of code is not accessible from the body of the main function, it will not be invoked throughout the execution.
- **size_t** is the same as **unsigned long int** -- it is the *defacto* type used for size. The definition is declared using **#define** preprocessing directive.



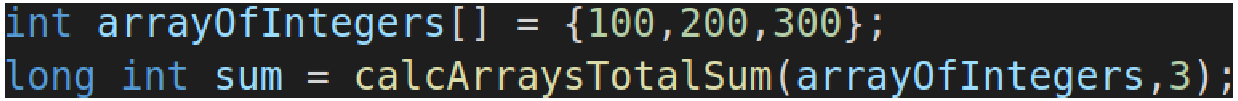
```
void printChar(char _c) {
    std::cout << "The charachter is: " << _c << std::endl;
}
```



```
char char_a = 'a';
printChar(char_a);
printChar('a');
```



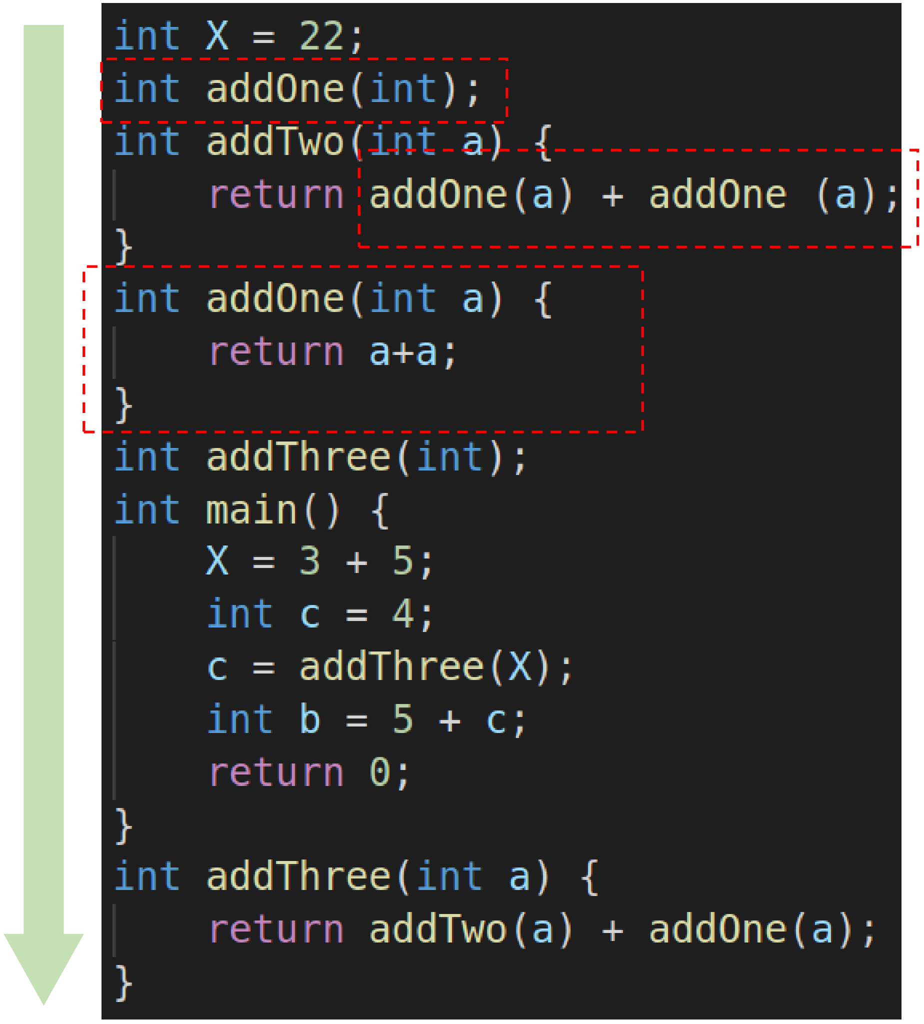
```
long int calcArraysTotalSum(int _array[], size_t _size) {
    long int sum = 0;
    for (size_t i = 0; i < _size; i ++) {
        sum += _array[i];
    }
    return sum;
}
```



```
int arrayOfIntegers[] = {100,200,300};
long int sum = calcArraysTotalSum(arrayOfIntegers,3);
```

Order

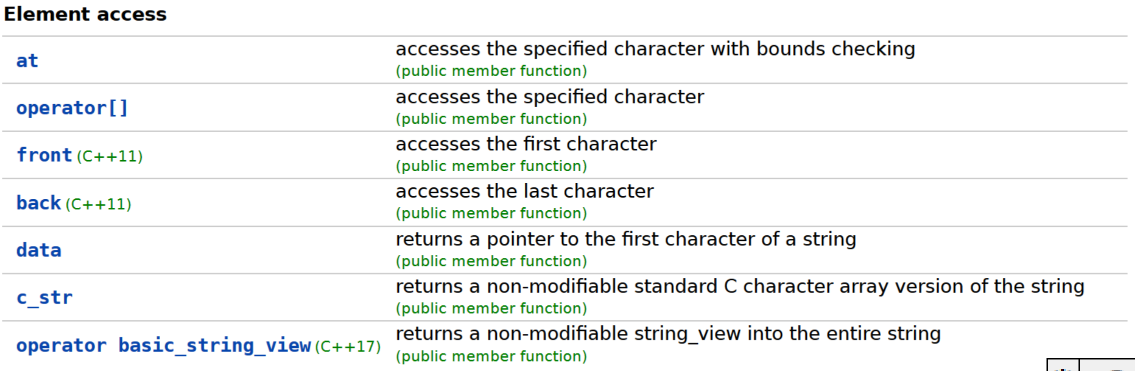
- Visibility order is downwards.
- Forward declaration could be used.



```
int X = 22;
int addOne(int);
int addTwo(int a) {
    return addOne(a) + addOne (a);
}
int addOne(int a) {
    return a+a;
}
int addThree(int);
int main() {
    X = 3 + 5;
    int c = 4;
    c = addThree(X);
    int b = 5 + c;
    return 0;
}
int addThree(int a) {
    return addTwo(a) + addOne(a);
}
```

Arrays & Strings

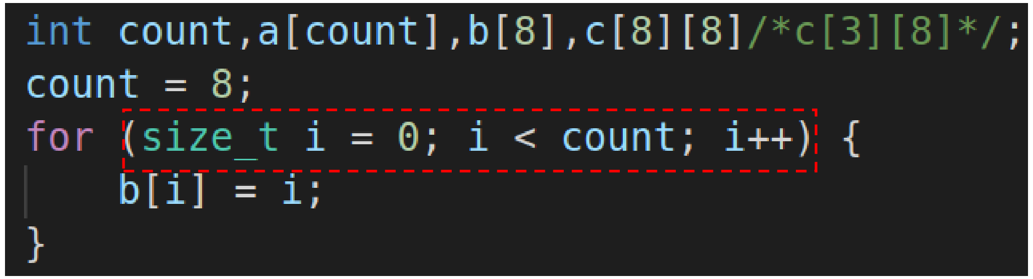
- There is no check on arrays’ boundaries.
- Strings are ASCI code null terminated.
- Standard library provides `string`.



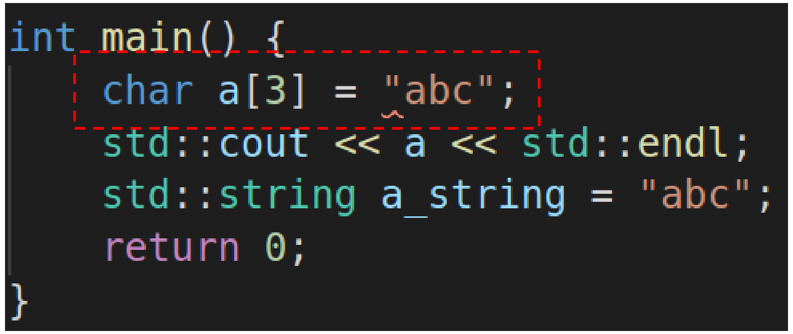
Element access

`at`	accesses the specified character with bounds checking (public member function)
`operator[]`	accesses the specified character (public member function)
`front` (C++11)	accesses the first character (public member function)
`back` (C++11)	accesses the last character (public member function)
`data`	returns a pointer to the first character of a string (public member function)
`c_str`	returns a non-modifiable standard C character array version of the string (public member function)
`operator basic_string_view` (C++17)	returns a non-modifiable string_view into the entire string (public member function)

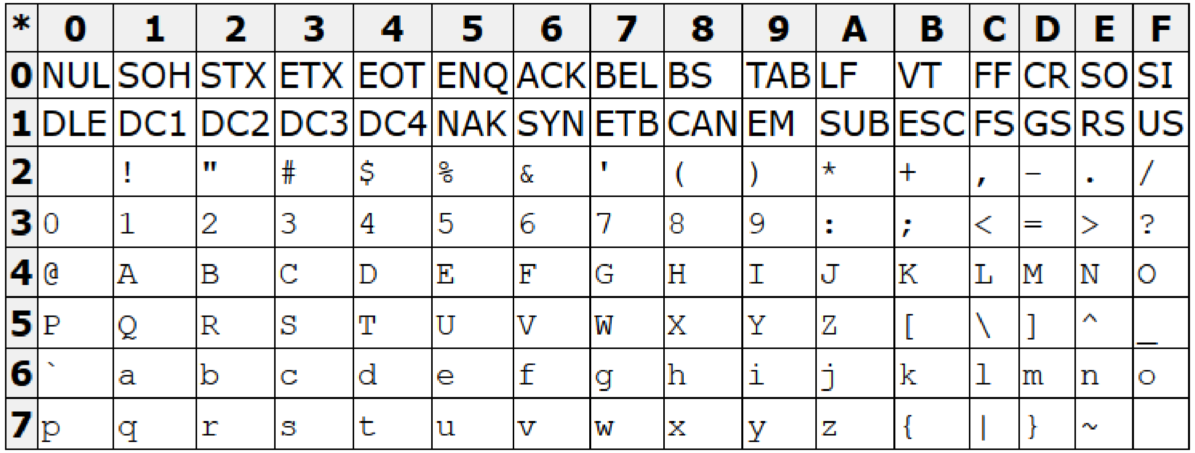
https://en.cppreference.com/w/cpp/string/basic_string



```
int count,a[count],b[8],c[8][8]/*c[3][8]*/;
count = 8;
for (size_t i = 0; i < count; i++) {
    b[i] = i;
}
```



```
int main() {
    char a[3] = "abc";
    std::cout << a << std::endl;
    std::string a_string = "abc";
    return 0;
}
```

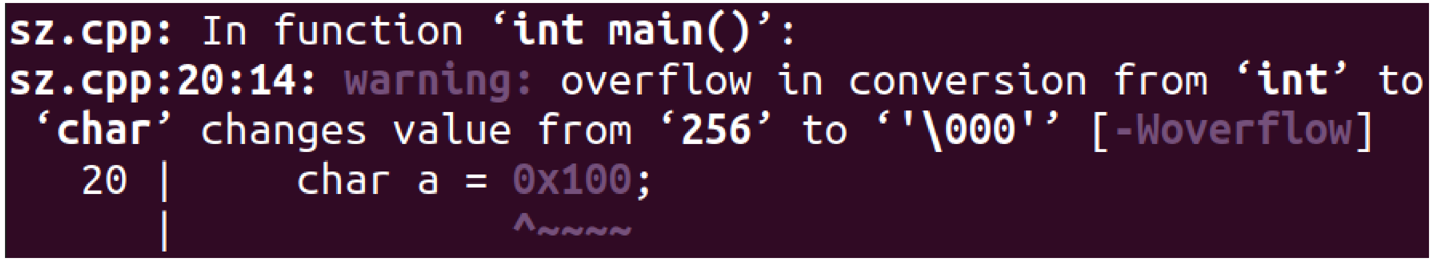


*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

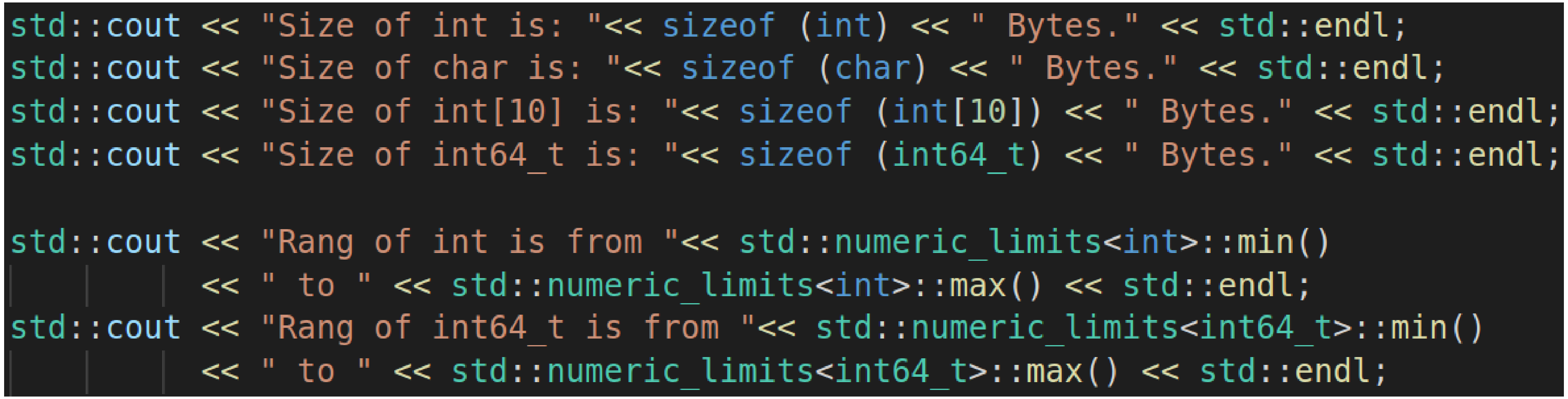
http://www.cplusplus.com/doc/ascii/

Size, Range & Overflow

- Fundamental datatypes have a range.
- Value higher than range is *overflow*.



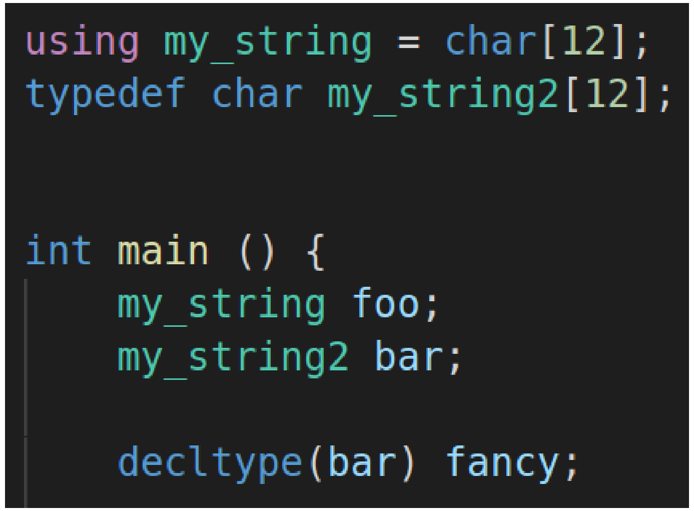
```
sz.cpp: In function ‘int main()’:
sz.cpp:20:14: warning: overflow in conversion from ‘int’ to
 ‘char’ changes value from ‘256’ to ‘'\000'’ [-Woverflow]
   20 |     char a = 0x100;
      |              ^~~~~
```



```
std::cout << "Size of int is: "<< sizeof (int) << " Bytes." << std::endl;
std::cout << "Size of char is: "<< sizeof (char) << " Bytes." << std::endl;
std::cout << "Size of int[10] is: "<< sizeof (int[10]) << " Bytes." << std::endl;
std::cout << "Size of int64_t is: "<< sizeof (int64_t) << " Bytes." << std::endl;

std::cout << "Rang of int is from "<< std::numeric_limits<int>::min()
          << " to " << std::numeric_limits<int>::max() << std::endl;
std::cout << "Rang of int64_t is from "<< std::numeric_limits<int64_t>::min()
          << " to " << std::numeric_limits<int64_t>::max() << std::endl;
```

Types

- Type alias is with `using` keyword.
- With `typedef` you can create synonym for a type.
- It is also possible to detect type of a variable at run time using `decltype`.

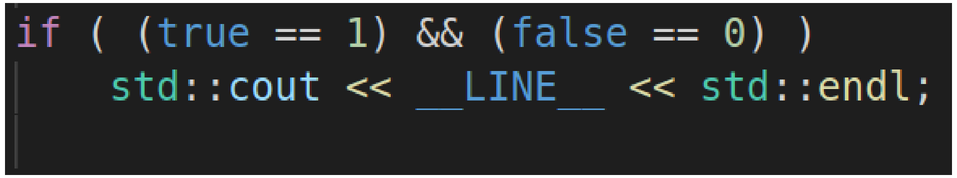


```
using my_string = char[12];
typedef char my_string2[12];


int main () {
    my_string foo;
    my_string2 bar;

    decltype(bar) fancy;
```

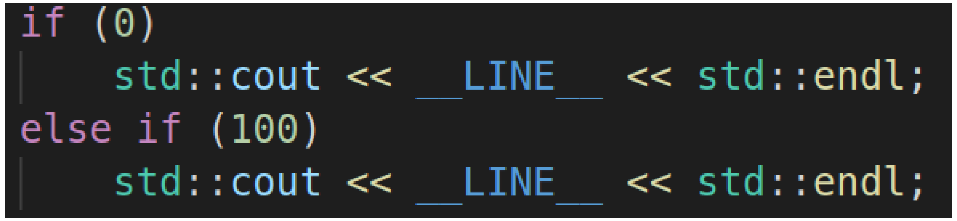
if

Boolean Value	Operand	Boolean Value	Result
true	&&	true	true
true	&&	false	false
false	&&	false	false
false	&&	true	false
true	||	true	true
true	||	false	true
false	||	false	false
false	||	true	true

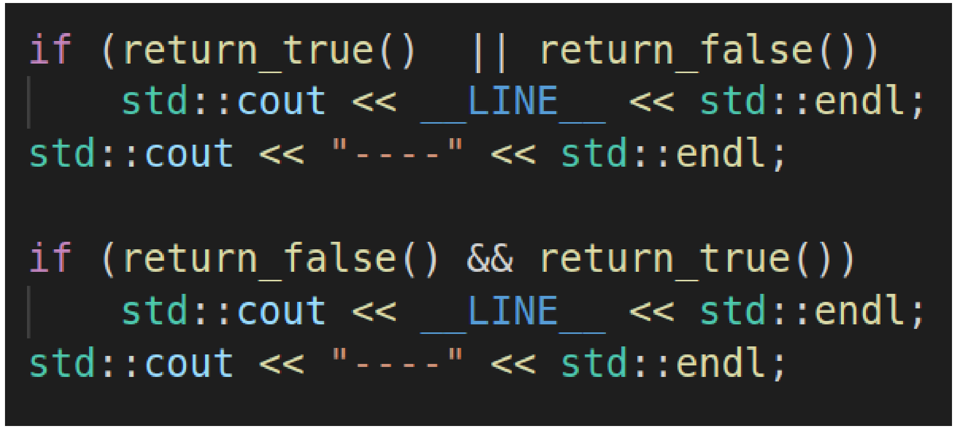
- **__LINE__** is a preprocessor Macro which *expands* to the line number. There are other useful Macros as well: https://stackoverflow.com/a/2849850



```
if ( (true == 1) && (false == 0) )
    std::cout << __LINE__ << std::endl;
```

