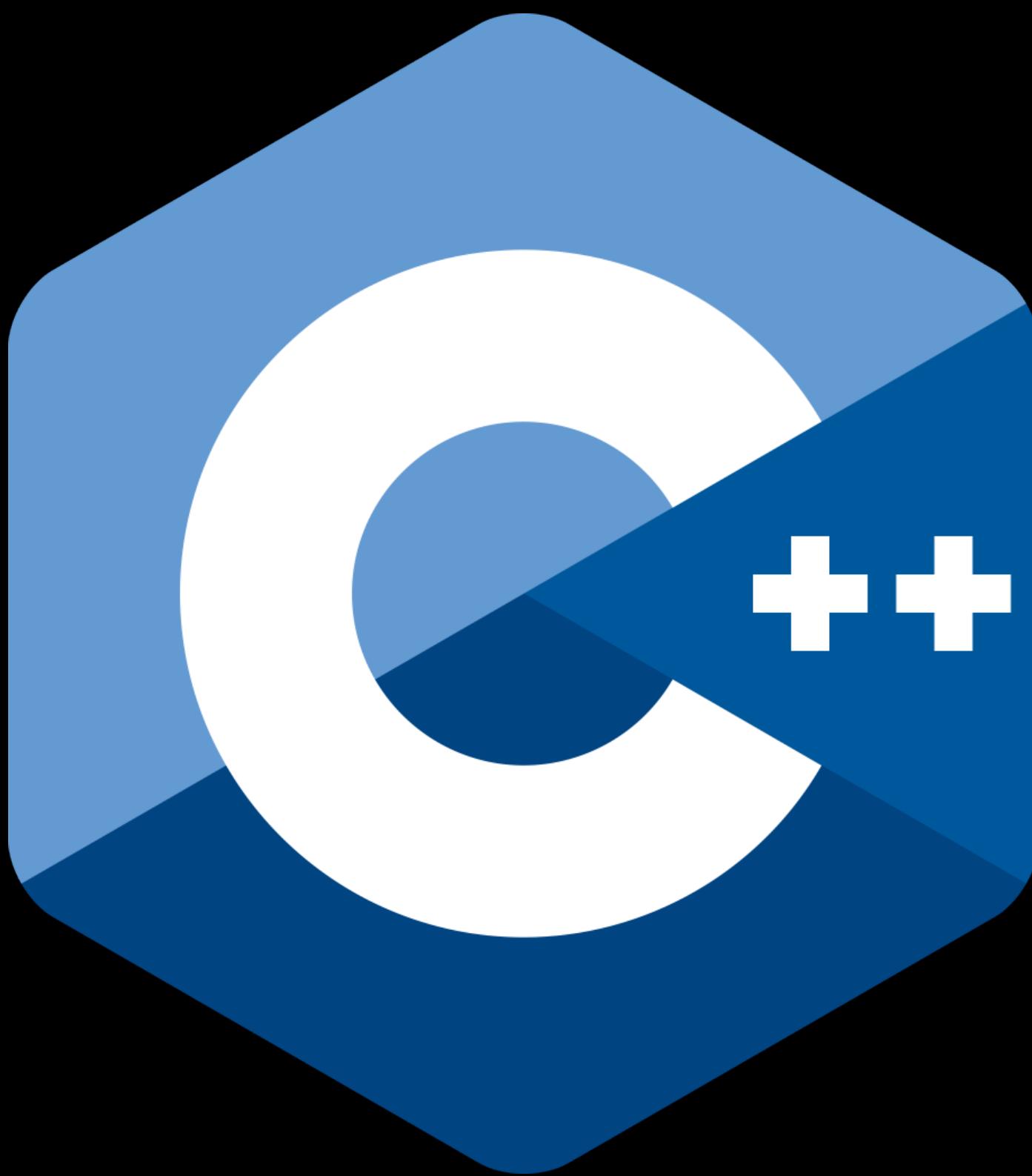


Modern C++ Course



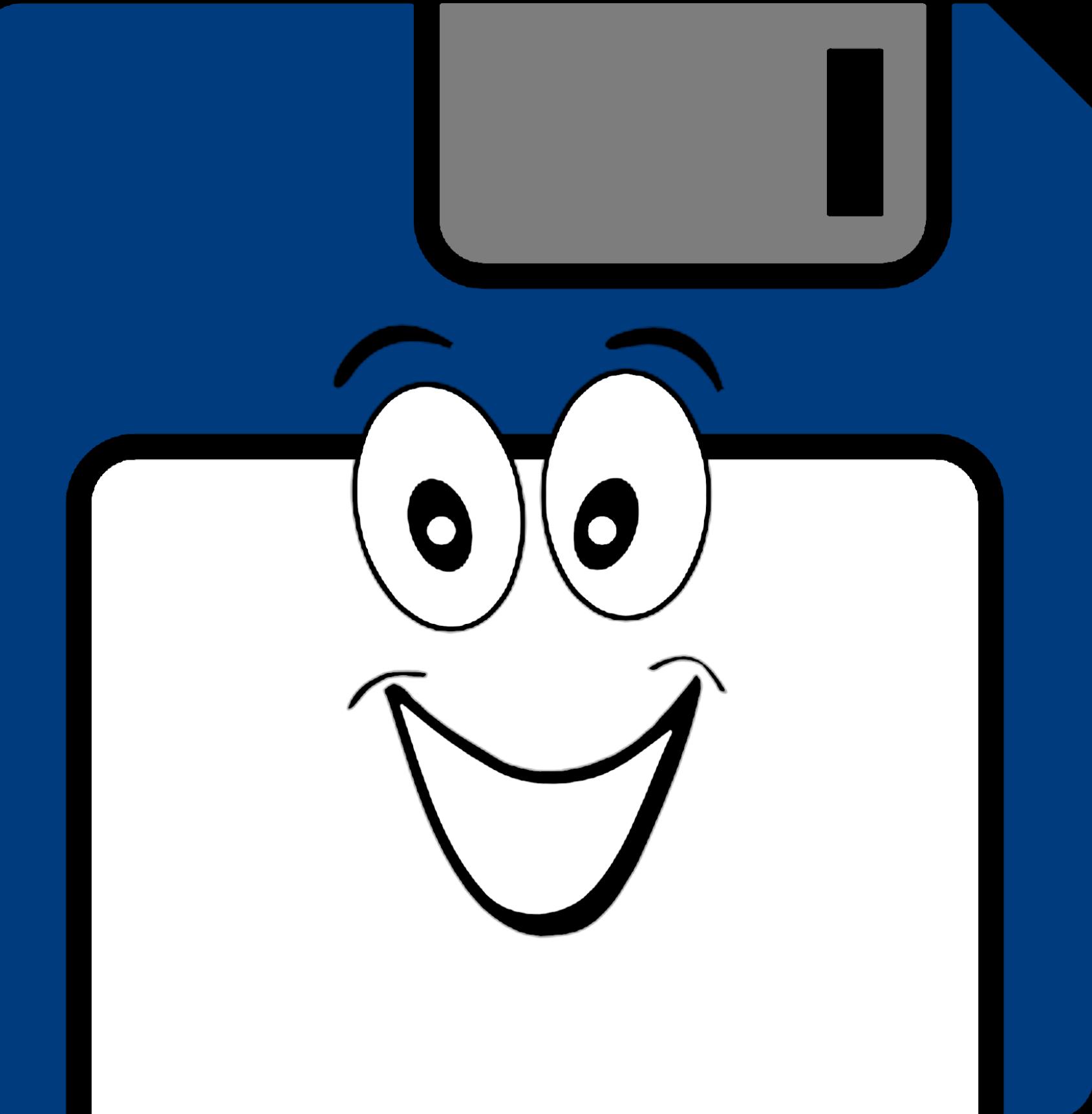
Who am I ?