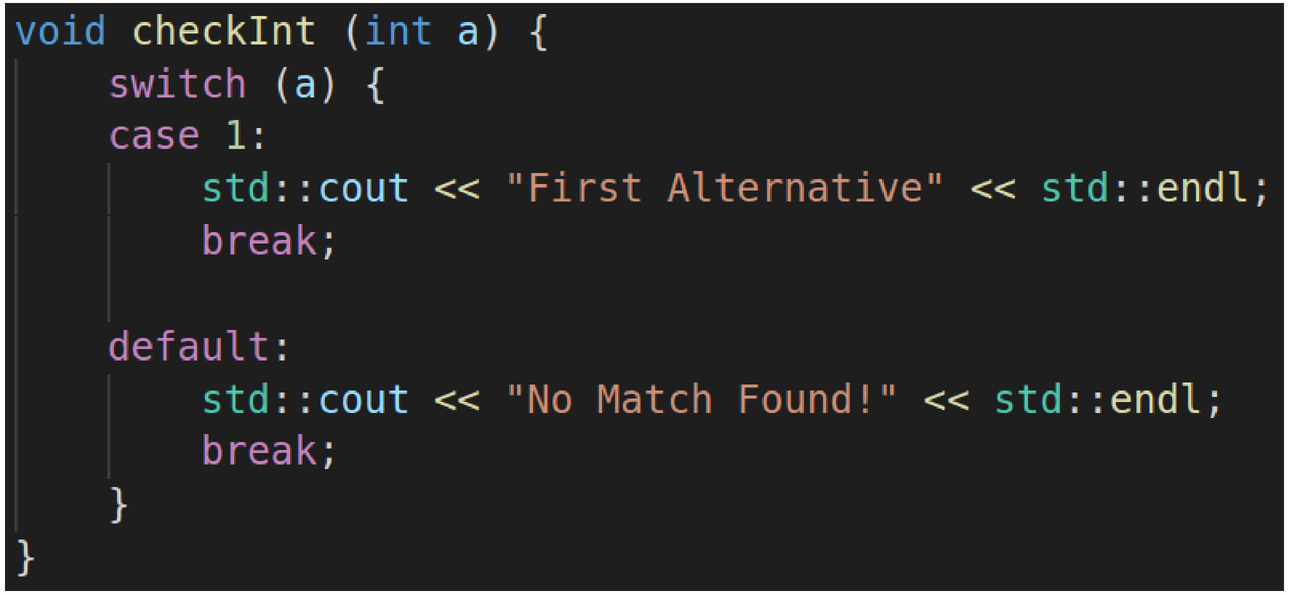


```
if (0)
    std::cout << __LINE__ << std::endl;
else if (100)
    std::cout << __LINE__ << std::endl;
```

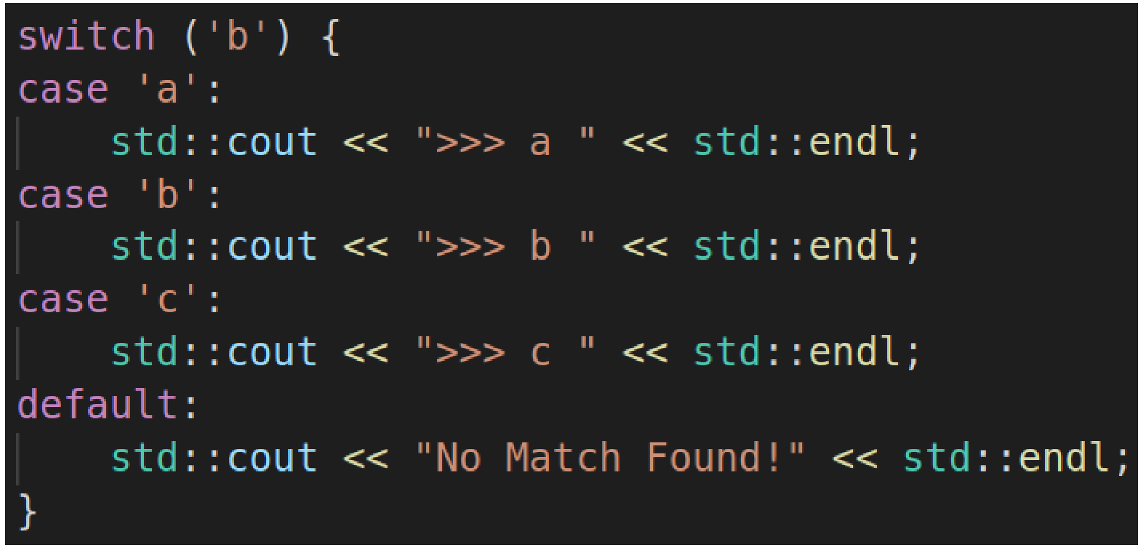


```
if (return_true()  || return_false())
    std::cout << __LINE__ << std::endl;
std::cout << "----" << std::endl;

if (return_false() && return_true())
    std::cout << __LINE__ << std::endl;
std::cout << "----" << std::endl;
```

switch

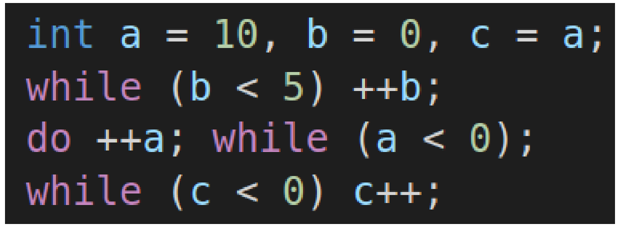


```cpp
void checkInt (int a) {
    switch (a) {
    case 1:
        std::cout << "First Alternative" << std::endl;
        break;

    default:
        std::cout << "No Match Found!" << std::endl;
        break;
    }
}
```

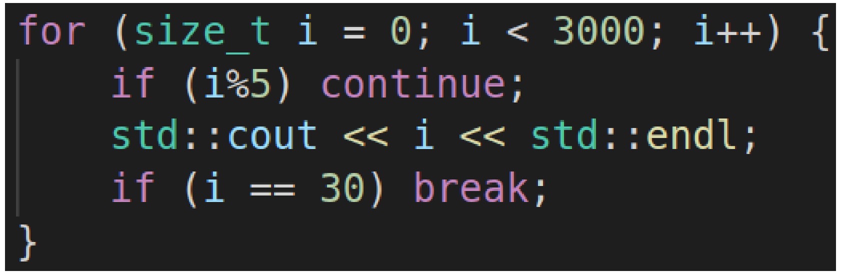


```cpp
switch ('b') {
case 'a':
    std::cout << ">>> a " << std::endl;
case 'b':
    std::cout << ">>> b " << std::endl;
case 'c':
    std::cout << ">>> c " << std::endl;
default:
    std::cout << "No Match Found!" << std::endl;
}
```

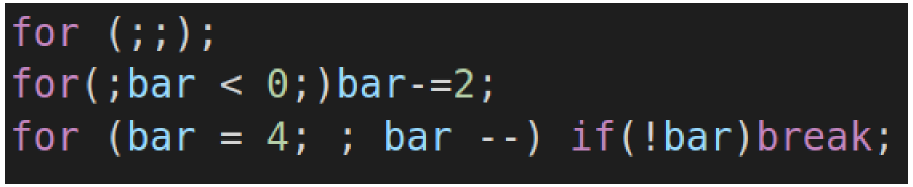
Loops



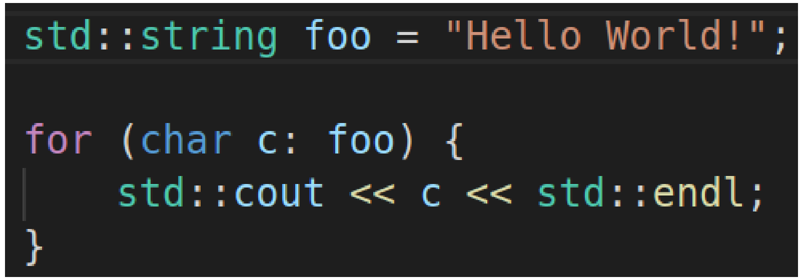
```cpp
int a = 10, b = 0, c = a;
while (b < 5) ++b;
do ++a; while (a < 0);
while (c < 0) c++;
```



```cpp
for (size_t i = 0; i < 3000; i++) {
    if (i%5) continue;
    std::cout << i << std::endl;
    if (i == 30) break;
}
```

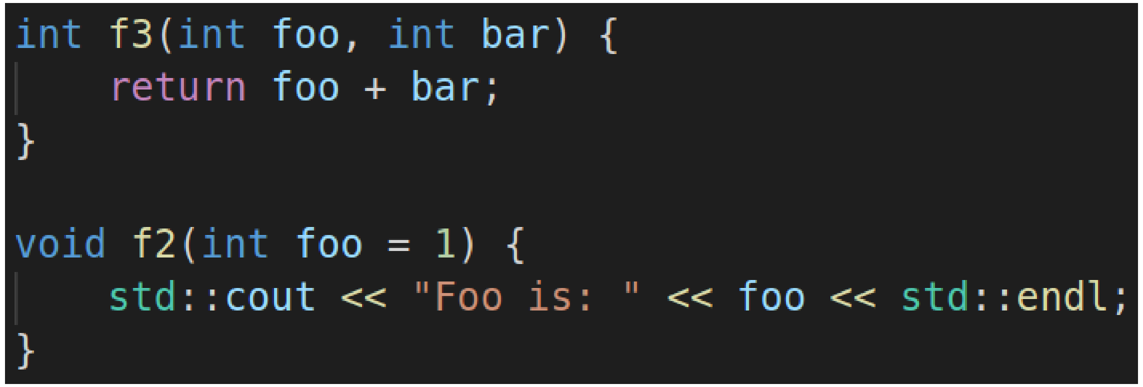


```cpp
for (;;);
for(;bar < 0;)bar-=2;
for (bar = 4; ; bar --) if(!bar)break;
```

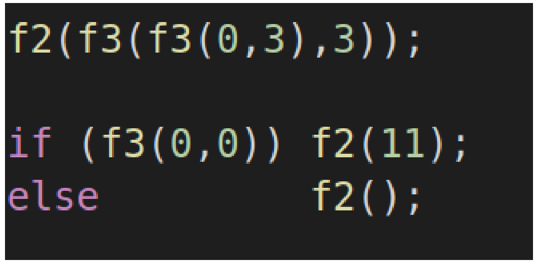


```cpp
std::string foo = "Hello World!";

for (char c: foo) {
    std::cout << c << std::endl;
}
```

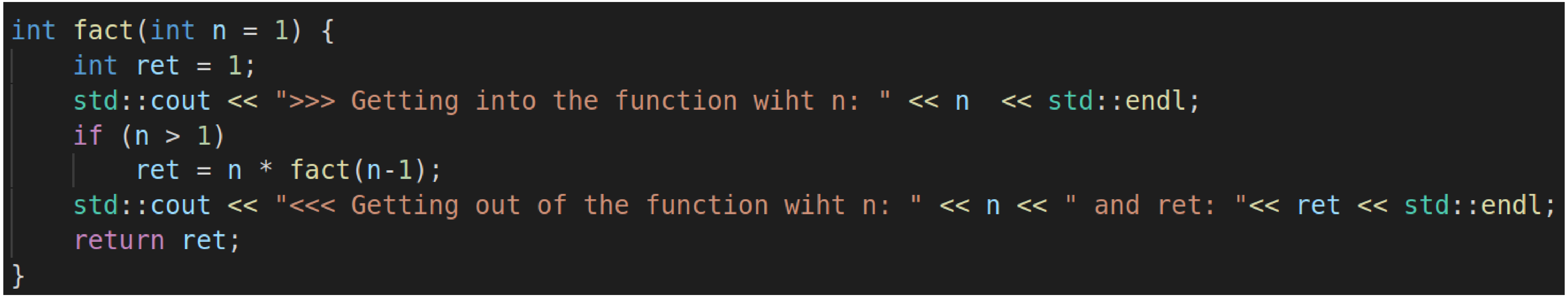
Functions



```cpp
int f3(int foo, int bar) {
    return foo + bar;
}

void f2(int foo = 1) {
    std::cout << "Foo is: " << foo << std::endl;
}
```

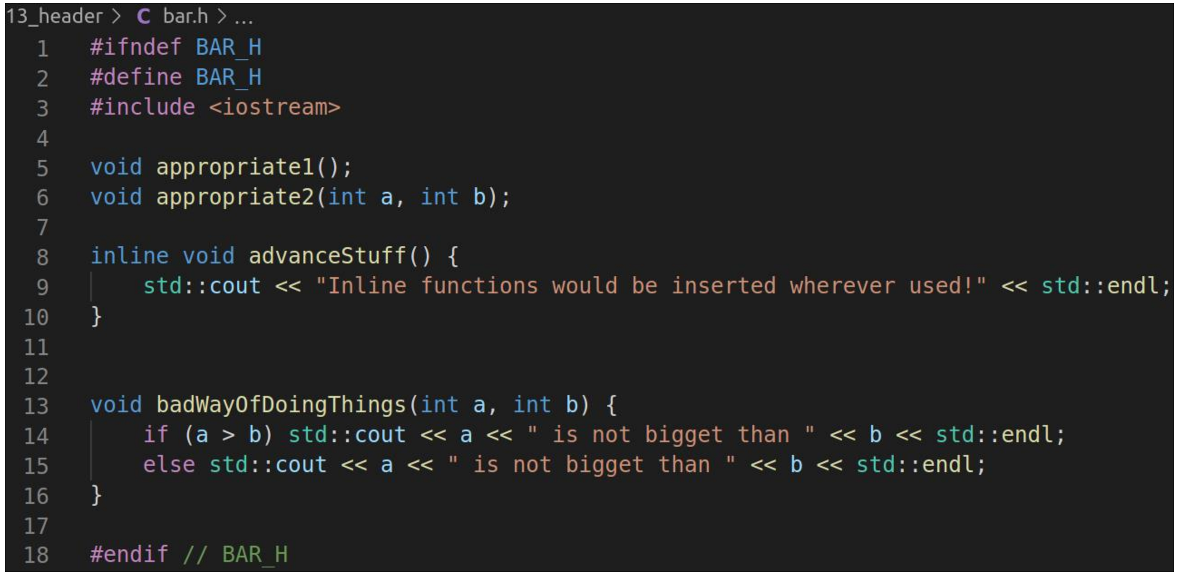


```cpp
f2(f3(f3(0,3),3));

if (f3(0,0)) f2(11);
else         f2();
```



```cpp
int fact(int n = 1) {
    int ret = 1;
    std::cout << ">>> Getting into the function wiht n: " << n  << std::endl;
    if (n > 1)
        ret = n * fact(n-1);
    std::cout << "<<< Getting out of the function wiht n: " << n << " and ret: "<< ret << std::endl;
    return ret;
}
```

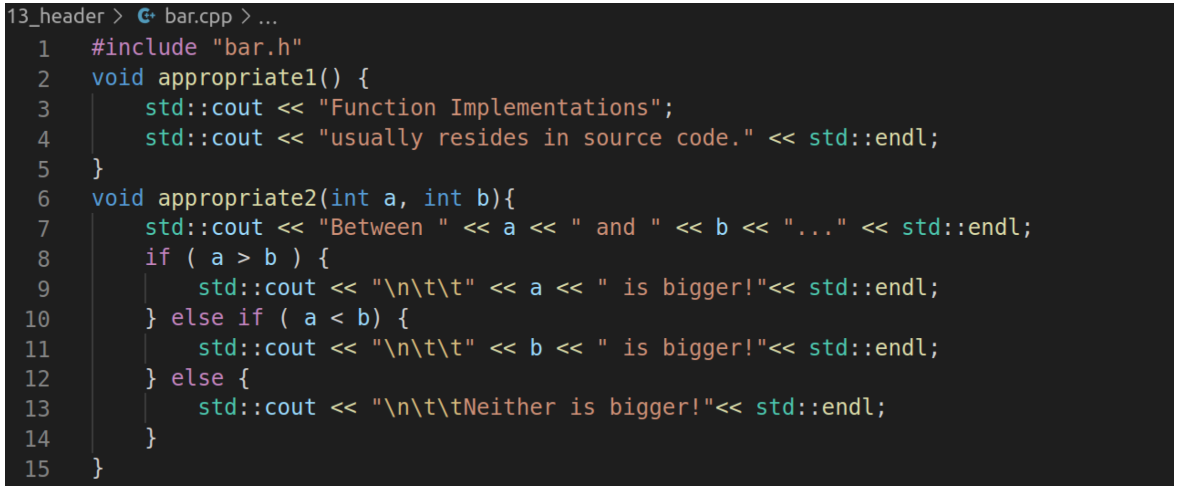
Header Files



13_header > bar.h > ...

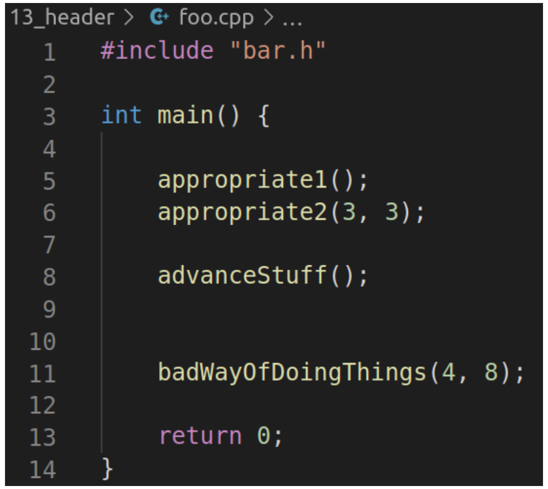
```cpp
#ifndef BAR_H
#define BAR_H
#include <iostream>

void appropriate1();
void appropriate2(int a, int b);

inline void advanceStuff() {
    std::cout << "Inline functions would be inserted wherever used!" << std::endl;
}


void badWayOfDoingThings(int a, int b) {
    if (a > b) std::cout << a << " is not bigget than " << b << std::endl;
    else std::cout << a << " is not bigget than " << b << std::endl;
}

#endif // BAR_H
```

> *In the C and C++ programming languages, an **#include guard**, sometimes called a **macro guard**, **header guard** or **file guard**, is a particular construct used to avoid the problem of double inclusion when dealing with the include directive.* https://en.wikipedia.org/wiki/Include_guard



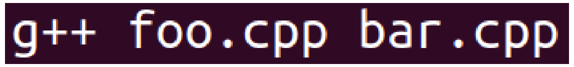
13_header > bar.cpp > ...

```cpp
#include "bar.h"
void appropriate1() {
    std::cout << "Function Implementations";
    std::cout << "usually resides in source code." << std::endl;
}
void appropriate2(int a, int b){
    std::cout << "Between " << a << " and " << b << "..." << std::endl;
    if ( a > b ) {
        std::cout << "\n\t\t" << a << " is bigger!"<< std::endl;
    } else if ( a < b) {
        std::cout << "\n\t\t" << b << " is bigger!"<< std::endl;
    } else {
        std::cout << "\n\t\tNeither is bigger!"<< std::endl;
    }
}
```



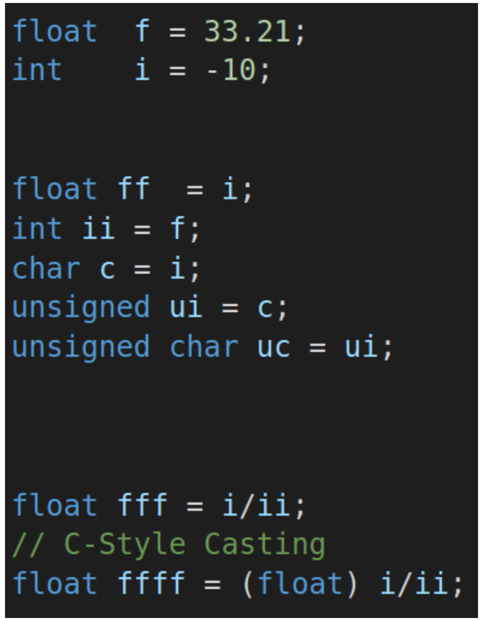
13_header > foo.cpp > ...

```cpp
#include "bar.h"

int main() {

    appropriate1();
    appropriate2(3, 3);

    advanceStuff();


    badWayOfDoingThings(4, 8);

    return 0;
}
```

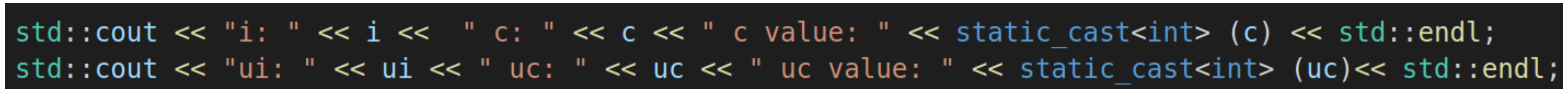


```
g++ foo.cpp bar.cpp
```

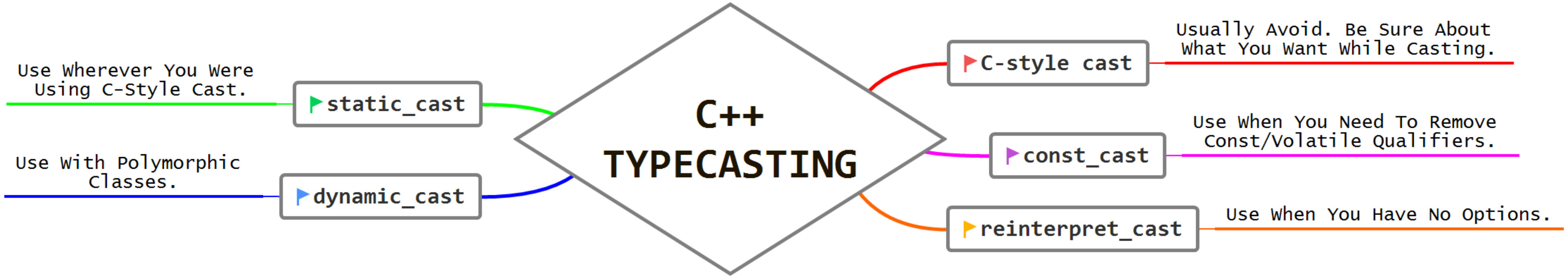
Type Alias



```
float  f = 33.21;
int    i = -10;


float ff  = i;
int ii = f;
char c = i;
unsigned ui = c;
unsigned char uc = ui;



float fff = i/ii;
// C-Style Casting
float ffff = (float) i/ii;
```



```
std::cout << "i: " << i <<  " c: " << c << " c value: " << static_cast<int> (c) << std::endl;
std::cout << "ui: " << ui << " uc: " << uc << " uc value: " << static_cast<int> (uc)<< std::endl;
```



C++
TYPECASTING
static_cast
Use Wherever You Were Using C-Style Cast.
dynamic_cast
Use With Polymorphic Classes.
C-style cast
Usually Avoid. Be Sure About What You Want While Casting.
const_cast
Use When You Need To Remove Const/Volatile Qualifiers.
reinterpret_cast
Use When You Have No Options.

http://www.vishalchovatiya.com/cpp-type-casting-with-example-for-c-developers/

Pointers

- Address to a specific memory cell

- Declared using *:

```
SomeDatatype *PointerName;
```

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

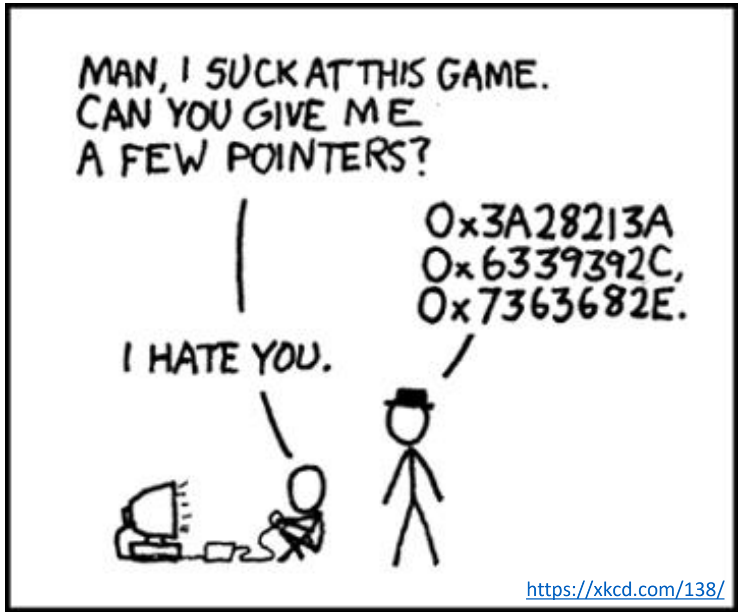
```
SomeDatatype *PointerName = &VariableName;
```

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

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

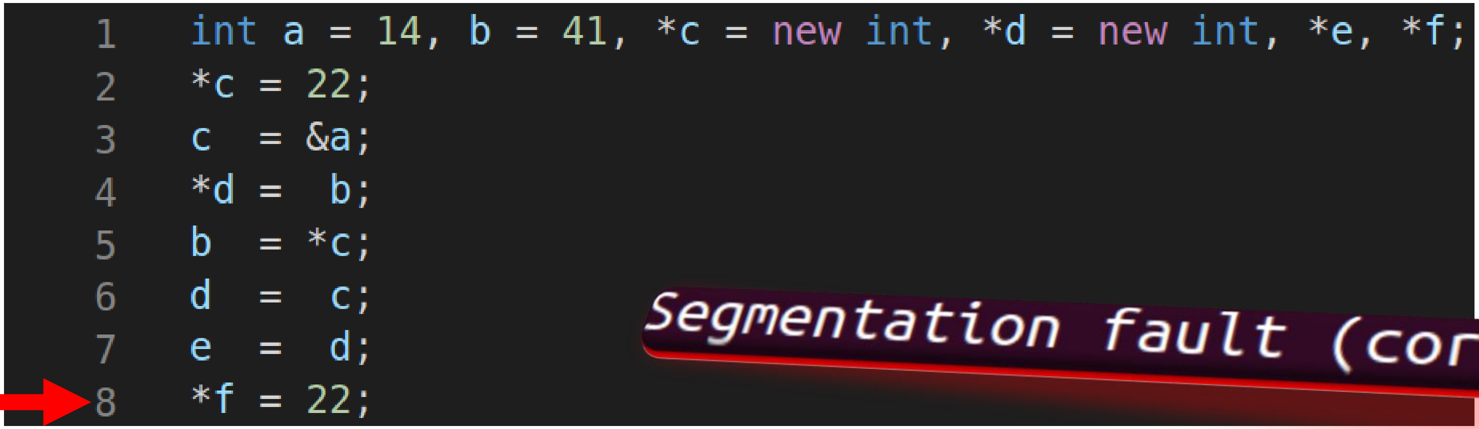
0x7ffd8a26b400	0x48
0x7ffd8a26b401	0x69
0x7ffd8a26b402	0x00

- `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"*.



https://xkcd.com/138/

Pointers In Action



```
int a = 14, b = 41, *c = new int, *d = new int, *e, *f;
*c = 22;
c  = &a;
*d =  b;
b  = *c;
d  =  c;
e  =  d;
*f = 22;
```

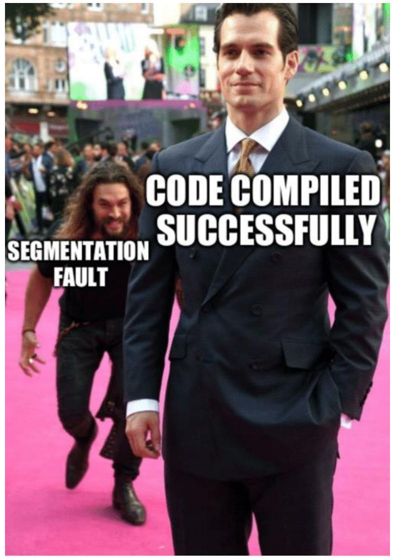
Segmentation fault (core dumped)

0x7ffd8a26b400	22
0x7ffd8a26b401	0x69
0x7ffd8a26b402	41
.	
.	

a	14	0x7ffd8a26b800
b	14	0x7ffd8a26b801
*c	0x7ffd8a26b800	0x7ffd8a26b802
*d	0x7ffd8a26b800	0x7ffd8a26b803
*e	0x7ffd8a26b800	0x7ffd8a26b804
*f	nullptr	0x7ffd8a26b805

- 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

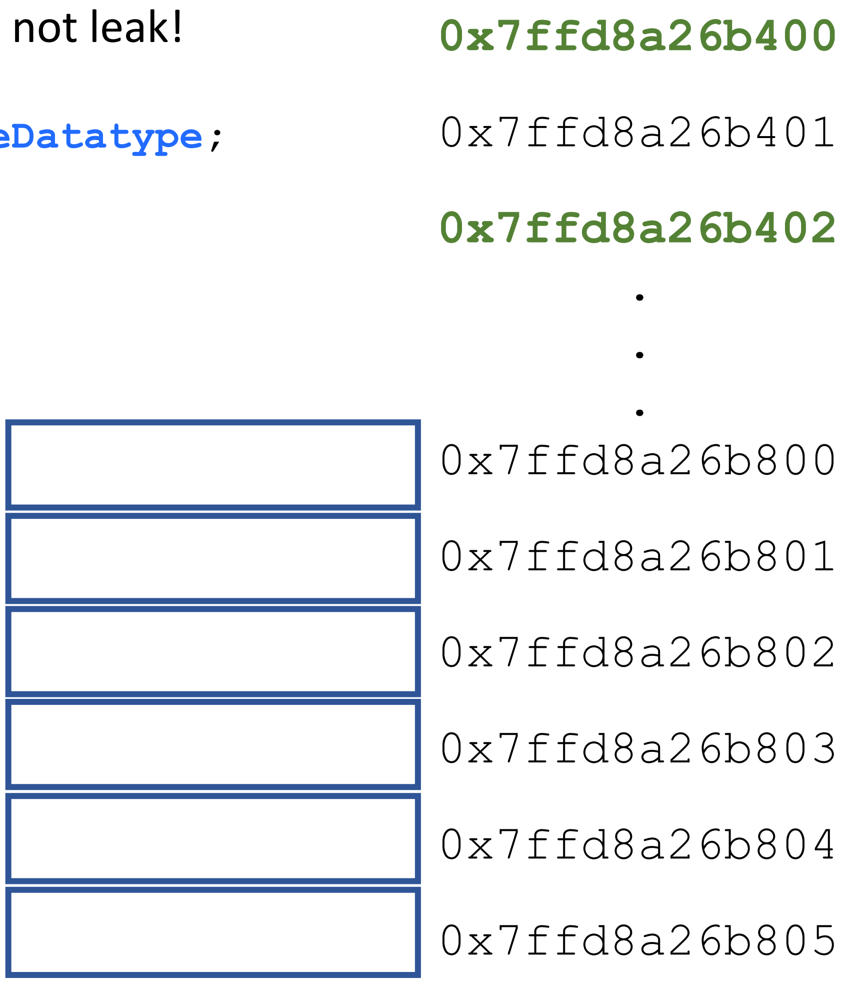


It happens every time...

Memory Leakage

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

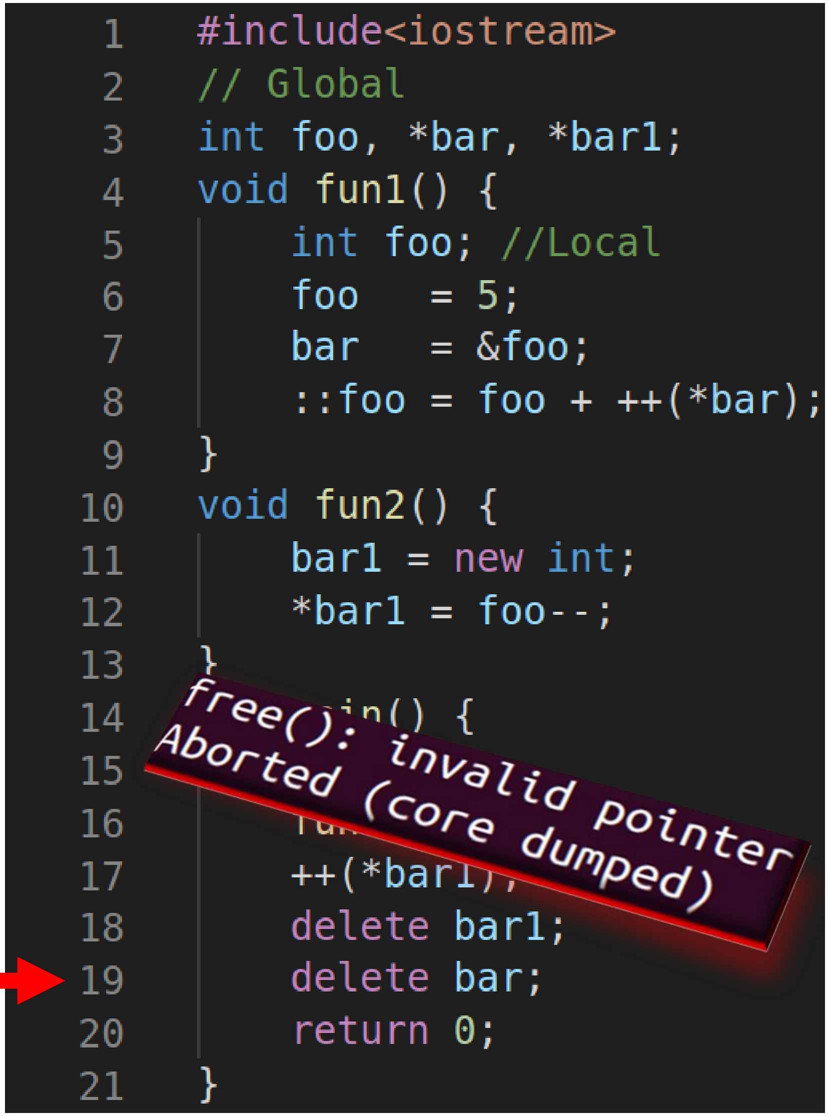
```
SomeDatatype *PointerName = new SomeDatatype;
delete PointerName;
```



- 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

Scope



```
#include<iostream>
// Global
int foo, *bar, *bar1;
void fun1() {
    int foo; //Local
    foo   = 5;
    bar   = &foo;
    ::foo = foo + ++(*bar);
}
void fun2() {
    bar1 = new int;
    *bar1 = foo--;
}
    in() {

    fun
    ++(*bar1);
    delete bar1;
    delete bar;
    return 0;
}
```

```
free(): invalid pointer
Aborted (core dumped)
```

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

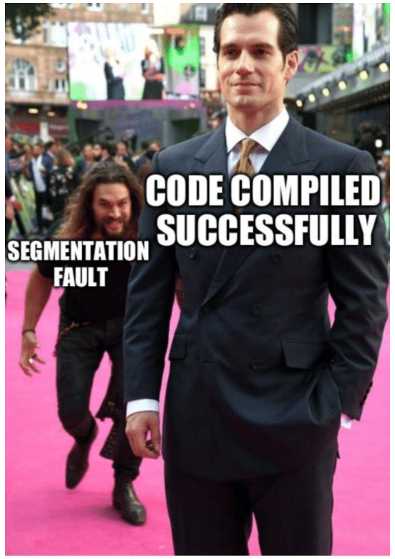
0x7ffd8a26b400	
0x7ffd8a26b401	0x69
0x7ffd8a26b402	41

.

.

.