Gammasoft

Gammasoft aims to make c++ fun again.

About

- Gammasoft is the nickname of Yves Fiumefreddo.
- More than thirty years of passion for high technology especially in development (c++, c#, objective-c, ...).
- Object-oriented programming is more than a mindset.
- more info see my GitHub : <https://github.com/gammasoft71>



Outline

1. Introduction
2. Language Basics
3. Object Oriented Programming (OOP)
4. Core Modern C++
5. Modern C++ Expert
6. Advanced Programming



Outline

1. Introduction
2. Language Basics
3. Object Oriented Programming (OOP)
4. Core Modern C++
5. Modern C++ Expert
6. Advanced Programming



Outline

1. Introduction
2. Language Basics
3. Object Oriented Programming (OOP)
4. Core Modern C++
5. Modern C++ Expert
6. Advanced Programming



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



The first classic application

The first classic application

program.cpp

```
● ● ●

#include <iostream>

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



The first classic application

program.cpp

```
● ● ●  
#include <iostream>  
  
int main() {  
    std::cout << "Hello, World!" << std::endl;  
}
```

CMakeLists.txt

```
● ● ●  
cmake_minimum_required(VERSION 3.20)  
  
project(hello_world)  
add_executable(${PROJECT_NAME} program.cpp)
```



The first classic application

program.cpp

```
● ● ●  
#include <iostream>  
  
int main() {  
    std::cout << "Hello, World!" << std::endl;  
}
```

CMakeLists.txt

```
● ● ●  
cmake_minimum_required(VERSION 3.20)  
  
project(hello_world)  
add_executable(${PROJECT_NAME} program.cpp)
```

Output

```
● ● ●  
> Hello, World!
```



The first classic application

program.cpp

```
● ● ●  
#include <iostream>  
  
auto main() -> int {  
    std::println("Hello, World!");  
}
```

CMakeLists.txt

```
● ● ●  
cmake_minimum_required(VERSION 3.20)  
  
project(hello_world)  
set(CMAKE_CXX_STANDARD 23)  
set(CMAKE_CXX_STANDARD_REQUIRED ON)  
add_executable(${PROJECT_NAME} program.cpp)
```

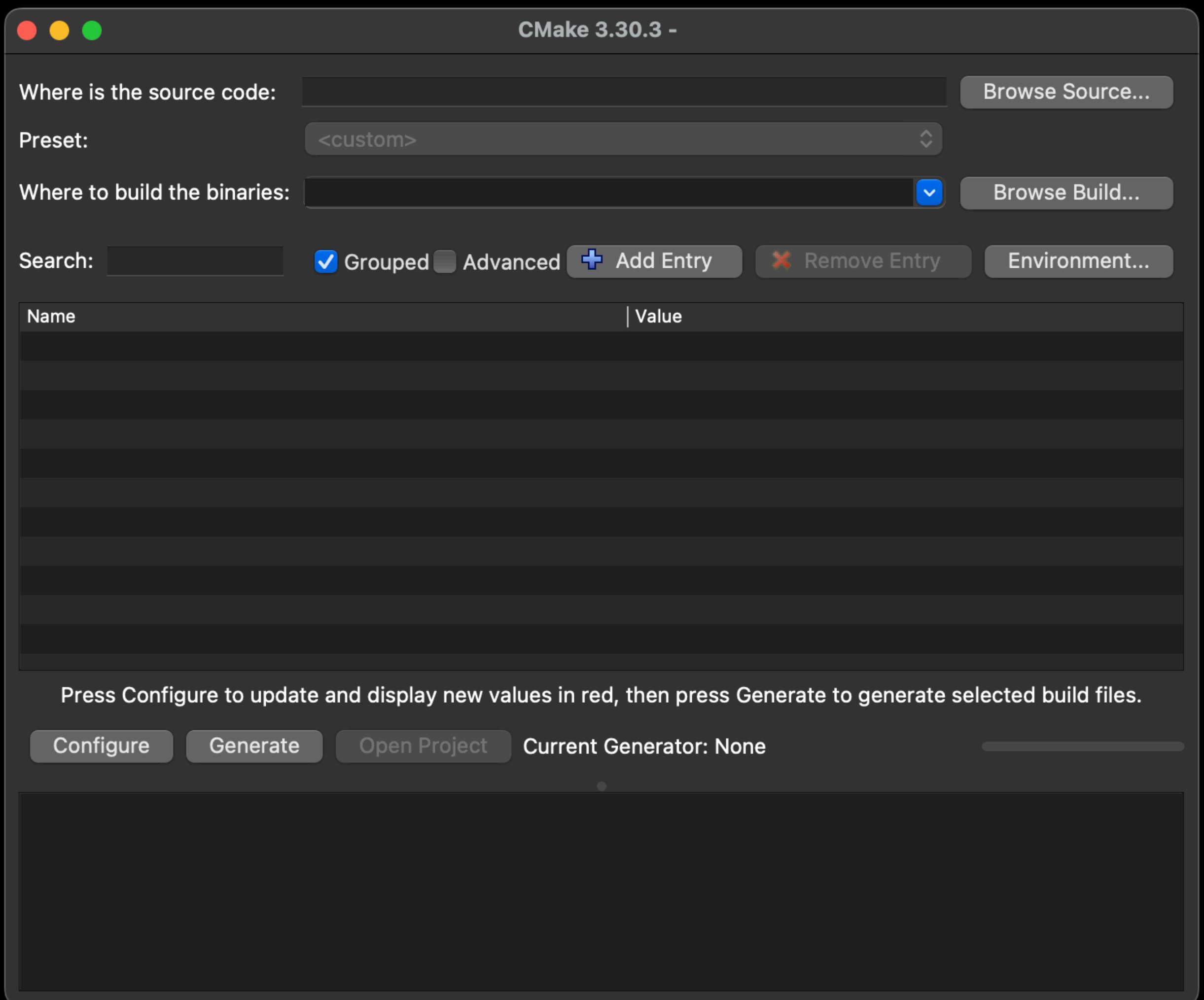
Output