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



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!
```

0x7ffd8a26b400	
0x7ffd8a26b401	0x69
0x7ffd8a26b402	41

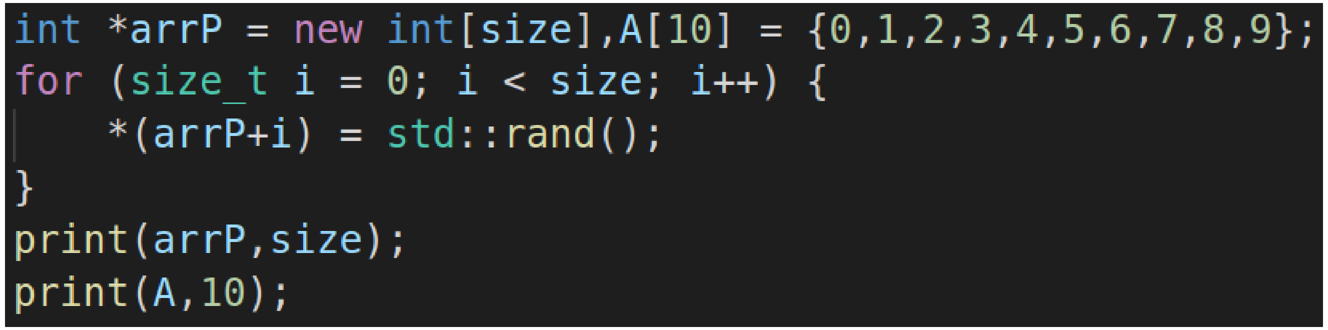
SIZE

.
.
.

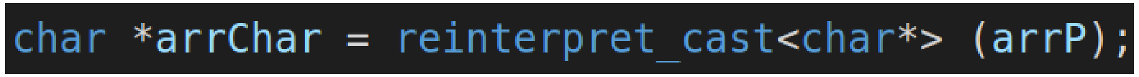
VariableName	SomeValue	0x7ffd8a26b800
*PtrName1	0x7ffd8a26b800	0x7ffd8a26b801
*PtrName2	0x7ffd8a26b400	0x7ffd8a26b802
*PtrName3	0x7ffd8a26b402	0x7ffd8a26b803

- 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.

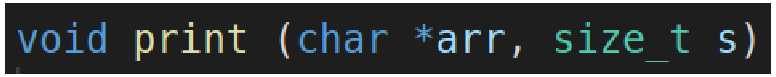
Pointers Are Arrays



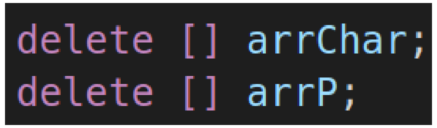
```cpp
int *arrP = new int[size],A[10] = {0,1,2,3,4,5,6,7,8,9};
for (size_t i = 0; i < size; i++) {
    *(arrP+i) = std::rand();
}
print(arrP,size);
print(A,10);
```



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



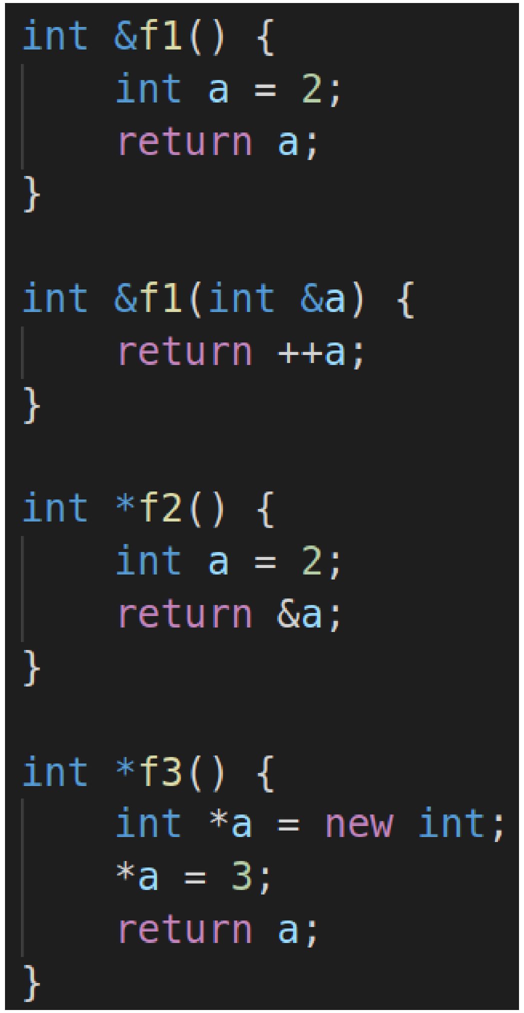
```cpp
void print (char *arr, size_t s)
```



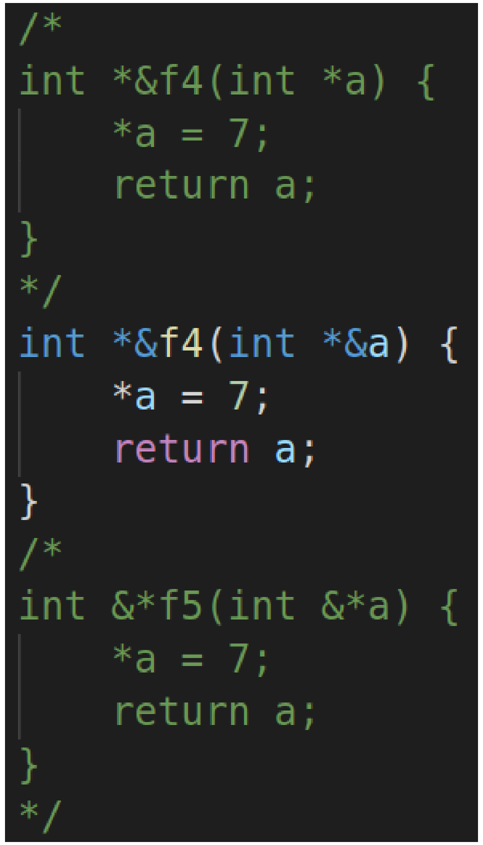
```cpp
delete [] arrChar;
delete [] arrP;
```

Reference vs Pointers

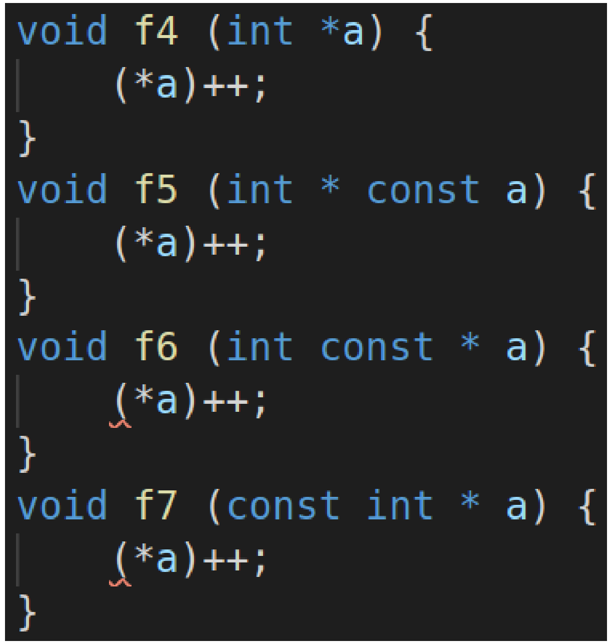
"reference ***is less powerful but safer*** *than the pointer"* https://en.wikipedia.org/wiki/Reference_(C%2B%2B)



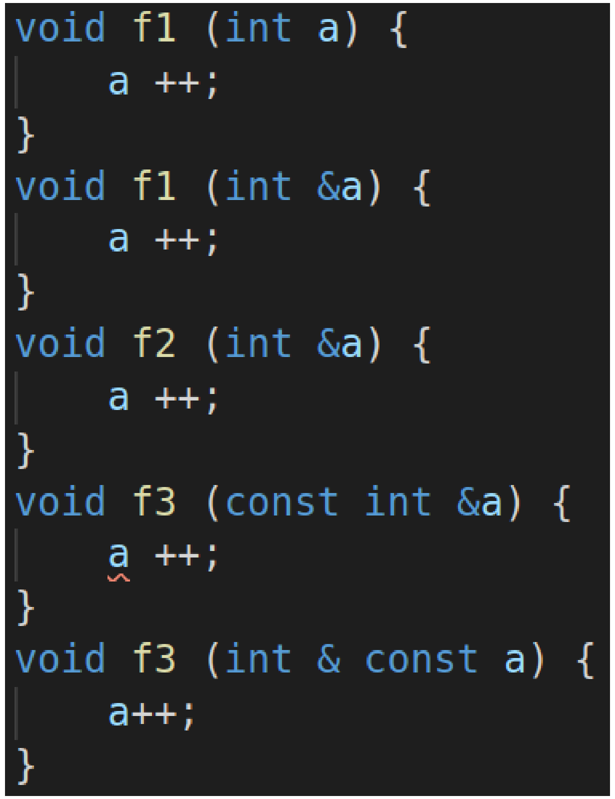
```
int &f1() {
    int a = 2;
    return a;
}

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)++;
}
```

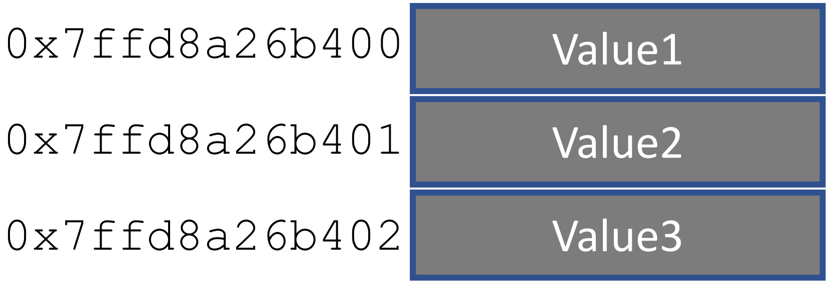


```
void f1 (int a) {
    a ++;
}
void f1 (int &a) {
    a ++;
}
void f2 (int &a) {
    a ++;
}
void f3 (const int &a) {
    a ++;
}
void f3 (int & const a) {
    a++;
}
```

- 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

struct

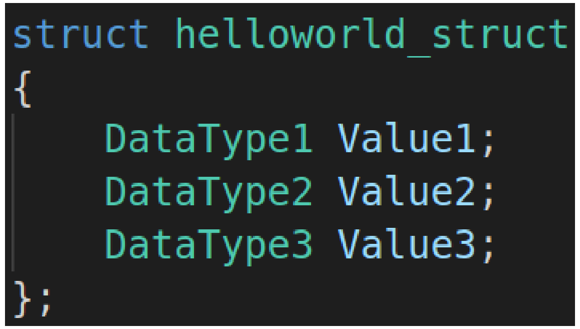
- **struct** is a type consisting of a sequence of *members* whose storage is allocated in an ***ordered sequence***.



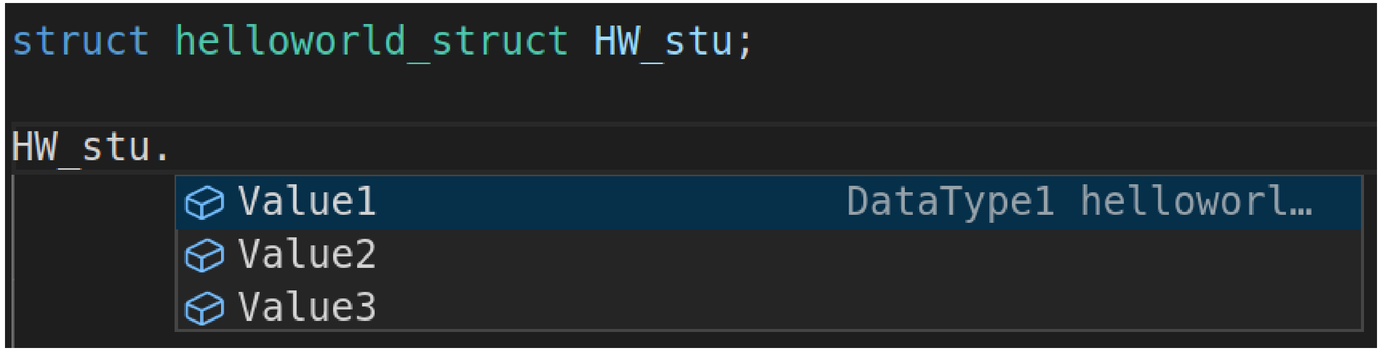
0x7ffd8a26b400 Value1

0x7ffd8a26b401 Value2

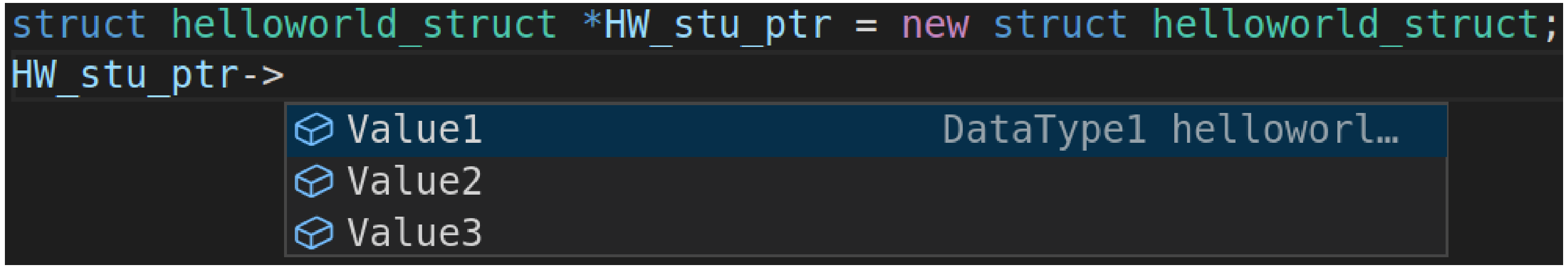
0x7ffd8a26b402 Value3



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



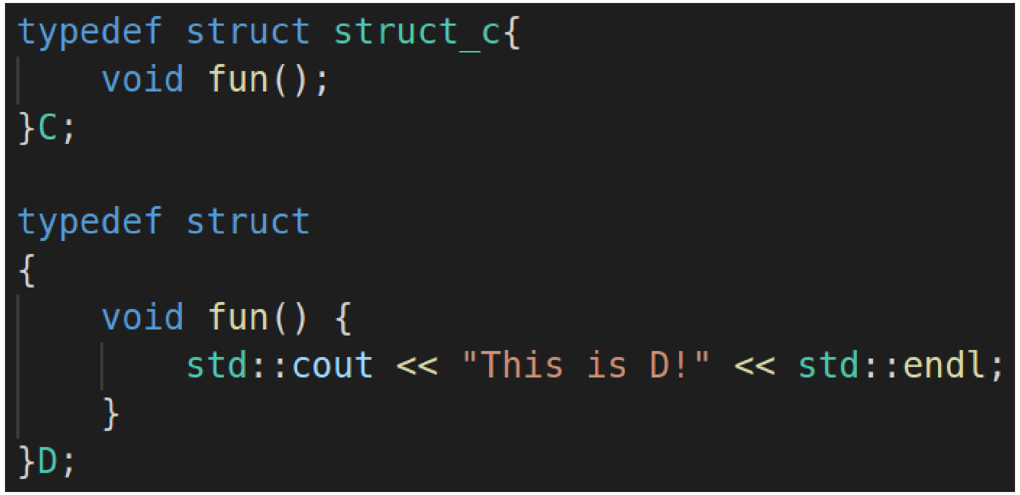
```
struct helloworld_struct HW_stu;

HW_stu.
        Value1                  DataType1 helloworl…
        Value2
        Value3
```



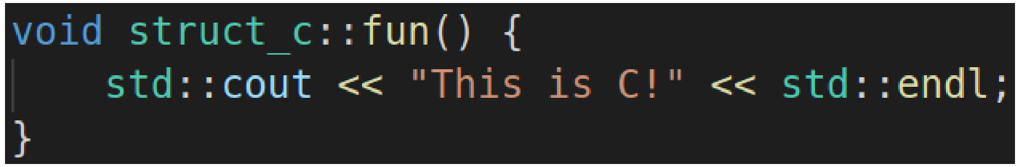
```
struct helloworld_struct *HW_stu_ptr = new struct helloworld_struct;
HW_stu_ptr->
            Value1                  DataType1 helloworl…
            Value2
            Value3
```

- **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.

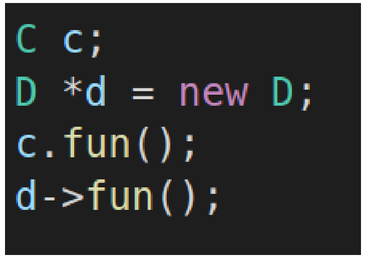
struct



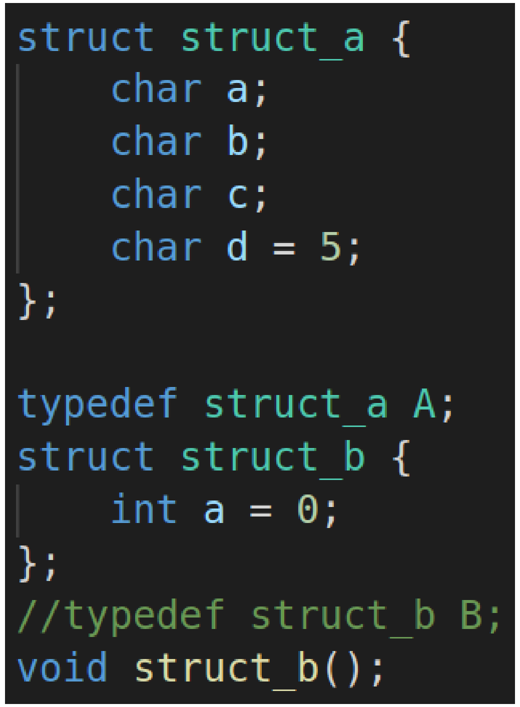
```
typedef struct struct_c{
    void fun();
}C;

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



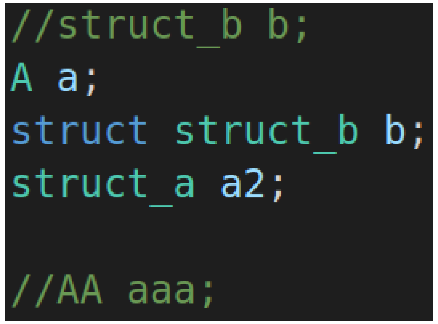
```
void struct_c::fun() {
    std::cout << "This is C!" << std::endl;
}
```



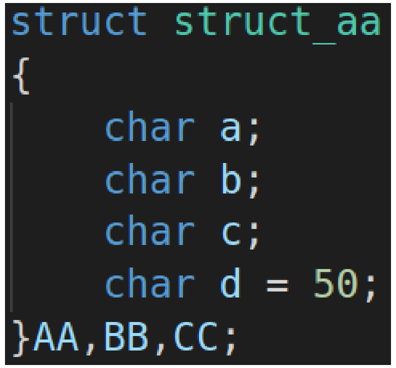
```
C c;
D *d = new D;
c.fun();
d->fun();
```



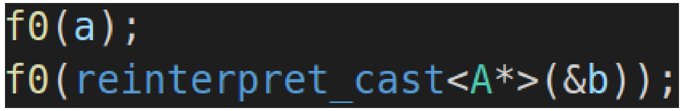
```
struct struct_a {
    char a;
    char b;
    char c;
    char d = 5;
};

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



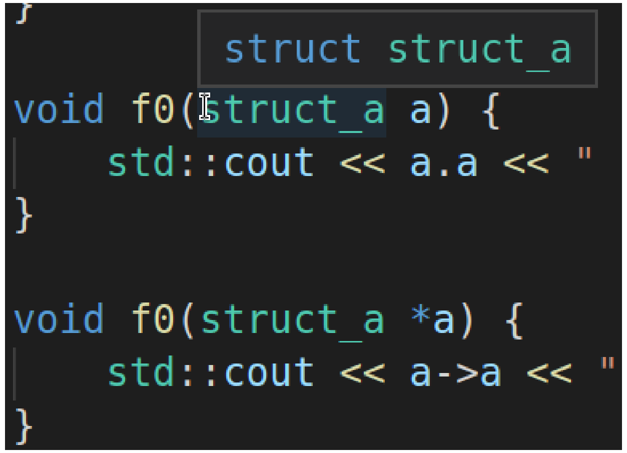
```
//struct_b b;
A a;
struct struct_b b;
struct_a a2;

//AA aaa;
```



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



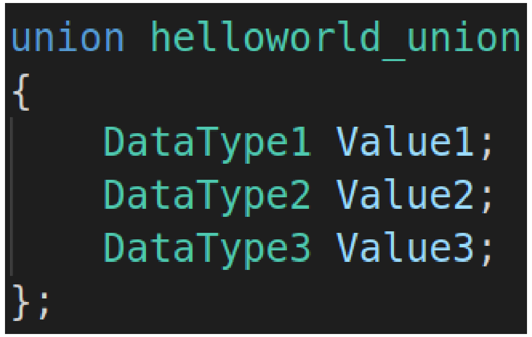
```
f0(a);
f0(reinterpret_cast<A*>(&b));
```



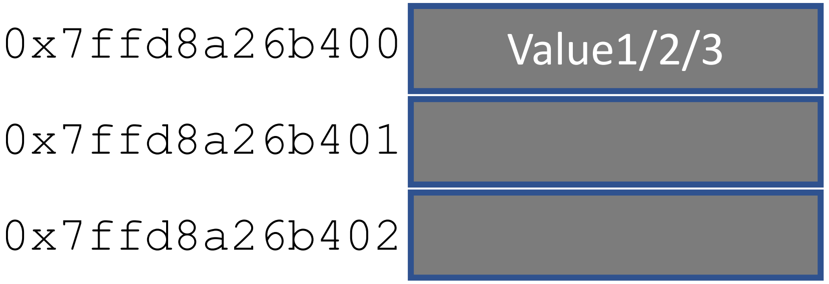
```
}
            struct struct_a

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

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

union

- **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.

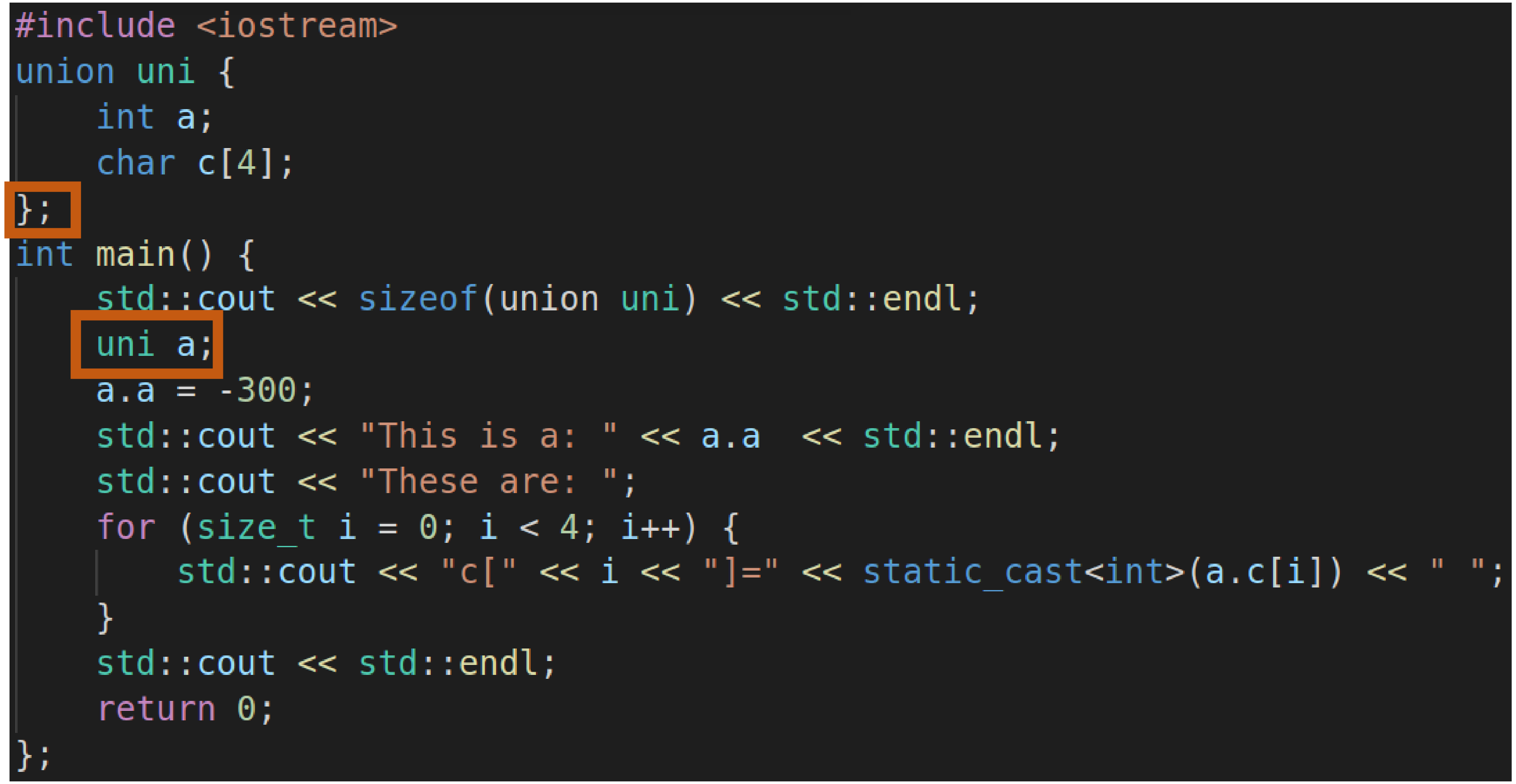
- Size of **union** is equal to the size of its biggest member, while size of **struct** is at least sum of all its member.

- 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.

union

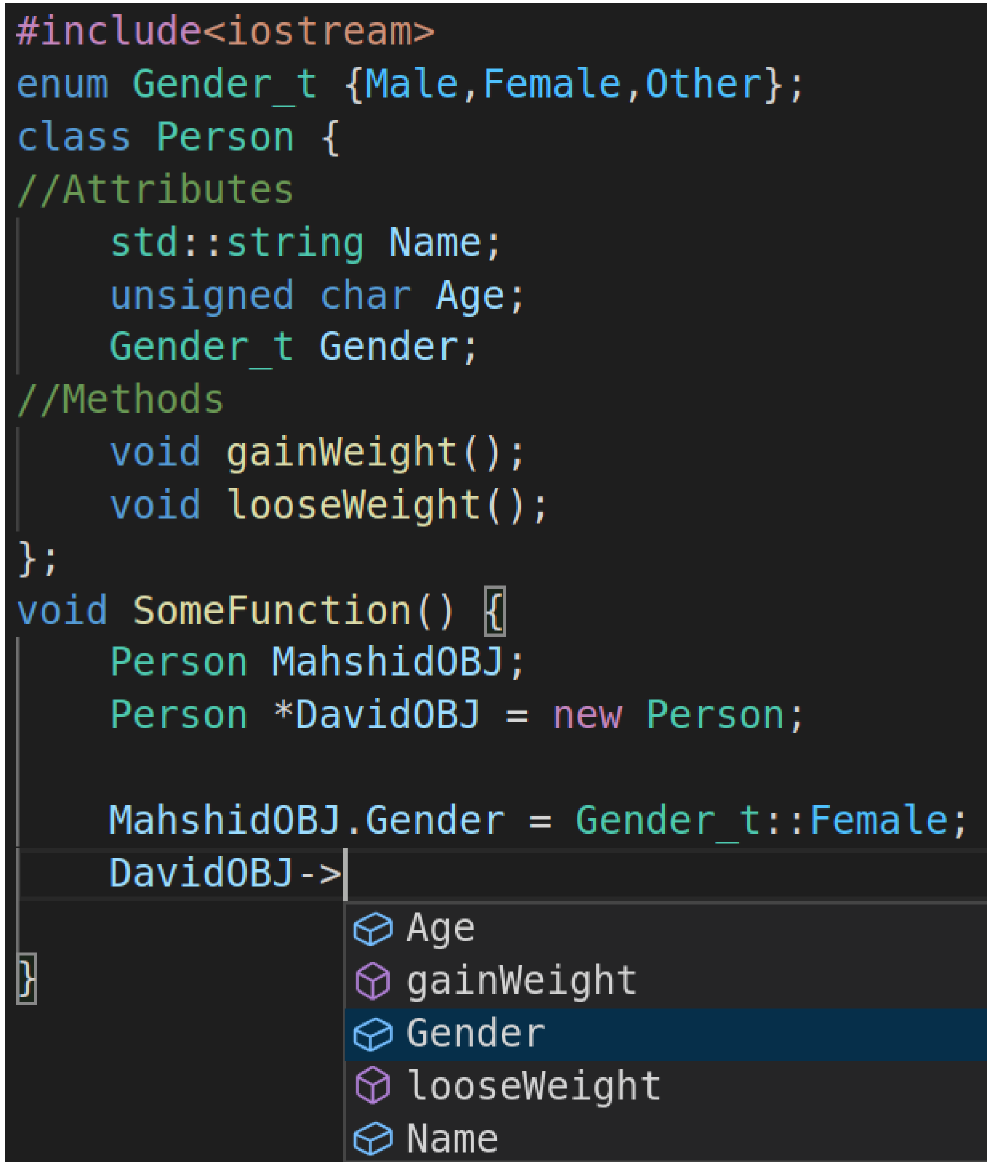
- **`union`**s are identical to **`struct`**s syntax wise; the only difference is the *storage* for union is *overlapping*!



```
#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;
};
```

BREAK!

Object & Classes



```
#include<iostream>
enum Gender_t {Male,Female,Other};
class Person {
//Attributes
    std::string Name;
    unsigned char Age;
    Gender_t Gender;
//Methods
    void gainWeight();
    void looseWeight();
};
void SomeFunction() {
    Person MahshidOBJ;
    Person *DavidOBJ = new Person;

    MahshidOBJ.Gender = Gender_t::Female;
    DavidOBJ->
}
```

	Class Person		Object1	Object2
data	Attributes	std::string Name;	Mahshid	David
		unsigned char Age;	47	30
		enum Gender;	Female	Male
		unsigned char Weight;	55	80
logic	Methods	void gainWeight();	Each object has "its own" methods	
		void looseWeight();		

- A **class** is a compound datatype and can be used like other datatypes. Similar access operator as **struct** and **union** are used to access members of an object, depending on the memory the object is instantiated in: '**.**' for static and automatic memory, and '**->**' for dynamic memory.
- Any instantiation of a **class** creates an object of that **class** which contains a *"copy"* of the declared members for itself.

Instantiation & Destruction – In Theory

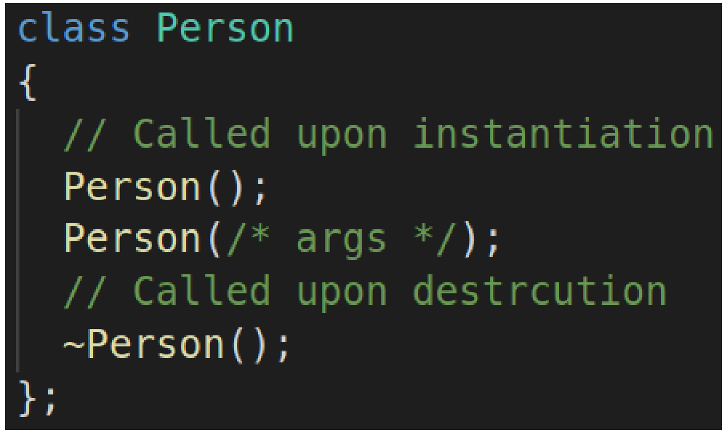
- Every `class` has two ***special*** methods:

  - *<u>Constructor</u>*: a method with the same name of the class. *Constructors* can be overloaded; providing the different possibilities for object instantiation.

  - *<u>Destructor</u>:* a method with the same name as the class with a prepending `` `~` ``. Destructors cannot be overloaded, and they do not accept any input arguments.

- Upon object initialization of an object an "*appropriate*" constructor is invoked.

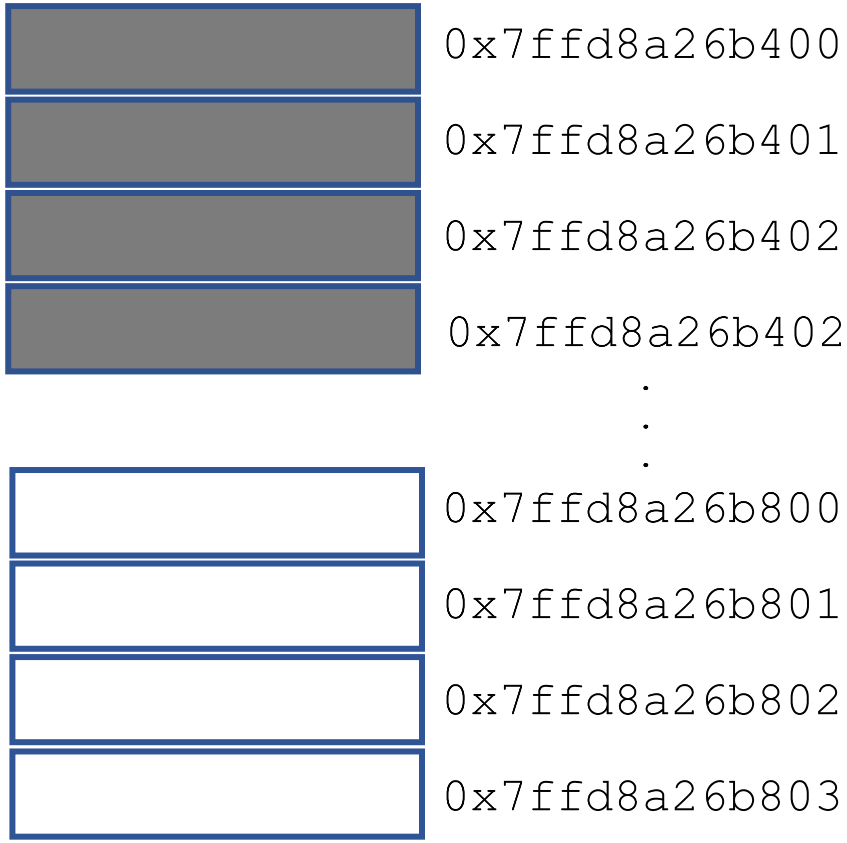
- Upon object destruction, the destructor of the object is called.

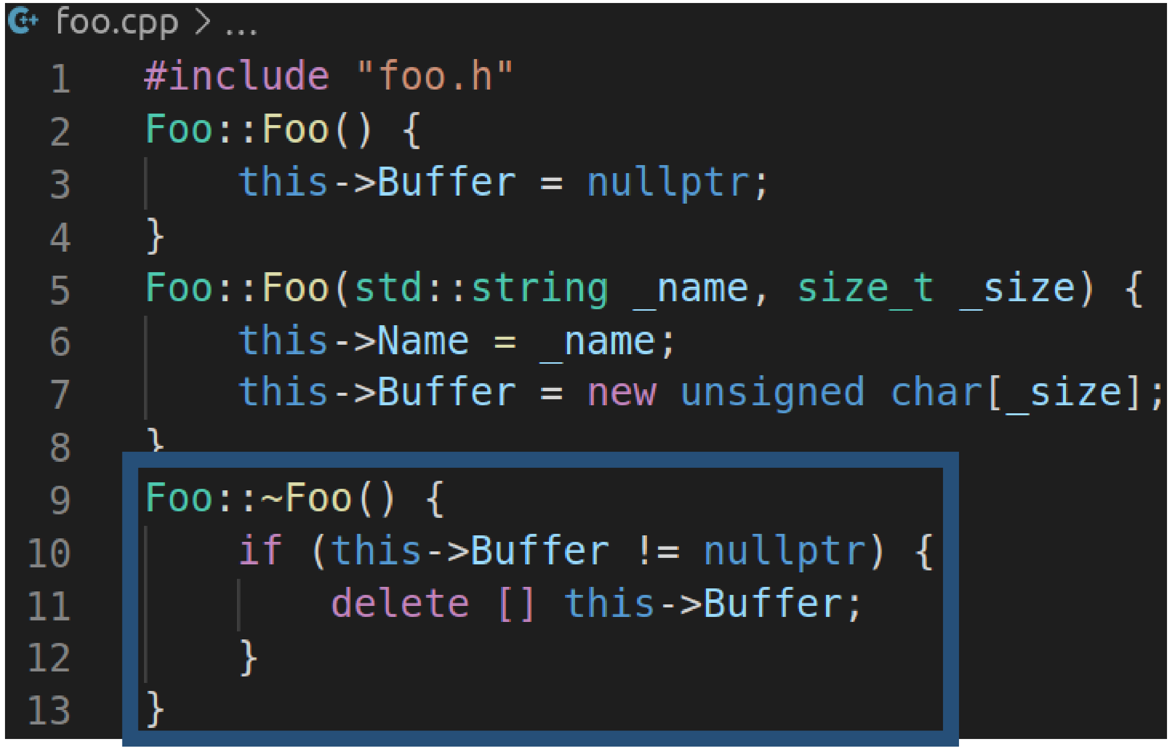


```
class Person
{
  // Called upon instantiation
  Person();
  Person(/* args */);
  // Called upon destrcution
  ~Person();
};
```

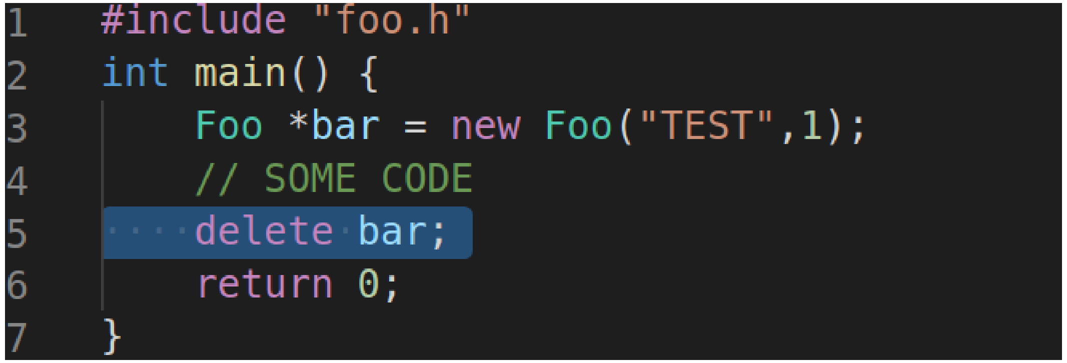
- If no constructor is provided for the class, the compiler includes a *default constructor.*
- *Default constructors* are those which could be called without any arguments i.e. constructor does not receive any arguments or there is a default value for each input to the constructor.

Object initialization and constructor destructor





```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```



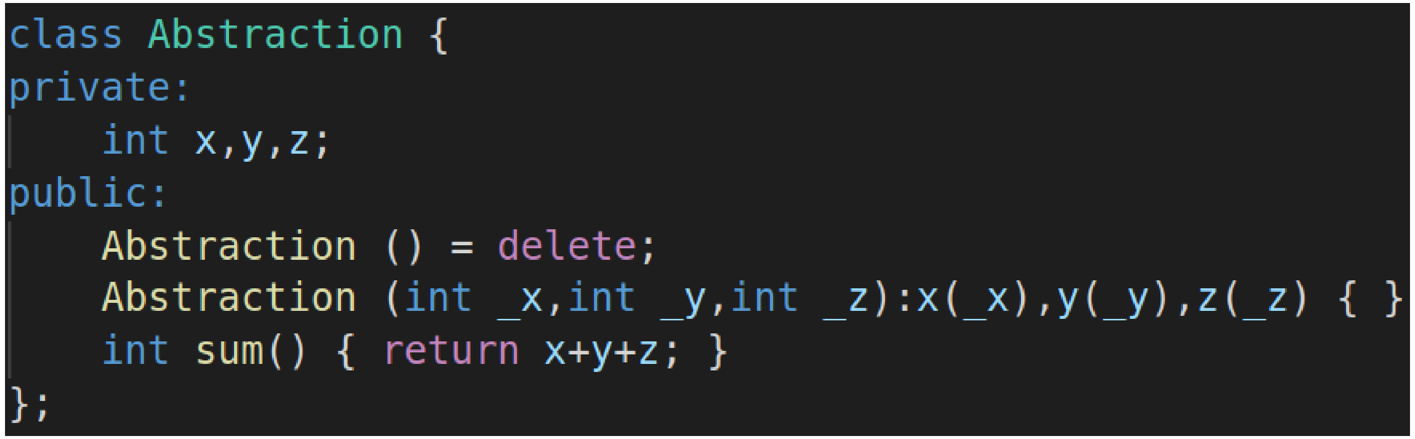
```cpp
#include "foo.h"
int main() {
    Foo *bar = new Foo("TEST",1);
    // SOME CODE
    delete bar;
    return 0;
}
```

- Destructor for objects declared in *dynamic memory* ***would be*** invoked if the pointer is `delete`d.

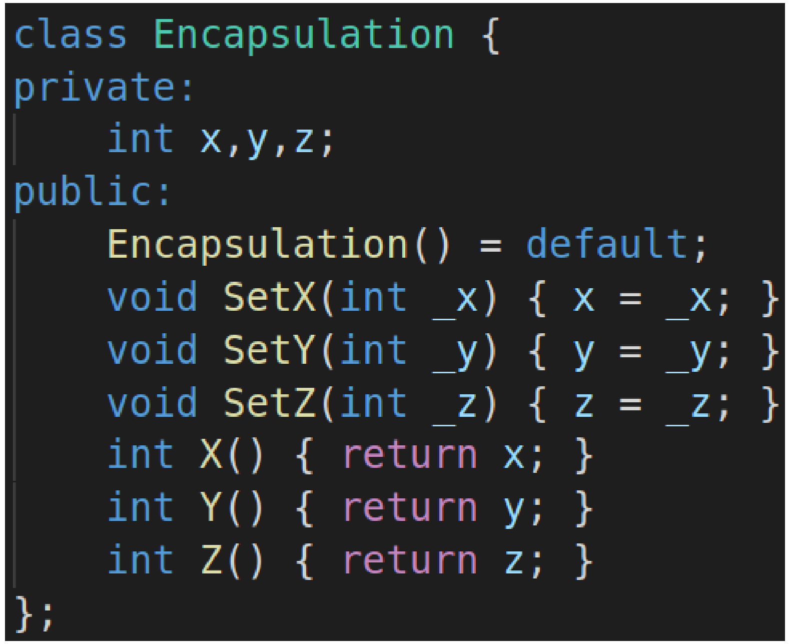
Object Oriented Programming

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Message Passing

Abstraction vs. Encapsulation



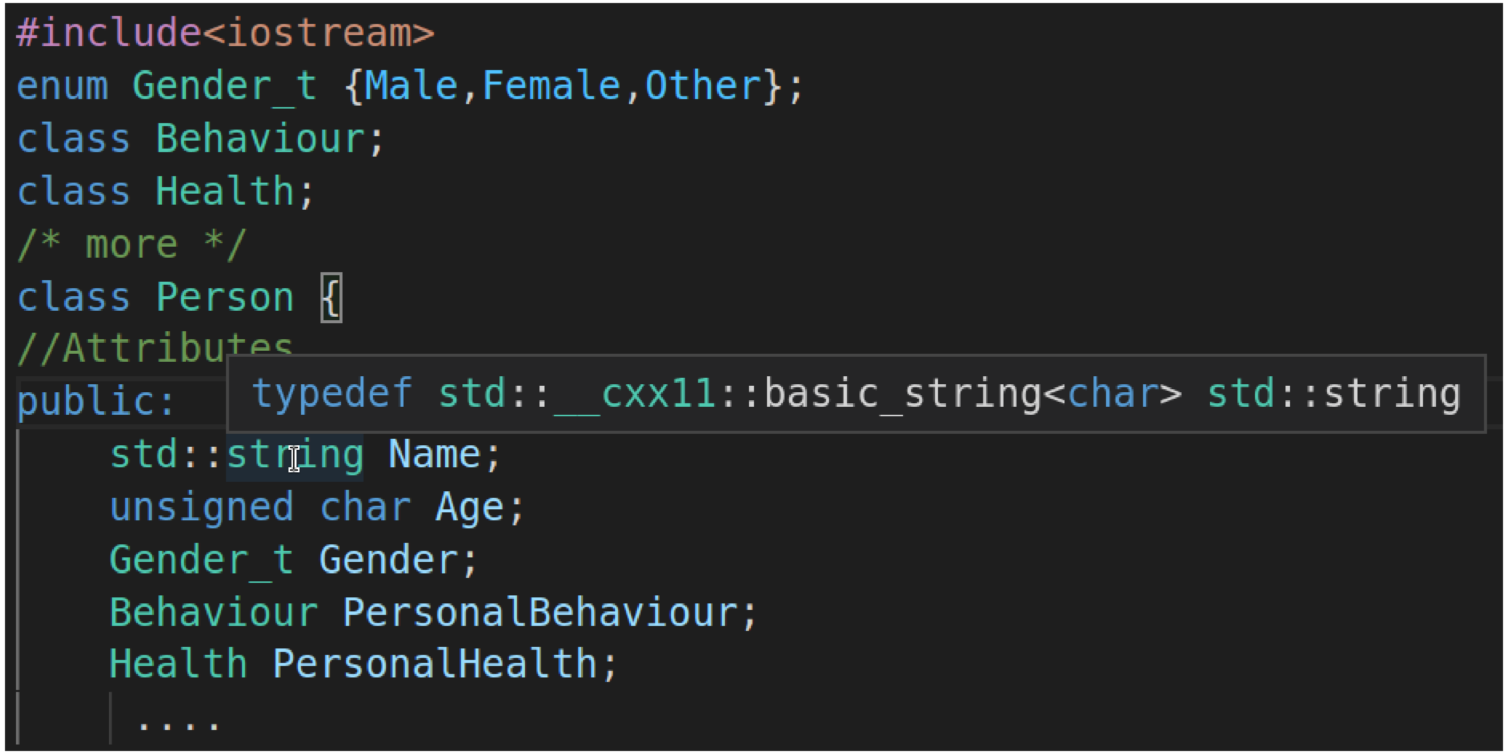
```cpp
class Abstraction {
private:
    int x,y,z;
public:
    Abstraction () = delete;
    Abstraction (int _x,int _y,int _z):x(_x),y(_y),z(_z) { }
    int sum() { return x+y+z; }
};
```