```
● ● ●  
> Hello, World!
```



Execute with CMake GUI

- Open the CMake GUI application
- Click on "Browse Sources..." button and select the "Hello world" folder.
- Copy the path from "Where is the source code:", past it in the "Where to build the binaries:" and add "/build".
- Click on "Configure" button and select the generator.
- Click on "Generate" button to generate the project.
- Finally, click on “Open project” button.



Main function



```
#include <iostream>

int main() {
    std::cout << "main without arguments" << std::endl;
}
```

Main function



```
#include <iostream>

int main() {
    std::cout << "main without arguments" << std::endl;
}
```



```
#include <iostream>

int main( int argc, char* argv[] ) {
    std::cout << "main with argc and argv arguments" << std::endl;
}
```

Exercise : environment

Learn how to use CMake to generate a project.

- Go to exercises/environment
- Look at environment.cpp and CMakeLists.txt
- Compile it (make) and run the program (./environment)
- Work on the tasks that you find in environment.cpp



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Comments



```
// single-line comment
int value = 0;

/*
 * multi-line comment
 */
std::string name();

/// Doxygen comments
/// @brief Adds two specified integers.
/// @param a the first integer to add.
/// @param b the second integer to add.
/// @return The result of the addition.
/// @see https://www.doxygen.nl/manual/commands.html
int add(int a, int b);
```



Basic types



```
bool b = true; // boolean, true or false

char c = 'a';          // min 8 bit integer
char cs = -1;          // may be signed
char cu = '\2';         // or not
                        // can store an ASCII character

signed char sc = -3;   // min 8 bit signed integer
unsigned char uc = 4;  // min 8 bit unsigned integer

short int si = -5;      // min 16 bit signed integer
short s = -6;           // int is optional
unsigned short int usi = 7; // min 16 bit unsigned integer
unsigned short us = 8;    // int is optional
```



Basic types



```
int i = -9;           // min 16, usually 32 bit
unsigned int ui = 10; // min 16, usually 32 bit

long l = -11l;        // min 32 bit signed integer
long int li = -12l;   // int is optional
unsigned long ul = 13Ul; // min 32 bit unsigned integer
unsigned long int uli = 14Ul; // int is optional

long long ll = -15ll;    // min 64 bit signed integer
long long int lli = -16ll; // int is optional
unsigned long long ull = 17ull; // min 64 bit unsigned integer
unsigned long long int ulli = 18ull; // int is optional
```



Basic types



```
float f = 0.19f;           // 32 (1+8+23) bit float
double d = 0.20;           // 64 (1+11+52) bit float
long double ld = 0.21l;    // min 64 bit float

const char* nstr = "native string"; // array of chars ended by \0
std::string str = "string";        // class provided by the STL
```

Fixed width integer types



```
#include <cstdint>

std::int8_t i8 = -1;      // 8 bit signed integer
std::uint8_t ui8 = 1;     // 8 bit unsigned integer

std::int16_t i16 = -2;    // 16 bit signed integer
std::uint16_t ui16 = 3;   // 16 bit unsigned integer

std::int32_t i32 = -4;    // 32 bit signed integer
std::uint32_t ui32 = 5;   // 32 bit unsigned integer

std::int64_t i64 = -4;    // 64 bit signed integer
std::uint64_t ui64 = 5;   // 64 bit unsigned integer
```



Fixed width floating-point types



```
#include <stdfloat> // may define these:

std::float16_t value = 3.14f16;    // 16 (1+5+10) bit float
std::float32_t value = 3.14f32;    // like float (1+8+23)
                                  // but different type
std::float64_t value = 3.14f64;    // like double (1+11+52)
                                  // but different type
std::float128_t value = 3.14f128; // 128 (1+15+112) bit float
std::bfloating16_t value = 3.14bf16; // 16 (1+8+7) bit float

// also F16, F32, F64, F128 or BF16 suffix possible
```

Integer literals



```
int value = 4284;           // decimal (base 10)
int value = 0b0001000010111100; // binary (base 2) since C++14
int value = 010274;          // octal (base 8)
int value = 0x10bc;          // hexadecimal (base 16)
int value = 0x10BC;          // hexadecimal (base 16)

int value = 123'456'789;      // digit separators, since C++14
int value = 0b0001'0000'1011'1100; // digit separators, since C++14

4284           // int
4284u, 4284U // unsigned int
4284l, 4284L // long
4284ul, 4284UL // unsigned long
4284ll, 4284LL // long long
4284ull, 4284ULL // unsigned long long
```



Floating-point literals



```
double value = 12.34;
double value = 12.;
double value = .34;
double value = 12e34;           // 12 * 10^34
double value = 12E34;          // 12 * 10^34
double value = 12e-34;         // 12 * 10^-34
double value = 12.34e34;       // 12.34 * 10^34

double value = 123'456.789'101; // digit separators, C++14
double value = 0x4d2.4p3;       // hexfloat, 0x4d2.4 * 2^3
                                // = 1234.25 * 2^3 = 9874

3.14f, 3.14F, // float
3.14, 3.14,   // double
3.14l, 3.14L, // long double
```



Sizeof



```
#include <cstddef> // (and others) defines:  
  
int value = 42;  
std::size_t size = sizeof(value); // 4  
std::size_t size = sizeof(int); // 4  
std::size_t size = sizeof(42); // 4
```

Pointer to integer



```
#include <cstdint> // defines:  
  
int value1 = 42;  
int value2 = 84;  
  
// can hold any diff between two pointers  
std::ptrdiff_t ptrdiff = &value2 - &value1;  
  
// can hold any pointer value  
std::intptr_t intptr = reinterpret_cast<intptr_t>(&value1);  
std::uintptr_t uintptr = reinterpret_cast<uintptr_t>(&value2);
```

Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Static arrays



```
int ints[4] = {1, 2, 3, 4};  
int ints[] = {1, 2, 3, 4}; // identical  
  
char chars[3] = {'a', 'b', 'c'};    // char array  
char chars[4] = "abc";           // valid native string  
char chars[4] = {'a', 'b', 'c', 0}; // same valid native string  
  
int i = ints[2];   // i = 3  
char c = chars[8]; // at best garbage, may segfault  
int i = ints[4];   // also garbage !
```

Pointers



```
int i = 4;
int* pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int* pai = ai; // decay to pointer
int* paj = paj + 1;
int k = *paj + 1;

// compile error
int* pak = k;

// segmentation fault !
int* pak = (int*)k;
int l = *pak;
```



Pointers



```
int i = 4;
int* pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int* pai = ai; // decay to pointer
int* paj = paj + 1;
int k = *paj + 1;

// compile error
int* pak = k;

// segmentation fault !
int* pak = (int*)k;
int l = *pak;
```

Memory layout

	0x3028
	0x3024
	0x3020
	0x301C
	0x3018
	0x3014
	0x3010
	0x300C
	0x3008
	0x3004
i = 4	0x3000

Pointers



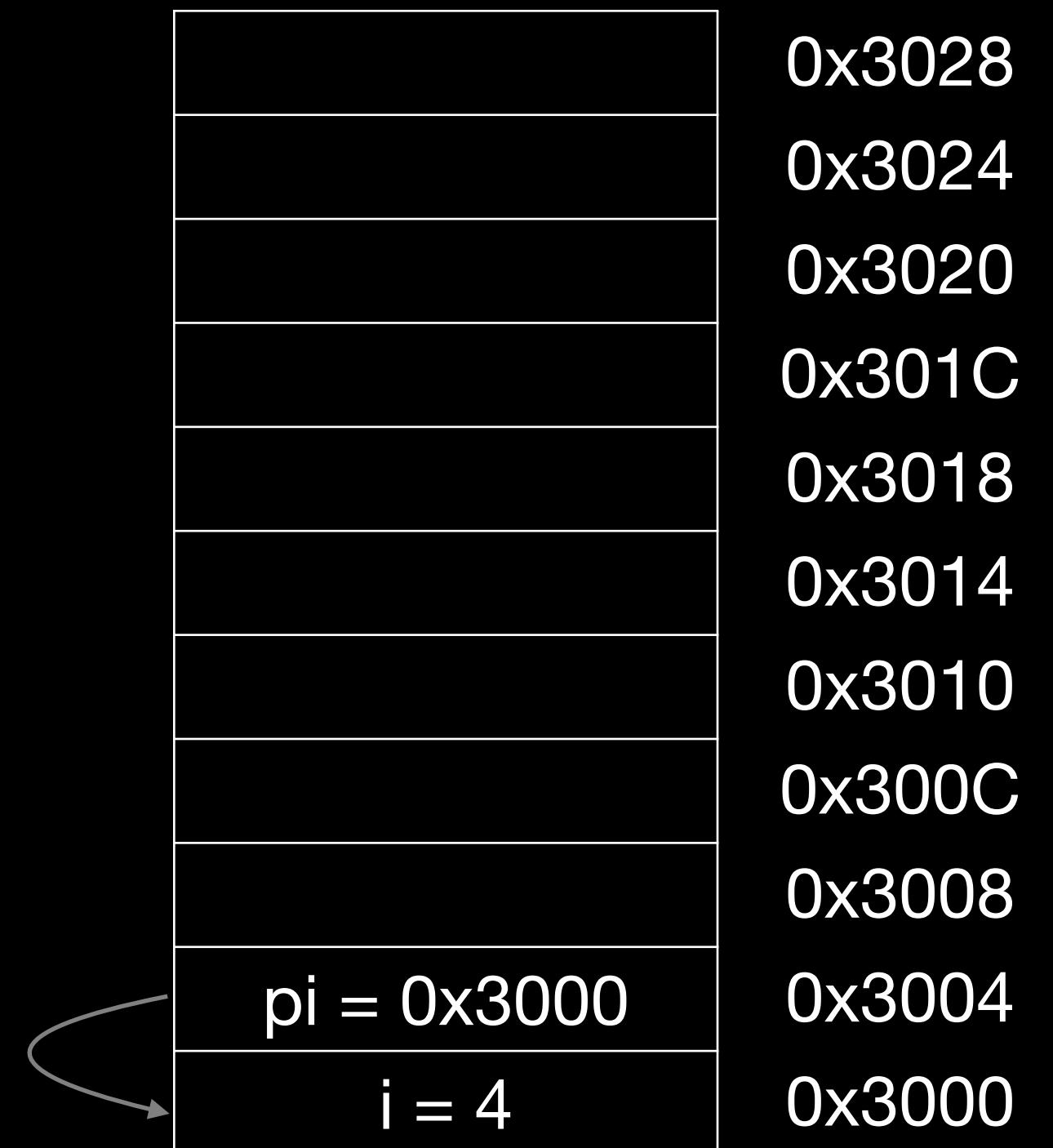
```
int i = 4;
int* pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int* pai = ai; // decay to pointer
int* paj = paj + 1;
int k = *paj + 1;

// compile error
int* pak = k;