```cpp
class Encapsulation {
private:
    int x,y,z;
public:
    Encapsulation() = default;
    void SetX(int _x) { x = _x; }
    void SetY(int _y) { y = _y; }
    void SetZ(int _z) { z = _z; }
    int X() { return x; }
    int Y() { return y; }
    int Z() { return z; }
};
```

Object Composition

> *"In computer science, object composition is a way to combine objects or data types into more complex ones."* https://en.wikipedia.org/wiki/Object_composition



```
#include<iostream>
enum Gender_t {Male,Female,Other};
class Behaviour;
class Health;
/* more */
class Person {
//Attributes
public:
    std::string Name;
    unsigned char Age;
    Gender_t Gender;
    Behaviour PersonalBehaviour;
    Health PersonalHealth;
     ....
```

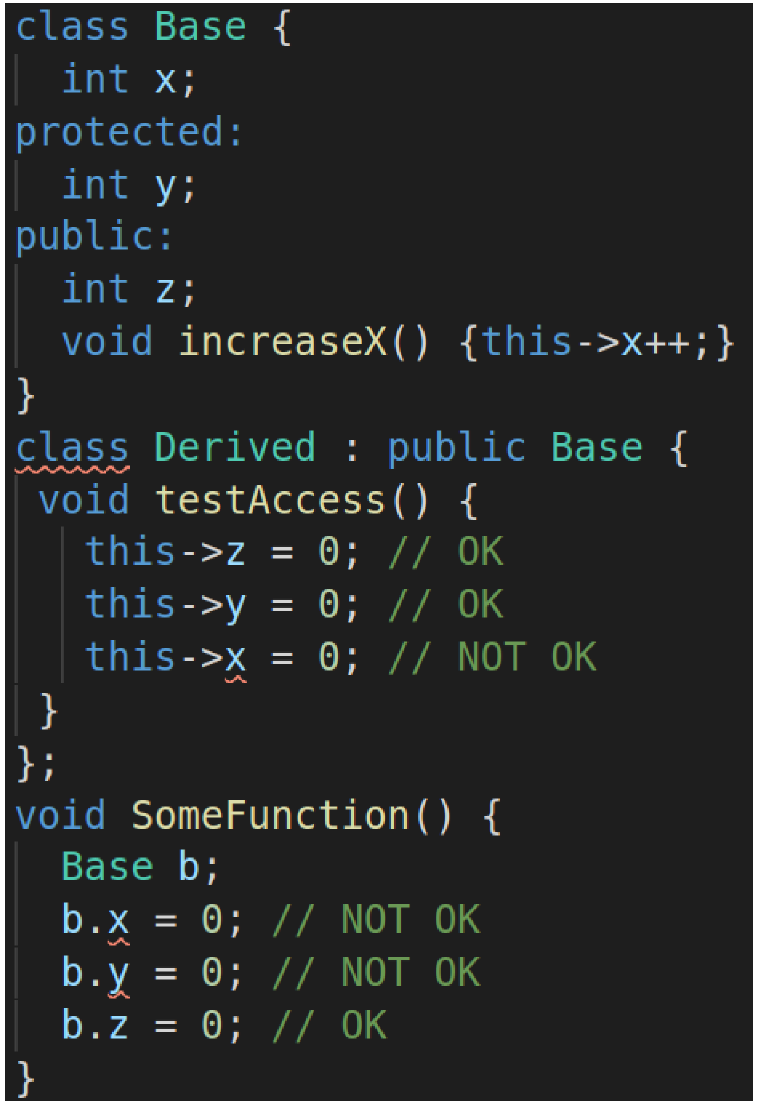
```
typedef std::__cxx11::basic_string<char> std::string
```

Inheritance

- Derived class inherits the `private` members of the *base* class yet **cannot** access them i.e. `private` members are not accessible from within derived classes.

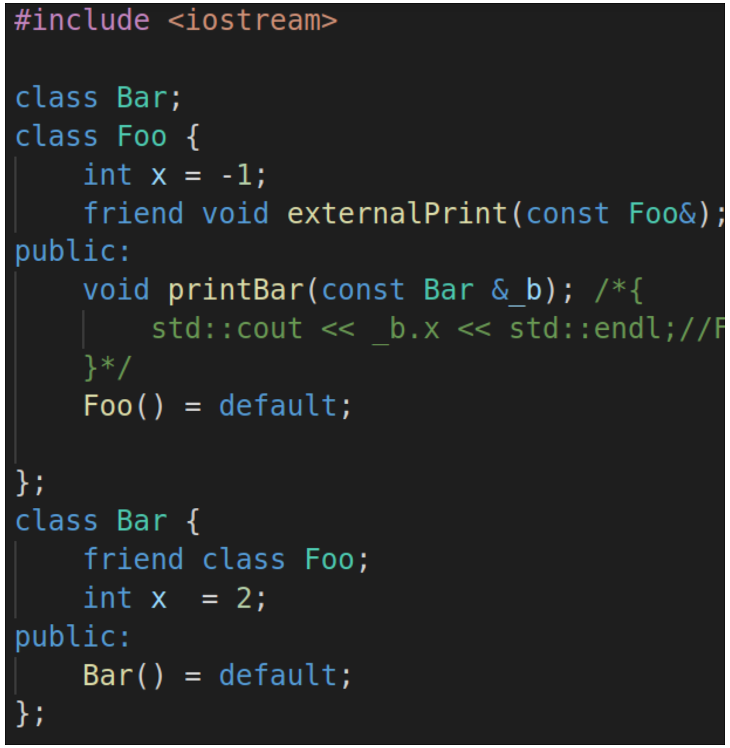
- Members declared `protected` are not accessible to public.

Keyword	Description For Specifying Inheritance Access Level
`public`	Inherited members keep the same access specifier from the *base* class.
`private`	All `public` and `protected` inherited members become `private`.
`protected`	All `public` and `protected` inherited members become `protected`.
Keyword	Description For Specifying Member Access Level
`public`	Accessible to everyone who has access to the class
`private`	Accessible only to the members of a class and its friends.
`protected`	Accessible only to the members and friends of a class, and members (and friends until C++17) of derived classes.

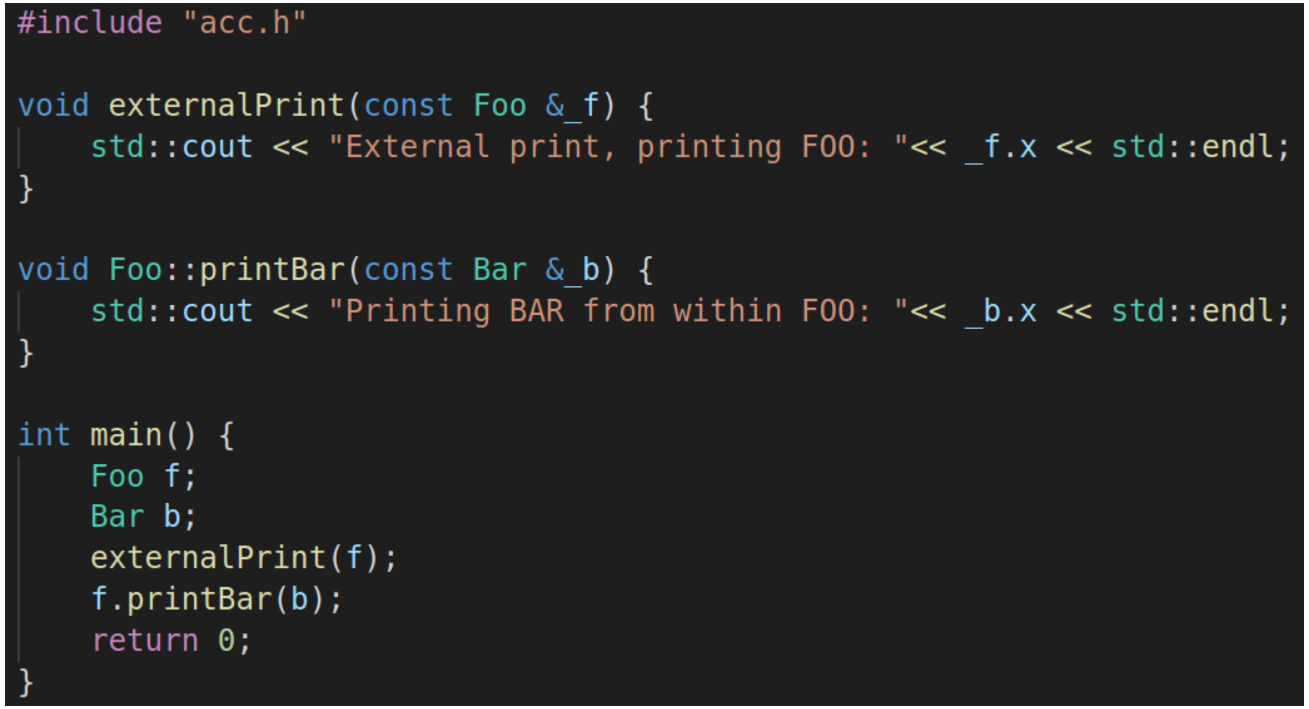


```
class Base {
  int x;
protected:
  int y;
public:
  int z;
  void increaseX() {this->x++;}
}
class Derived : public Base {
 void testAccess() {
   this->z = 0; // OK
   this->y = 0; // OK
   this->x = 0; // NOT OK
 }
};
void SomeFunction() {
  Base b;
  b.x = 0; // NOT OK
  b.y = 0; // NOT OK
  b.z = 0; // OK
}
```

Friendship

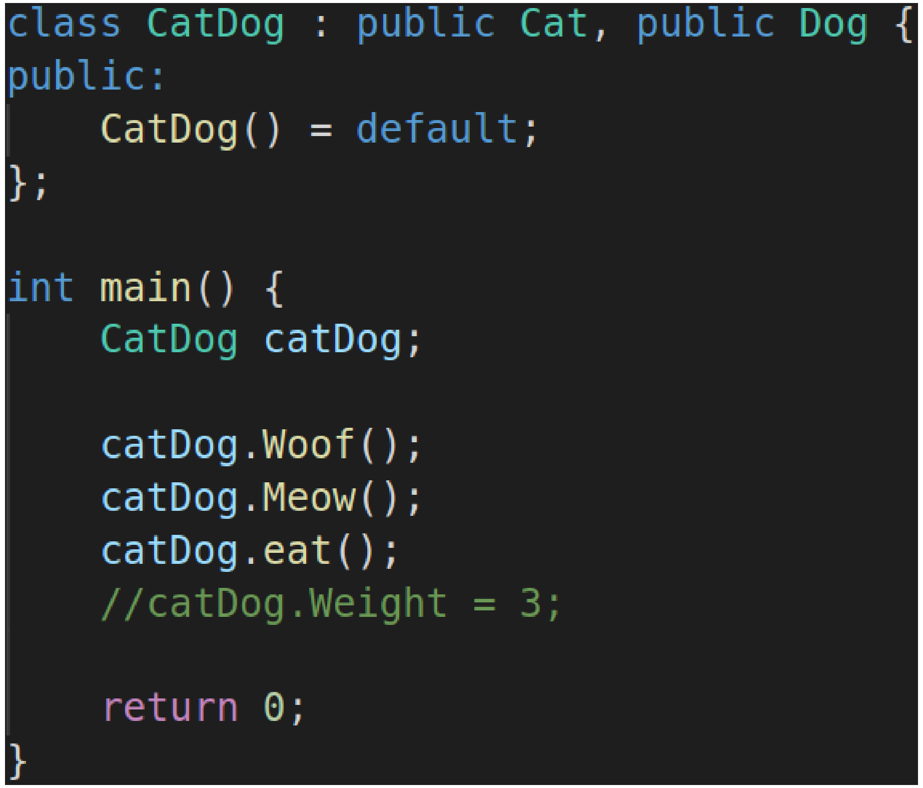


```cpp
#include <iostream>

class Bar;
class Foo {
    int x = -1;
    friend void externalPrint(const Foo&);
public:
    void printBar(const Bar &_b); /*{
        std::cout << _b.x << std::endl;//F
    }*/
    Foo() = default;

};
class Bar {
    friend class Foo;
    int x  = 2;
public:
    Bar() = default;
};
```

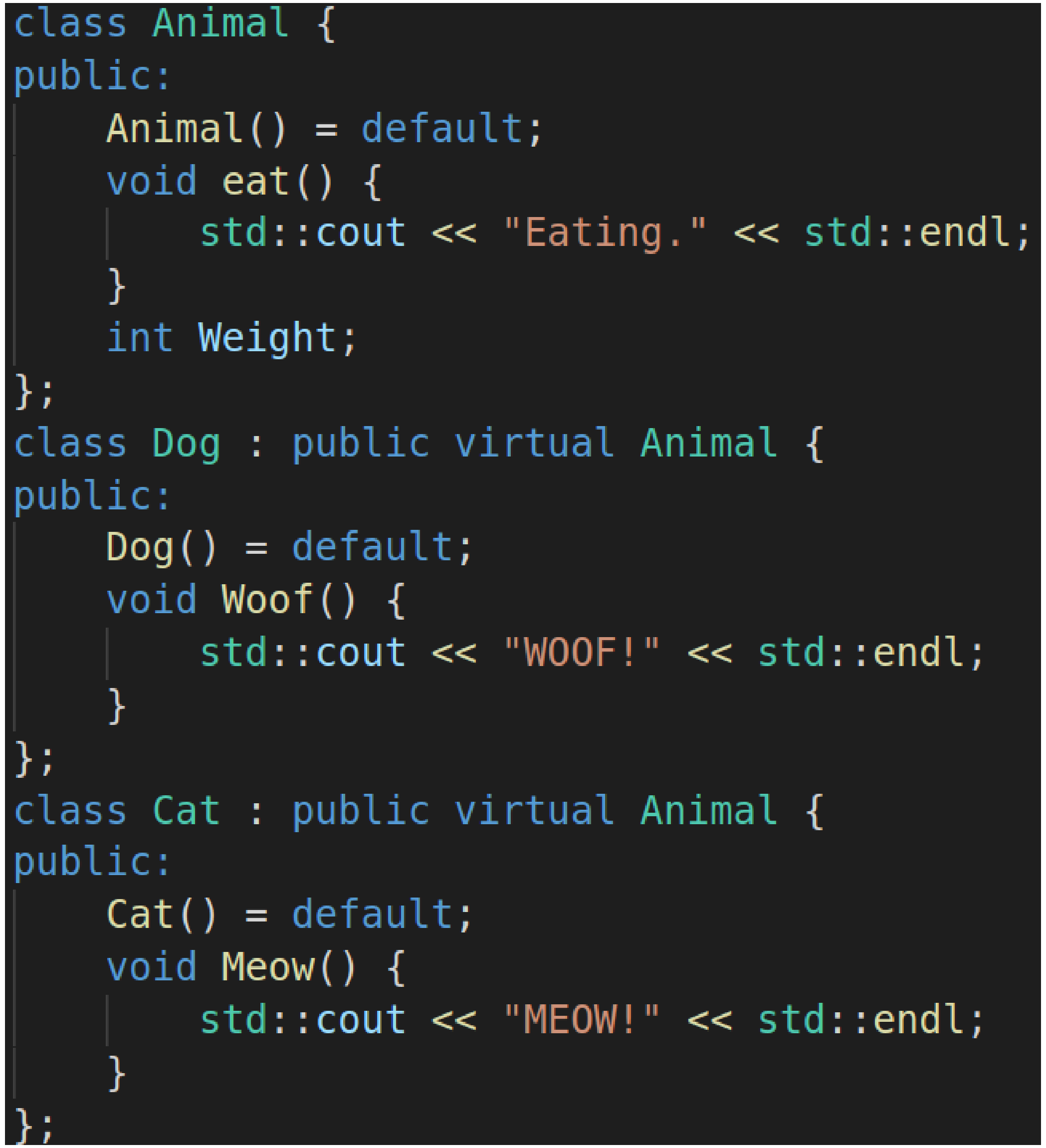