// segmentation fault !
int* pak = (int*)k;
int l = *pak;
```

Memory layout



Pointers



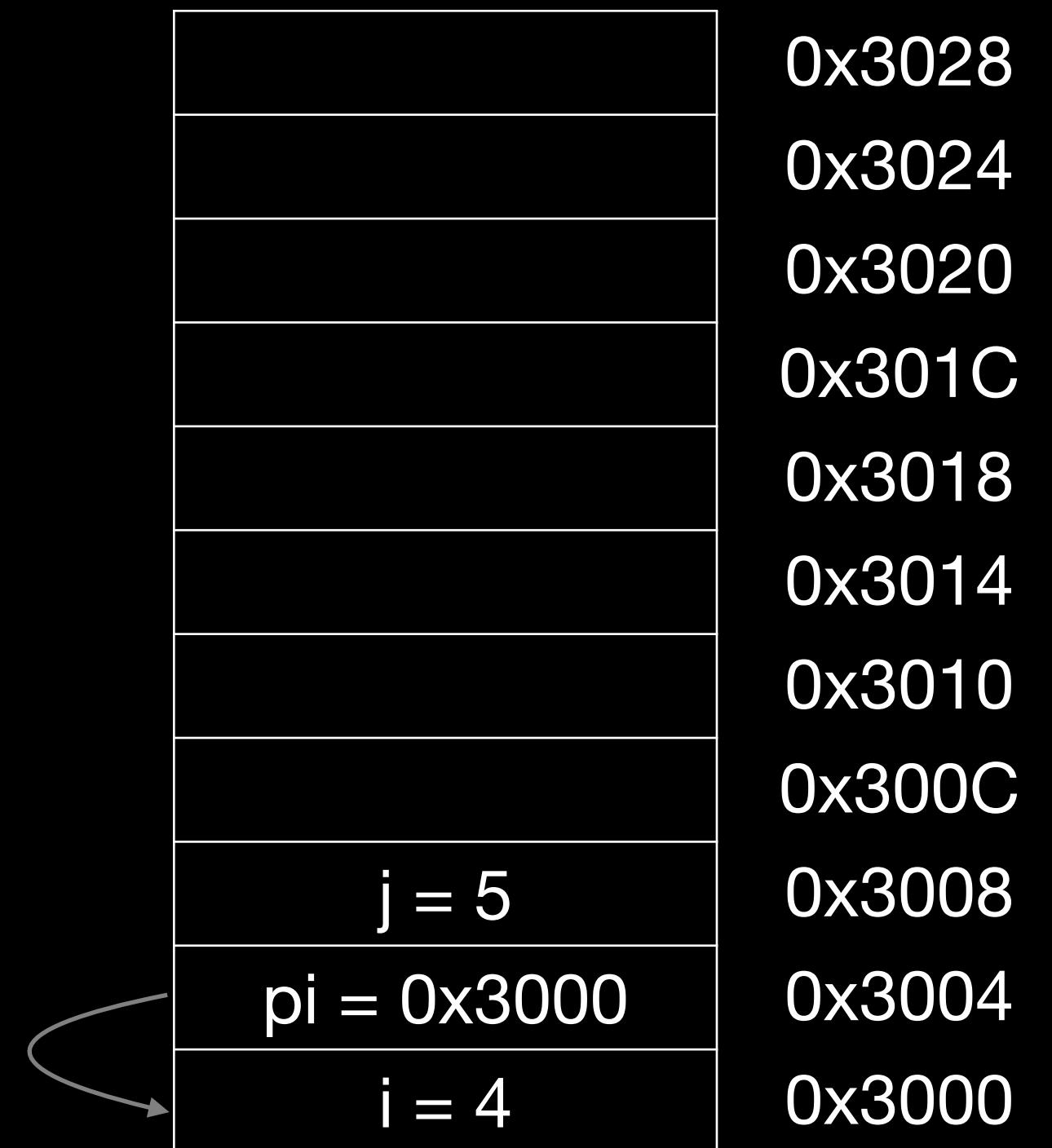
```
int i = 4;
int* pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int* pai = ai; // decay to pointer
int* paj = paj + 1;
int k = *paj + 1;

// compile error
int* pak = k;

// segmentation fault !
int* pak = (int*)k;
int l = *pak;
```

Memory layout



Pointers



```
int i = 4;
int* pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int* pai = ai; // decay to pointer
int* paj = paj + 1;
int k = *paj + 1;

// compile error
int* pak = k;

// segmentation fault !
int* pak = (int*)k;
int l = *pak;
```

Memory layout

	0x3028
	0x3024
	0x3020
	0x301C
	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000



Pointers



```
int i = 4;
int* pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int* pai = ai; // decay to pointer
int* paj = pai + 1;
int k = *paj + 1;

// compile error
int* pak = k;

// segmentation fault !
int* pak = (int*)k;
int l = *pak;
```

Memory layout

	0x3028
	0x3024
	0x3020
	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

Pointers



```
int i = 4;
int* pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int* pai = ai; // decay to pointer
int* paj = paj + 1;
int k = *paj + 1;

// compile error
int* pak = k;

// segmentation fault !
int* pak = (int*)k;
int l = *pak;
```

Memory layout

	0x3028
	0x3024
	0x3020
paj = 0x3010	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

Pointers



```
int i = 4;
int* pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int* pai = ai; // decay to pointer
int* paj = paj + 1;
int k = *paj + 1;

// compile error
int* pak = k;

// segmentation fault !
int* pak = (int*)k;
int l = *pak;
```

Memory layout

	0x3028
	0x3024
k = 3	0x3020
paj = 0x3010	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000



Pointers



```
int i = 4;
int* pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int* pai = ai; // decay to pointer
int* paj = paj + 1;
int k = *paj + 1;

// compile error
int* pak = k;

// segmentation fault !
int* pak = (int*)k;
int l = *pak;
```

Memory layout

??	0x3028
pak = 3	0x3024
k = 3	0x3020
paj = 0x3010	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

nullptr

- if a pointer doesn't point to anything, set it to `nullptr`
 - useful to e.g. mark the end of a linked data structure
 - or absence of an optional function argument (pointer)
- same as setting it to 0 or `NULL` (before C++ 11)
- triggers compilation error when assigned to integer



nullptr

- if a pointer doesn't point to anything, set it to `nullptr`
 - useful to e.g. mark the end of a linked data structure
 - or absence of an optional function argument (pointer)
- same as setting it to 0 or `NULL` (before C++ 11)
- triggers compilation error when assigned to integer



```
int* ip = nullptr;
int i = NULL; // compiles, bug?
int i = nullptr; // ERROR
```

Dynamic arrays using C



```
#include <cstdlib>
#include <cstring>

int* bad;          // pointer to random address
int* ai = nullptr; // better, deterministic, testable