```cpp
#include "acc.h"

void externalPrint(const Foo &_f) {
    std::cout << "External print, printing FOO: "<< _f.x << std::endl;
}

void Foo::printBar(const Bar &_b) {
    std::cout << "Printing BAR from within FOO: "<< _b.x << std::endl;
}

int main() {
    Foo f;
    Bar b;
    externalPrint(f);
    f.printBar(b);
    return 0;
}
```

Diamond & Virtual

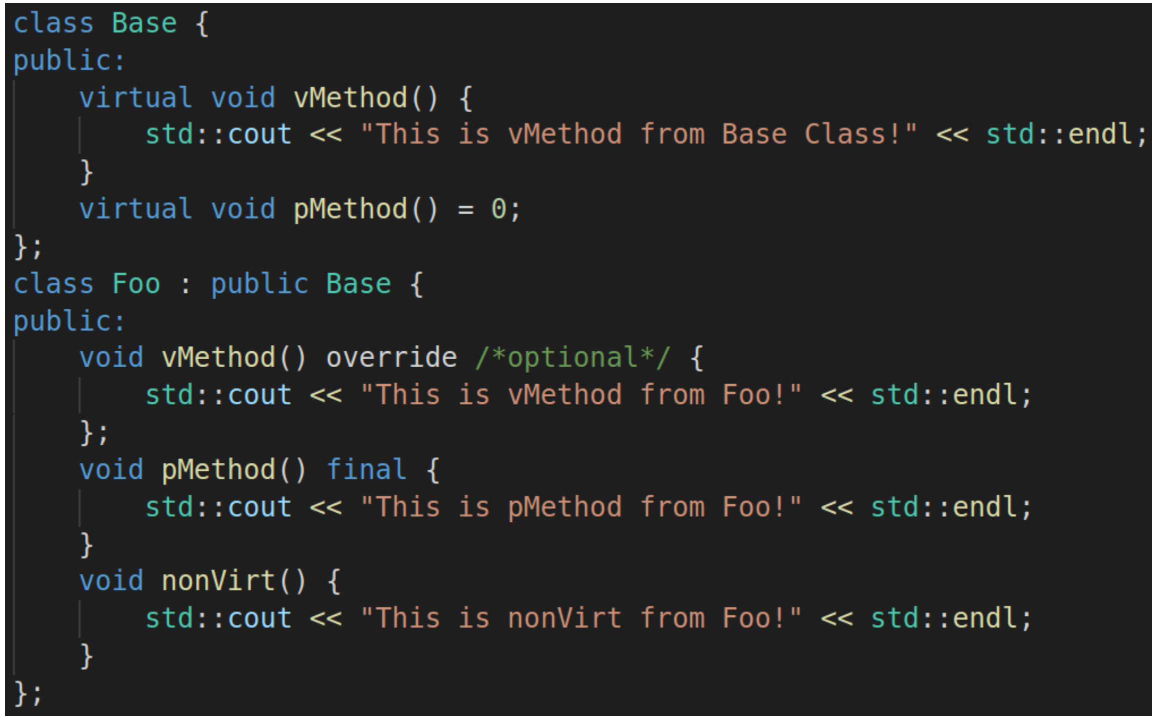


```cpp
class CatDog : public Cat, public Dog {
public:
    CatDog() = default;
};

int main() {
    CatDog catDog;

    catDog.Woof();
    catDog.Meow();
    catDog.eat();
    //catDog.Weight = 3;

    return 0;
}
```

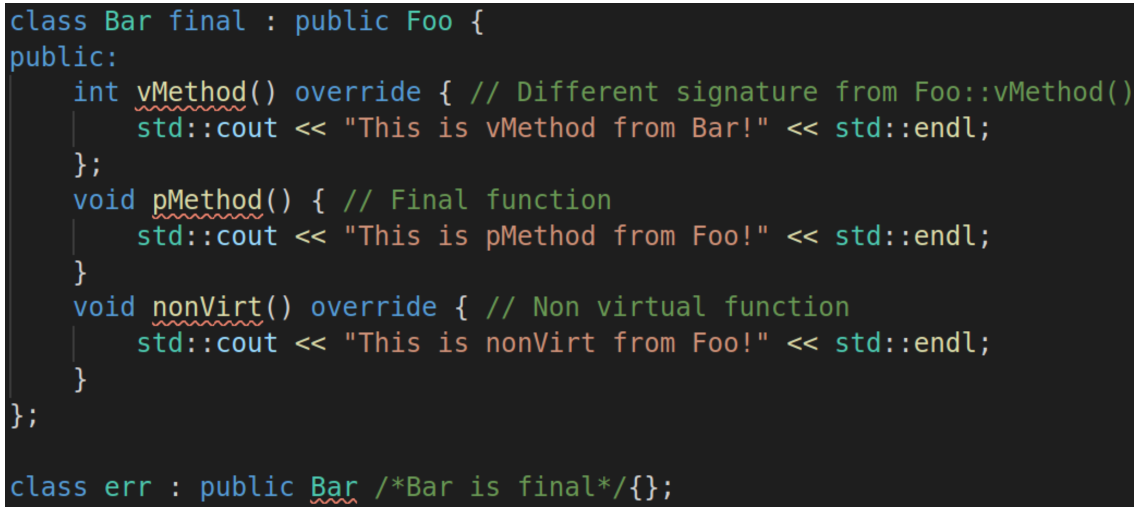


```cpp
class Animal {
public:
    Animal() = default;
    void eat() {
        std::cout << "Eating." << std::endl;
    }
    int Weight;
};
class Dog : public virtual Animal {
public:
    Dog() = default;
    void Woof() {
        std::cout << "WOOF!" << std::endl;
    }
};
class Cat : public virtual Animal {
public:
    Cat() = default;
    void Meow() {
        std::cout << "MEOW!" << std::endl;
    }
};
```

final & override



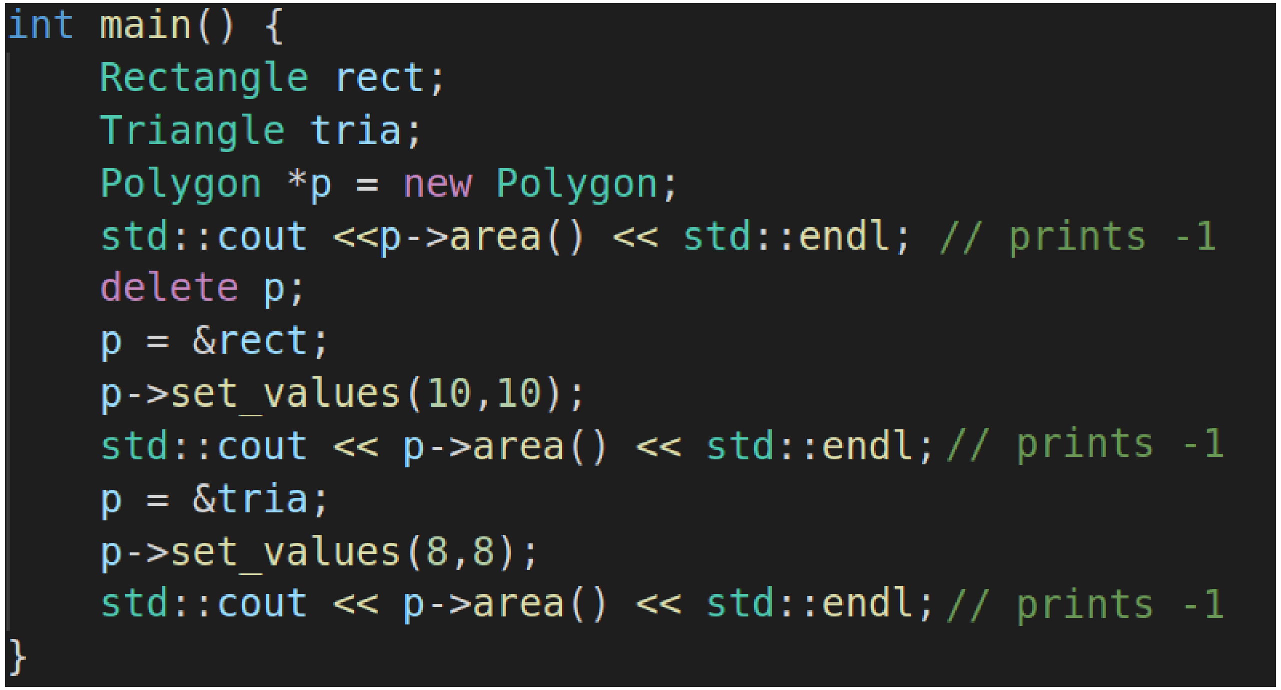
```
class Base {
public:
    virtual void vMethod() {
        std::cout << "This is vMethod from Base Class!" << std::endl;
    }
    virtual void pMethod() = 0;
};
class Foo : public Base {
public:
    void vMethod() override /*optional*/ {
        std::cout << "This is vMethod from Foo!" << std::endl;
    };
    void pMethod() final {
        std::cout << "This is pMethod from Foo!" << std::endl;
    }
    void nonVirt() {
        std::cout << "This is nonVirt from Foo!" << std::endl;
    }
};
```



```
class Bar final : public Foo {
public:
    int vMethod() override { // Different signature from Foo::vMethod()
        std::cout << "This is vMethod from Bar!" << std::endl;
    };
    void pMethod() { // Final function
        std::cout << "This is pMethod from Foo!" << std::endl;
    }
    void nonVirt() override { // Non virtual function
        std::cout << "This is nonVirt from Foo!" << std::endl;
    }
};

class err : public Bar /*Bar is final*/{};
```

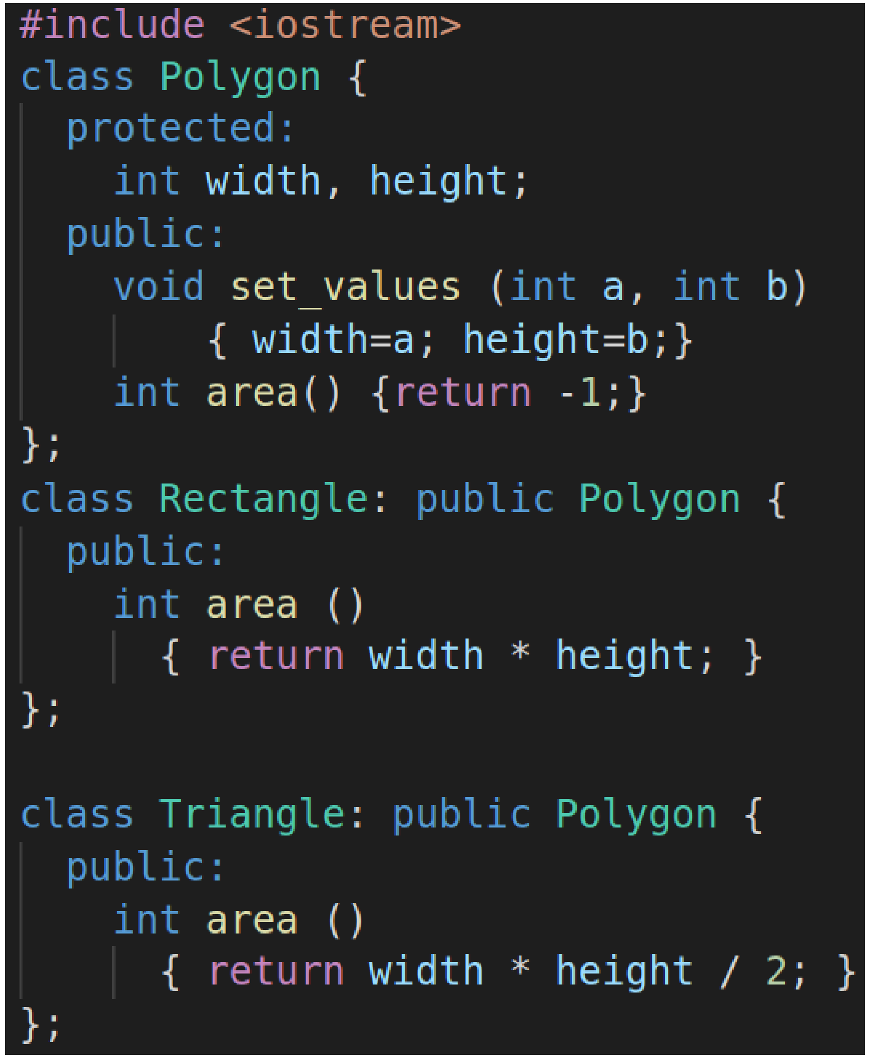
Base Class Type

- In case both *base* and *derived* class implement the same function, they **datatype** of the object upon invocation of the function determines which implementation would be invoked.



```cpp
int main() {
    Rectangle rect;
    Triangle tria;
    Polygon *p = new Polygon;
    std::cout <<p->area() << std::endl; // prints -1
    delete p;
    p = &rect;
    p->set_values(10,10);
    std::cout << p->area() << std::endl;// prints -1
    p = &tria;
    p->set_values(8,8);
    std::cout << p->area() << std::endl;// prints -1
}
```

- There is no way to reach the specific implementation of **area** function in *derived* classes from the *base* class pointer.

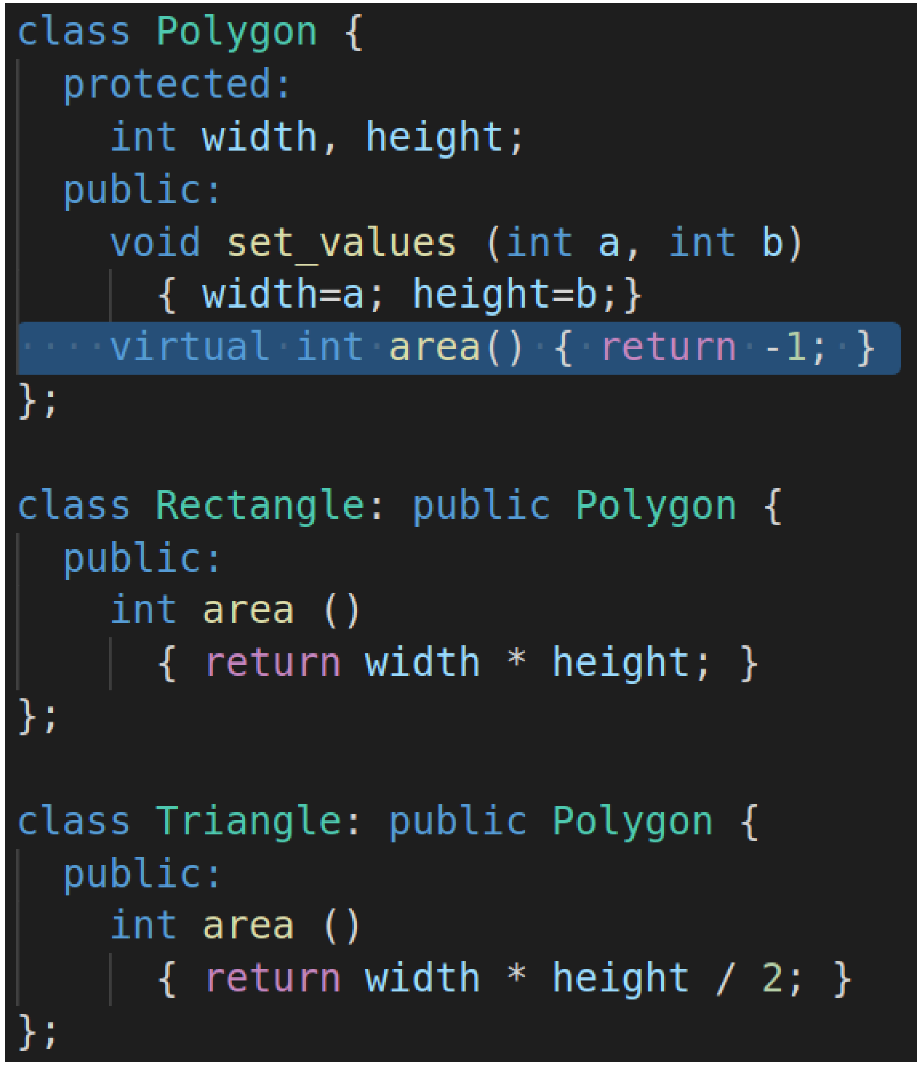


```cpp
#include <iostream>
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
        { width=a; height=b;}
    int area() {return -1;}
};
class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

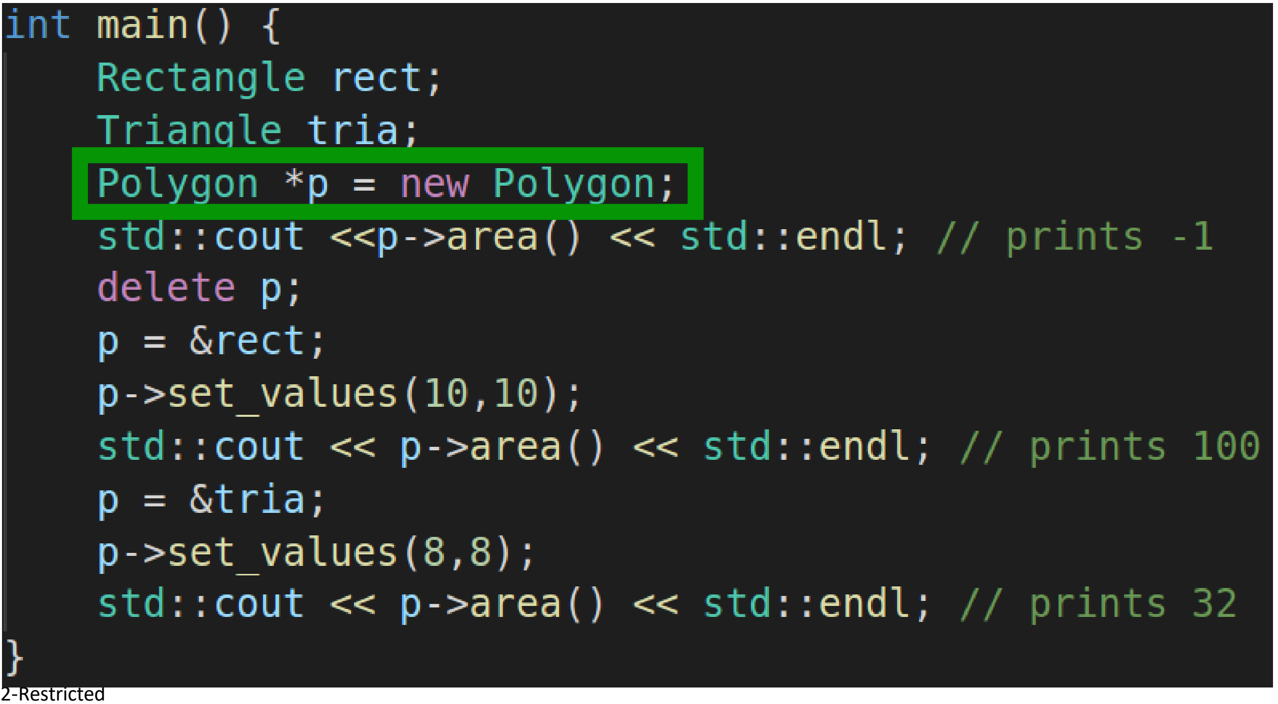
Polymorphic Class

- A class defining or inheriting a `virtual` function is called a ***polymorphic*** class.

- Polymorphic classes could be instantiated.