// allocate array of 10 ints (uninitialized)
ai = (int*)malloc(10 * sizeof(int));
memset(ai, 0, 10 * sizeof(int)); // and set them to 0

ai = (int*)calloc(10, sizeof(int)); // both in one go

free(ai); // release memory
```



Dynamic arrays using C++



```
#include <cstdlib>
#include <cstring>

// allocate array of 10 ints
int* ai = new int[10];      // uninitialized
int* ai = new int[10] {};    // zero-initialized

delete[] ai; // release array memory

// allocate a single int
int* pi = new int;
int* pi = new int {};

delete pi; // release scalar memory
```



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Scope

Portion of the source code where a given name is valid

Typically :

- simple block of code, within {}
- function, class, namespace
- the global scope, i.e. translation unit (.cpp file + all includes)

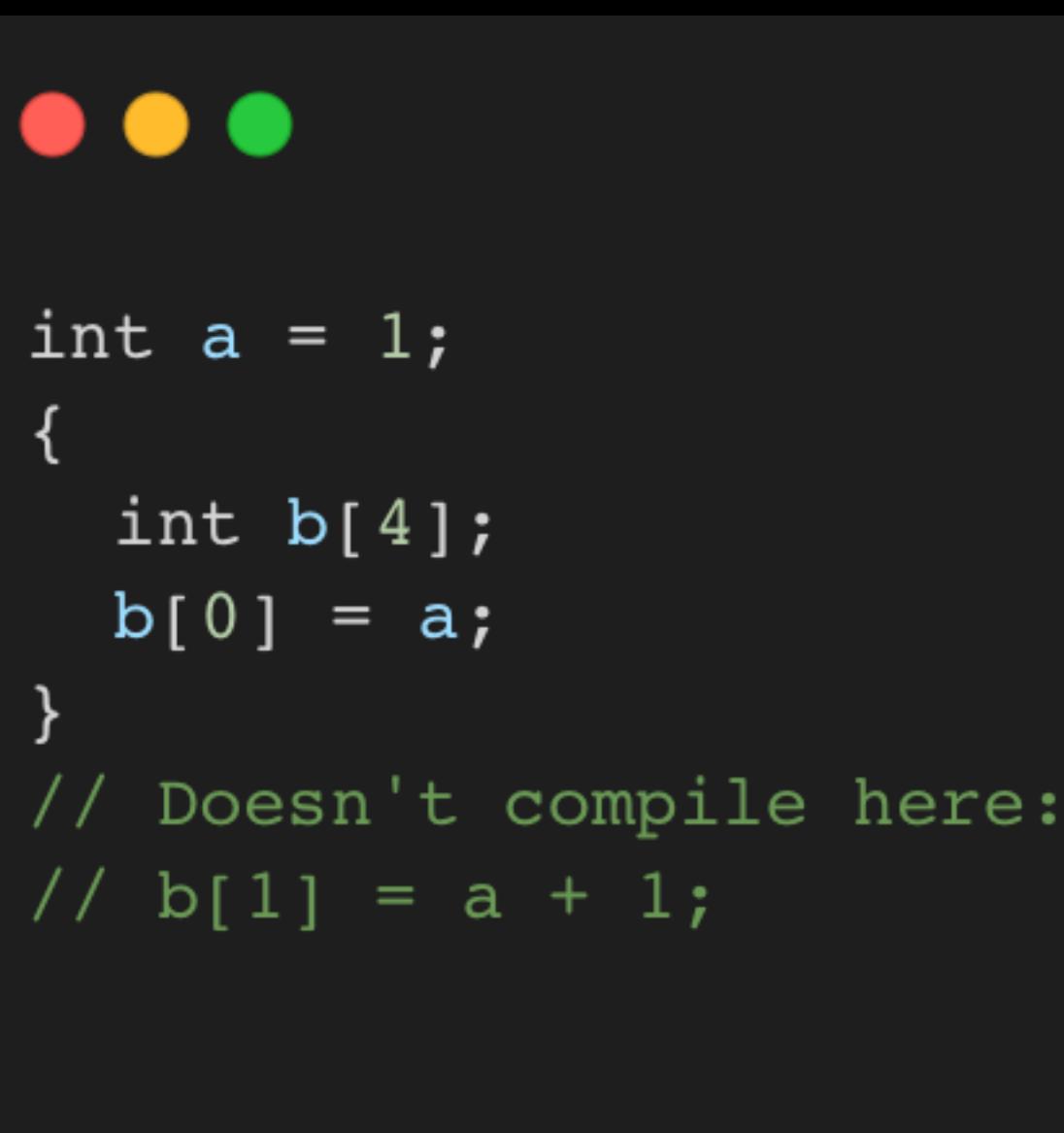
```
● ● ●  
{  
    int a = 0;  
    {  
        int b = 0;  
    } // end of b scope  
}  
// end of a scope
```

Scope and lifetime of variables

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope

Scope and lifetime of variables

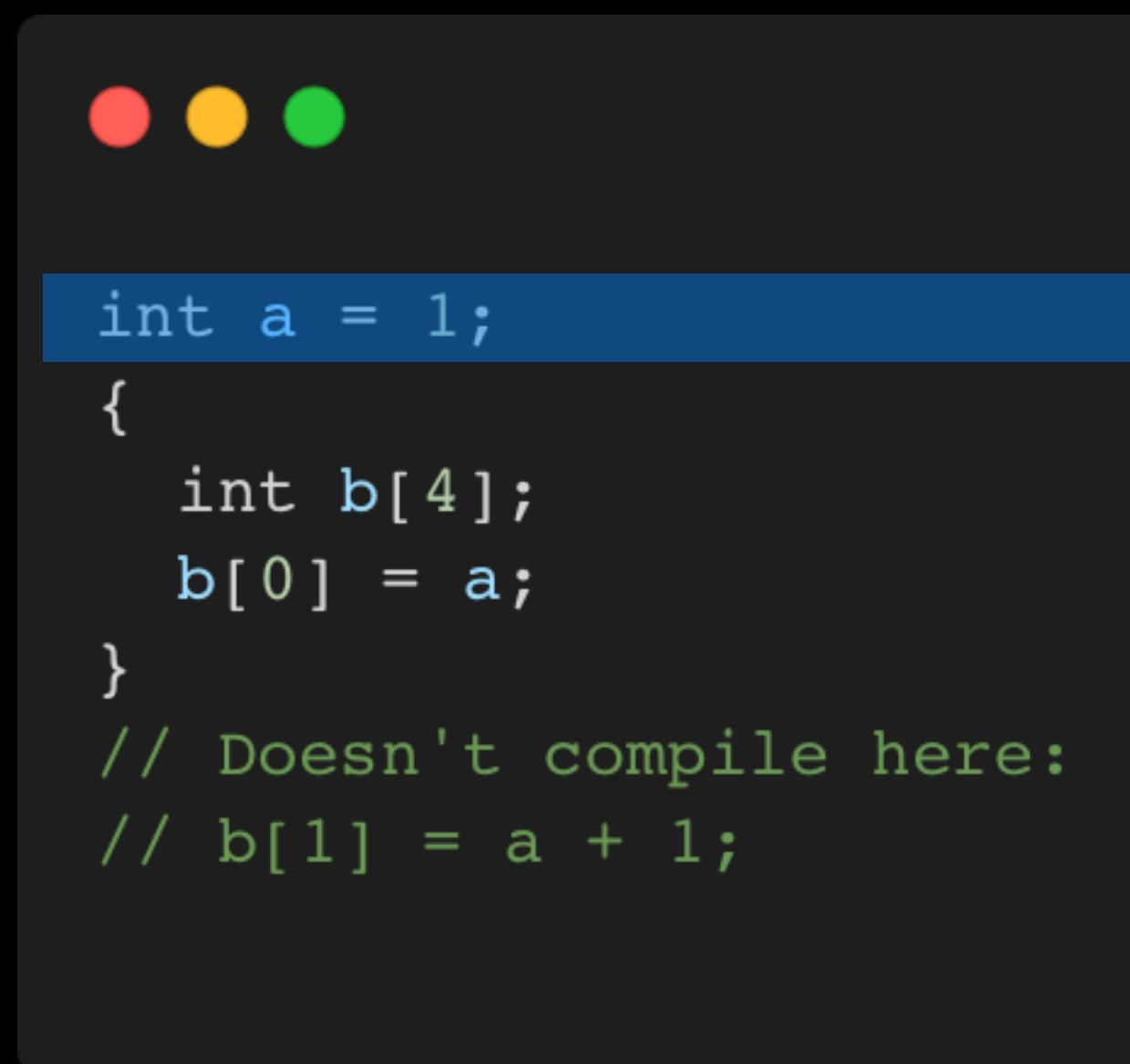
- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope



```
int a = 1;
{
    int b[4];
    b[0] = a;
}
// Doesn't compile here:
// b[1] = a + 1;
```

Scope and lifetime of variables

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope



```
● ● ●  
int a = 1;  
{  
    int b[4];  
    b[0] = a;  
}  
// Doesn't compile here:  
// b[1] = a + 1;
```

Memory layout

	0x3010
	0x300C
	0x3008
	0x3004
i = 4	0x3000

Scope and lifetime of variables

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope



```
int a = 1;
{
    int b[4];
    b[0] = a;
}
// Doesn't compile here:
// b[1] = a + 1;
```

Memory layout

b[3] = ?	0x3010
b[2] = ?	0x300C
b[1] = ?	0x3008
b[0] = ?	0x3004
i = 4	0x3000

Scope and lifetime of variables

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope



```
int a = 1;
{
    int b[4];
    b[0] = a;
}
// Doesn't compile here:
// b[1] = a + 1;
```

Memory layout

b[3] = ?	0x3010
b[2] = ?	0x300C
b[1] = ?	0x3008
b[0] = 1	0x3004
i = 4	0x3000

Scope and lifetime of variables

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope



```
int a = 1;
{
    int b[4];
    b[0] = a;
}
// Doesn't compile here:
// b[1] = a + 1;
```

Memory layout

?	0x3010
?	0x300C
?	0x3008
1	0x3004
i = 4	0x3000

Namepaces

- Namespaces allow to segment your code to avoid name clashes
- They can be embedded to create hierarchies (separator is `::`)



```
int value = 0;
namespace n {
    int value = 0;
}

namespace p {
    int value = 0;
    namespace inner {
        int value = 0;
    }
}

void f() {
    ::value = 42;
    n::value = 84;
    n::inner::value = 21;
}
```

Nested namespaces

- Easier way to declare nested namespaces

C++98

```
● ● ●  
namespace a {  
    namespace b {  
        namespace c {  
            // ...  
        }  
    }  
}
```

C++17

```
● ● ●  
namespace a::b::c {  
    // ...  
}
```

namespace alias

- The **namespace** keyword can be used to create an alias on an other namespace.



```
namespace very_long_namespace {
    int value = 0;
}

void f() {
    very_long_namespace::value = 42;
}
```

namespace alias

- The **namespace** keyword can be used to create an alias on an other namespace.



```
namespace very_long_namespace {
    int value = 0;
}

void f() {
    namespace vln = very_long_namespace;
    vln::value = 42;
}
```

namespace alias

- The **namespace** keyword can be used to create an alias on an other namespace.
- Or on nested namespaces.



```
namespace a::b::c::d {  
    int value = 0;  
}  
  
void f() {  
    a::b::c::d::value = 42;  
}
```

namespace alias

- The **namespace** keyword can be used to create an alias on an other namespace.
- Or on nested namespaces.



```
namespace a::b::c::d {  
    int value = 0;  
}  
  
void f() {  
    namespace l = a::b::c::d;  
    l::value = 42;  
}
```

Using namespace directive

- The `using namespace` directive make all members of the specified namespace visible in current scope.



```
namespace a {  
    int value = 0;  
}  
  
void f() {  
    a::value = 42;  
}
```

Using namespace directive

- The **using namespace** directive make all members of the specified namespace visible in current scope.



```
namespace a {  
    int value = 0;  
}  
  
void f() {  
    using namespace a;  
    value = 42;  
}
```

Using namespace directive

- The **using namespace** directive make all members of the specified namespace visible in current scope.
- The same for nested namespaces.



```
namespace a::b::c {  
    int value = 0;  
}  
  
void f() {  
    using namespace a::b::c;  
    value = 42;  
}
```

Using namespace directive

- The **using namespace** directive make all members of the specified namespace visible in current scope.
- The same for nested namespaces.
- As well as for any part of the nested namespaces.