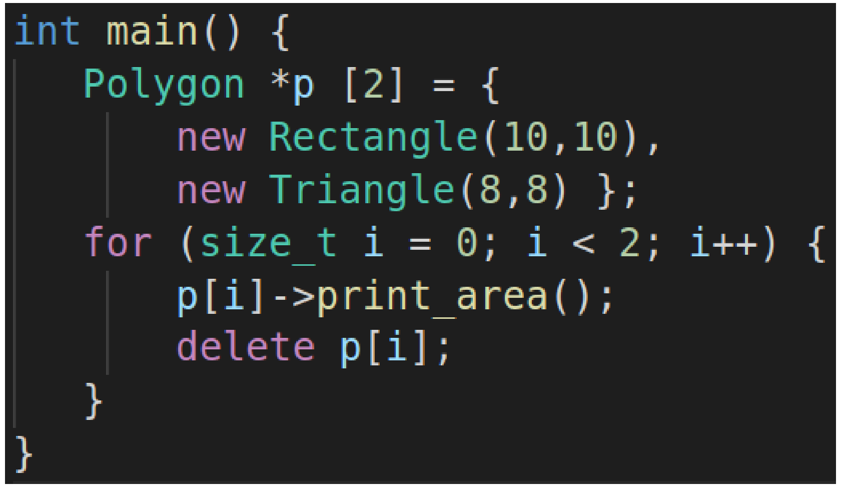


```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
    virtual int area() { return -1; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```



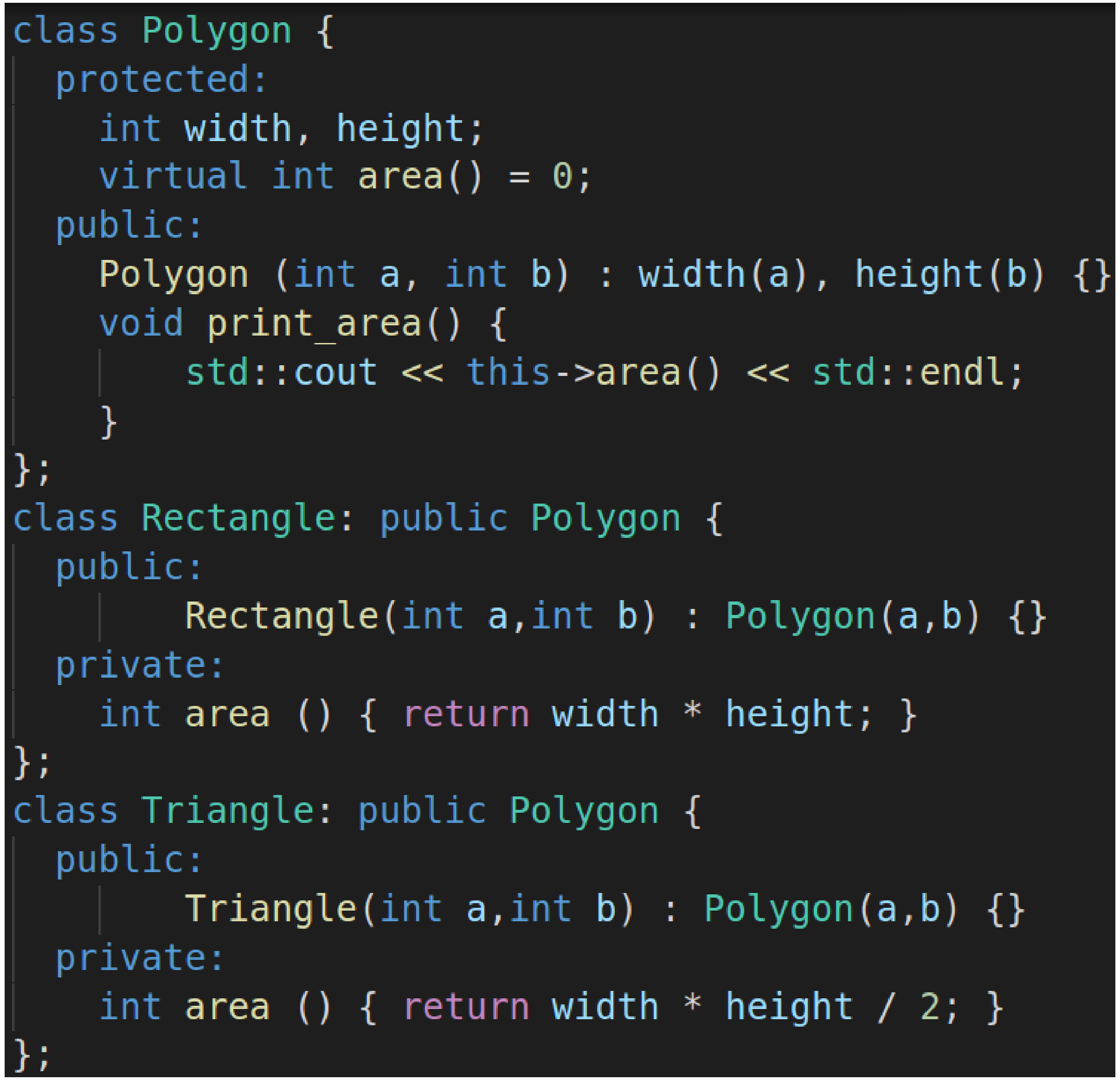
```
int main() {
    Rectangle rect;
    Triangle tria;
    Polygon *p = new Polygon;
    std::cout <<p->area() << std::endl; // prints -1
    delete p;
    p = &rect;
    p->set_values(10,10);
    std::cout << p->area() << std::endl; // prints 100
    p = &tria;
    p->set_values(8,8);
    std::cout << p->area() << std::endl; // prints 32
}
```

Abstract Class



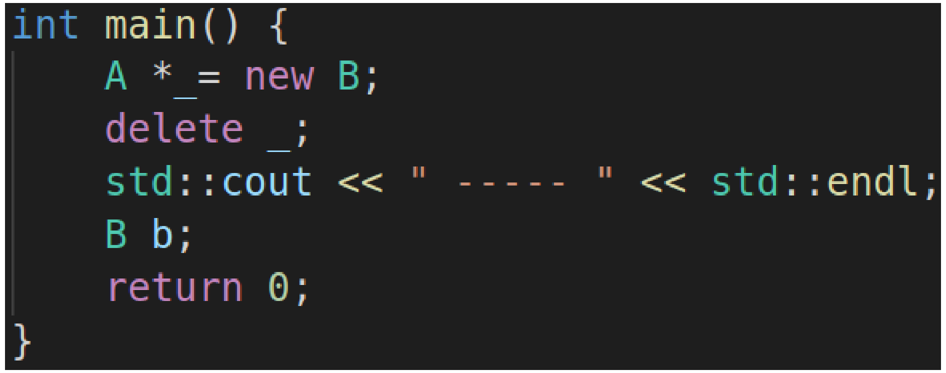
```cpp
int main() {
    Polygon *p [2] = {
        new Rectangle(10,10),
        new Triangle(8,8) };
    for (size_t i = 0; i < 2; i++) {
        p[i]->print_area();
        delete p[i];
    }
}
```

- A call to a pure virtual or an abstract function within the abstract class is allowed. At runtime, the appropriate implementation of the function (depending on which object of derived class the pointer of the base class refer to) would be invoked.

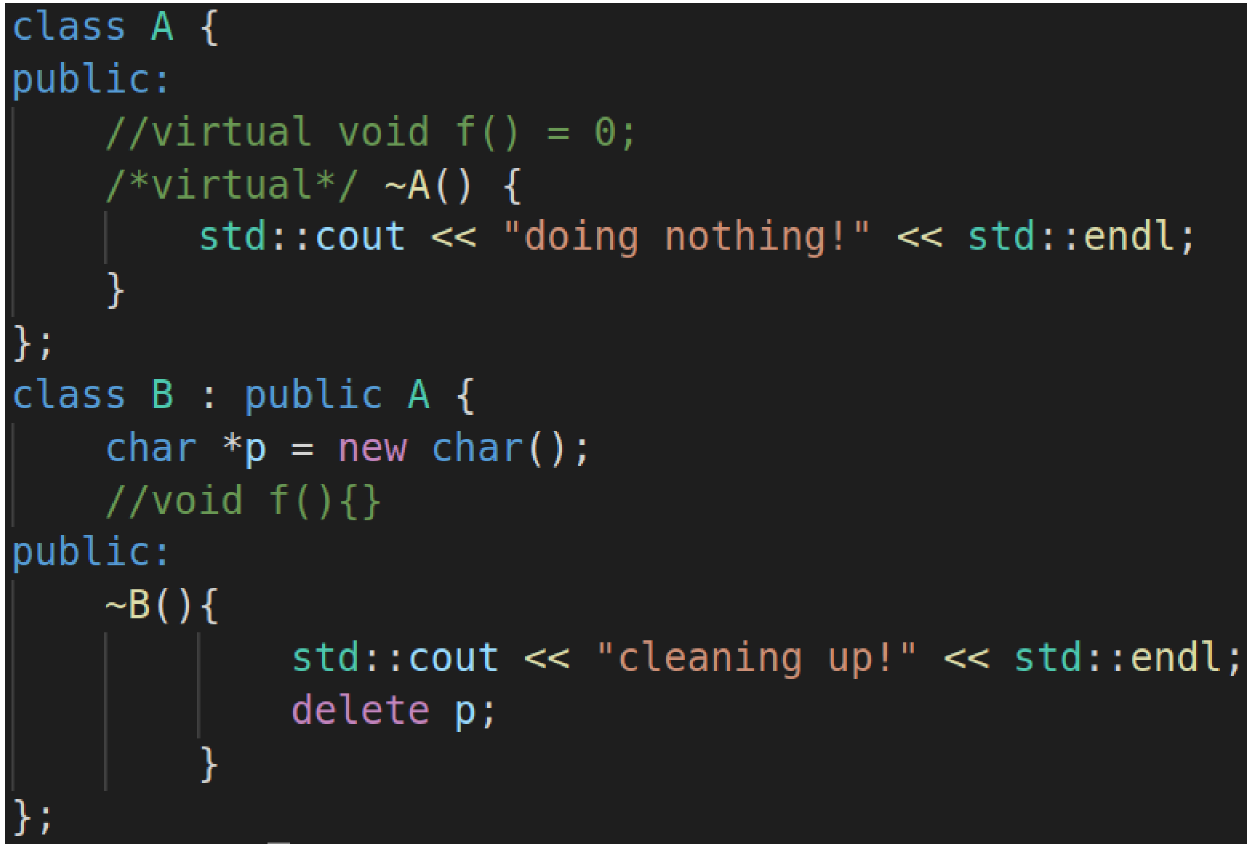


```cpp
class Polygon {
  protected:
    int width, height;
    virtual int area() = 0;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    void print_area() {
        std::cout << this->area() << std::endl;
    }
};
class Rectangle: public Polygon {
  public:
        Rectangle(int a,int b) : Polygon(a,b) {}
  private:
    int area () { return width * height; }
};
class Triangle: public Polygon {
  public:
        Triangle(int a,int b) : Polygon(a,b) {}
  private:
    int area () { return width * height / 2; }
};
```

Virtual Destructor



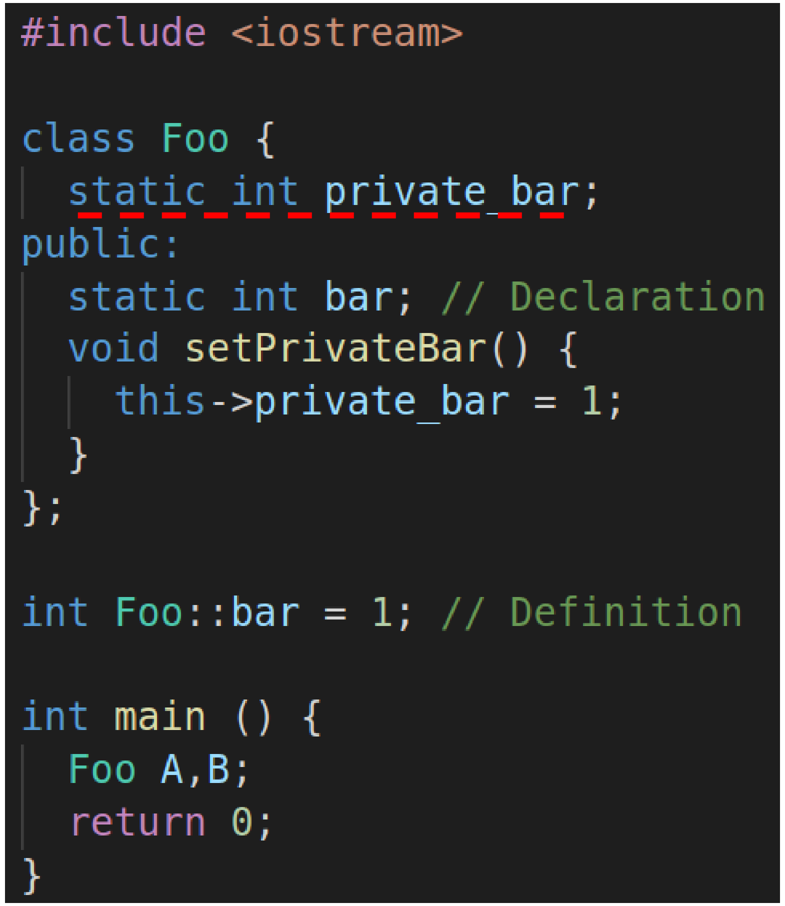
```
int main() {
    A *_= new B;
    delete _;
    std::cout << " ----- " << std::endl;
    B b;
    return 0;
}
```



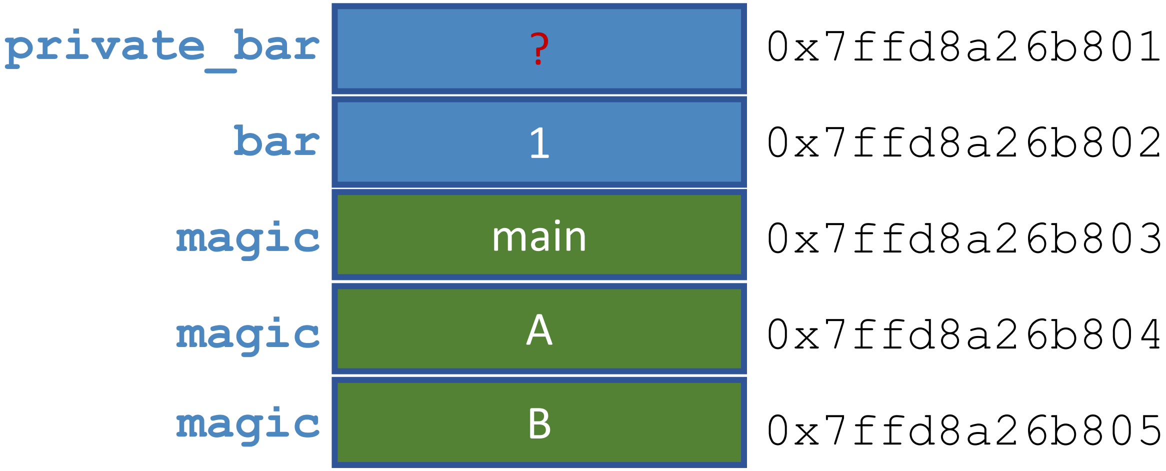
```
class A {
public:
    //virtual void f() = 0;
    /*virtual*/ ~A() {
        std::cout << "doing nothing!" << std::endl;
    }
};
class B : public A {
    char *p = new char();
    //void f(){}
public:
    ~B(){
            std::cout << "cleaning up!" << std::endl;
            delete p;
        }
};
```

static

- Classes can also have members declared with keyword **static** to store them in *static* memory.



```
#include <iostream>

class Foo {
  static int private_bar;
public:
  static int bar; // Declaration
  void setPrivateBar() {
    this->private_bar = 1;
  }
};

int Foo::bar = 1; // Definition

int main () {
  Foo A,B;
  return 0;
}
```

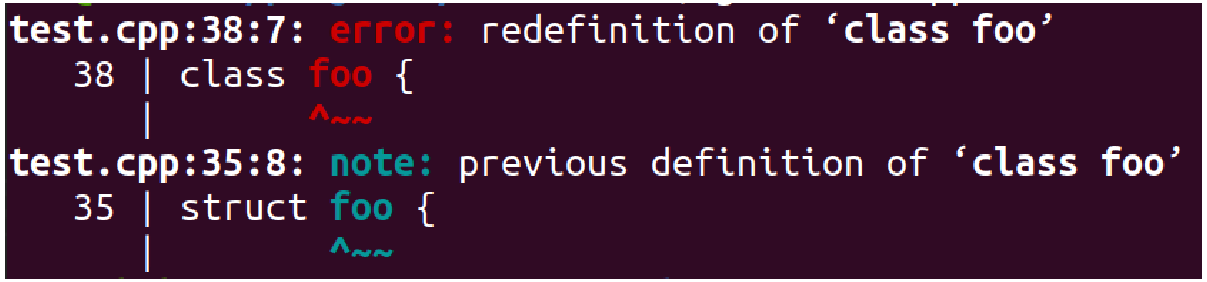


private_bar	?	0x7ffd8a26b801
bar	1	0x7ffd8a26b802
magic	main	0x7ffd8a26b803
magic	A	0x7ffd8a26b804
magic	B	0x7ffd8a26b805

- Static variables insides a class are not associated with the object of the class as they are allocated in different region of memory and treated as independent variables.
- Static variables are *shared* by all the objects of the class, and cannot be initialized by the constructor, and shall be initialized explicitly.
- Static variables must be defined separately (in source code), and do not need to have **static** keyword any more.

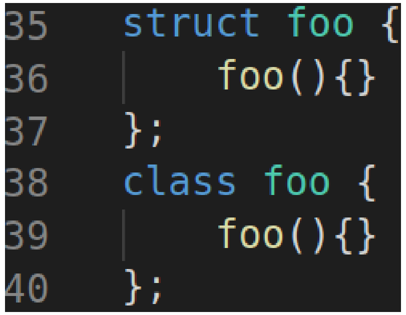
struct vs class

- To define a class (object) either **struct** or **class** keyword could be used.
- Default access for memebrs of **class** is **private**, as it is for inheritance.
- Default access for memebrs of **struct** is **public**, as it is for inheritance.

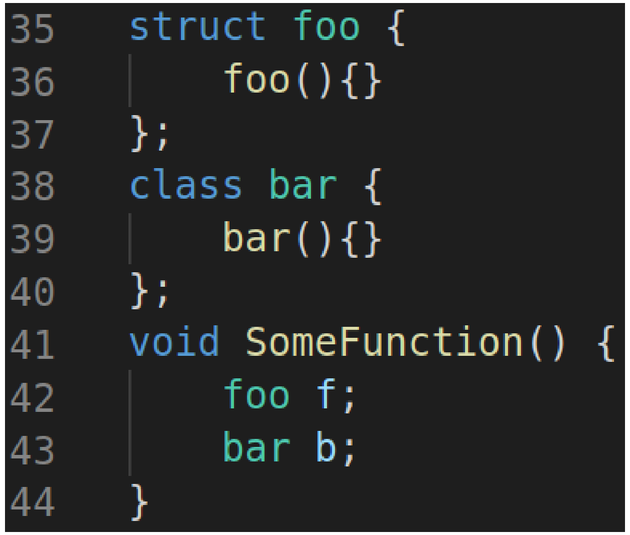


```
test.cpp:38:7: error: redefinition of ‘class foo’
   38 | class foo {
      |       ^~~
test.cpp:35:8: note: previous definition of ‘class foo’
   35 | struct foo {
      |        ^~~
```

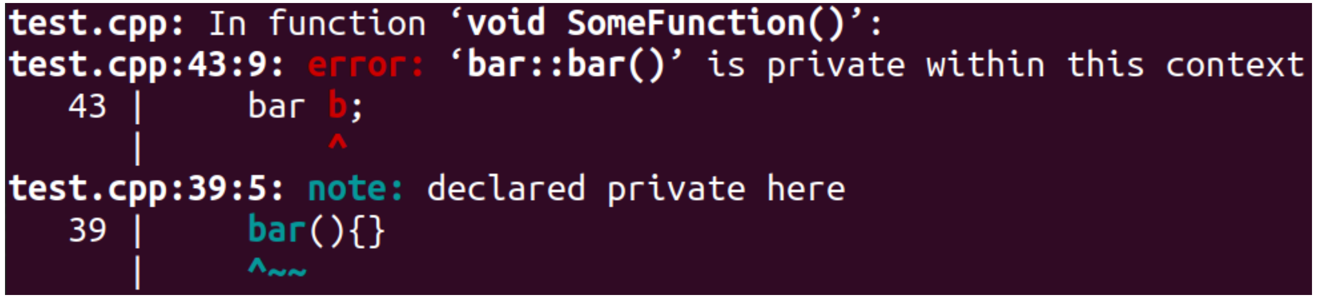
- Although both **class** and **struct** keywords could be used interchangeably, yet the keyword **struct** is generally used to specify plain data structures.
- Classes could also be defined using **union**, however only one data member is accessible at a time.



```
35   struct foo {
36       foo(){}
37   };
38   class foo {
39       foo(){}
40   };
```



```
35   struct foo {
36       foo(){}
37   };
38   class bar {
39       bar(){}
40   };
41   void SomeFunction() {
42       foo f;
43       bar b;
44   }
```



```
test.cpp: In function ‘void SomeFunction()’:
test.cpp:43:9: error: ‘bar::bar()’ is private within this context
   43 |     bar b;
      |         ^
test.cpp:39:5: note: declared private here
   39 |     bar(){}
      |     ^~~
```