```
namespace a::b::c {  
    int value = 0;  
}  
  
void f() {  
    using namespace a::b;  
    c::value = 42;  
}
```

Anonymous namespace

- groups a number of declarations
- visible only in the current translation unit
- but not reusable outside
- allows much better compiler optimizations and checking
 - e.g. unused function warning
 - context dependent optimizations

```
● ● ●  
namespace {  
    int locale_variable = 0;  
}
```

equivalent

```
● ● ●  
static int locale_variable = 0;
```

Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Struct

“members” grouped together under one name

```
● ● ●

struct individual {
    unsigned char age;
    float weight;
};

individual student;
student.age = 25;
student.weight = 78.5f;

individual teacher = {45, 67.0f};

individual* ptr = &student;
ptr->age = 24; // same as: (*ptr).age = 24;
```



Struct

“members” grouped together under one name



```
struct individual {  
    unsigned char age;  
    float weight;  
};  
  
individual student;  
student.age = 25;  
student.weight = 78.5f;  
  
individual teacher = {45, 67.0f};  
  
individual* ptr = &student;  
ptr->age = 24; // same as: (*ptr).age = 24;
```

Memory layout

0x3010
0x300C
0x3008
0x3004
0x3000



Struct

“members” grouped together under one name



```
struct individual {  
    unsigned char age;  
    float weight;  
};  
  
individual student;  
student.age = 25;  
student.weight = 78.5f;  
  
individual teacher = {45, 67.0f};  
  
individual* ptr = &student;  
ptr->age = 24; // same as: (*ptr).age = 24;
```

Memory layout

				0x3010
				0x300C
				0x3008
	?	?	?	0x3004
Student	?	?	?	0x3000



Struct

“members” grouped together under one name

```
● ● ●  
struct individual {  
    unsigned char age;  
    float weight;  
};  
  
individual student;  
student.age = 25;  
student.weight = 78.5f;  
  
individual teacher = {45, 67.0f};  
  
individual* ptr = &student;  
ptr->age = 24; // same as: (*ptr).age = 24;
```

Memory layout

				0x3010
				0x300C
				0x3008
	?	?	?	0x3004
25	?	?	?	0x3000

Student {

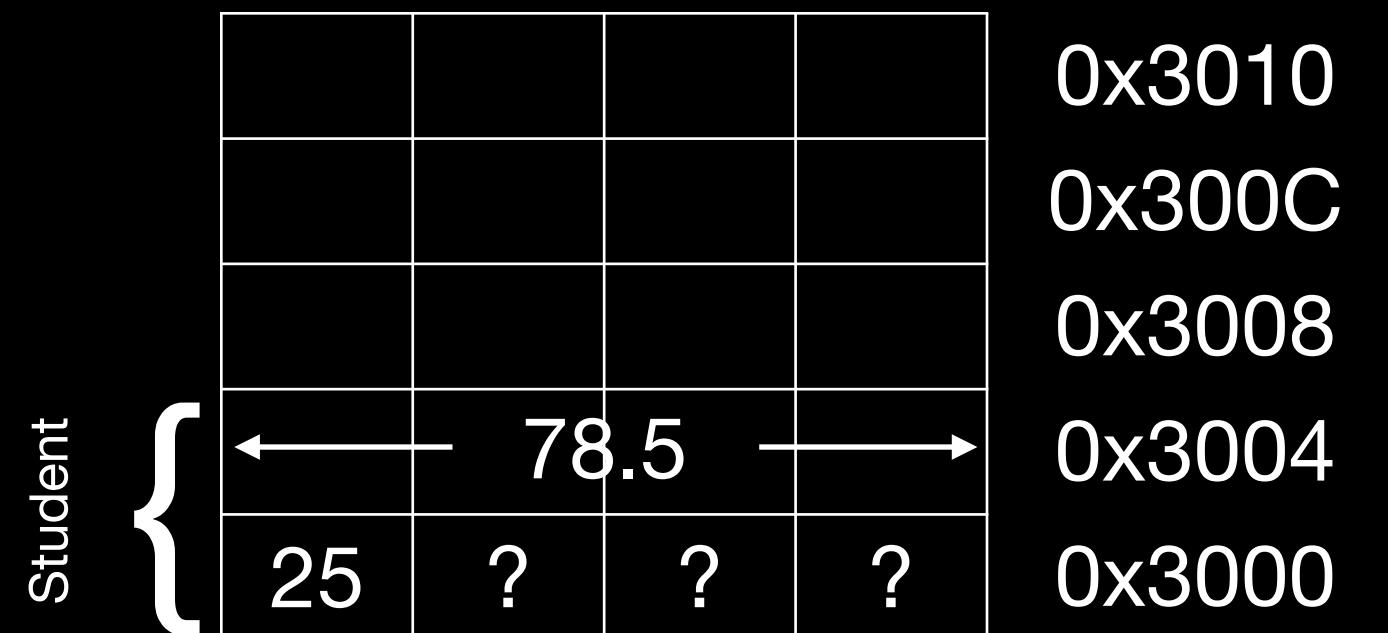


Struct

“members” grouped together under one name

```
● ● ●  
struct individual {  
    unsigned char age;  
    float weight;  
};  
  
individual student;  
student.age = 25;  
student.weight = 78.5f;  
  
individual teacher = {45, 67.0f};  
  
individual* ptr = &student;  
ptr->age = 24; // same as: (*ptr).age = 24;
```

Memory layout



Struct

“members” grouped together under one name

```
● ● ●  
struct individual {  
    unsigned char age;  
    float weight;  
};  
  
individual student;  
student.age = 25;  
student.weight = 78.5f;  
  
individual teacher = {45, 67.0f};
```

```
individual* ptr = &student;  
ptr->age = 24; // same as: (*ptr).age = 24;
```

Memory layout

Teacher	0x3010
67.0	0x300C
45	?
?	?
?	?
Student	0x3008
78.5	0x3004
25	?
?	?
?	?

Struct

“members” grouped together under one name

```
● ● ●  
struct individual {  
    unsigned char age;  
    float weight;  
};  
  
individual student;  
student.age = 25;  
student.weight = 78.5f;  
  
individual teacher = {45, 67.0f};  
  
individual* ptr = &student;  
ptr->age = 24; // same as: (*ptr).age = 24;
```

Memory layout

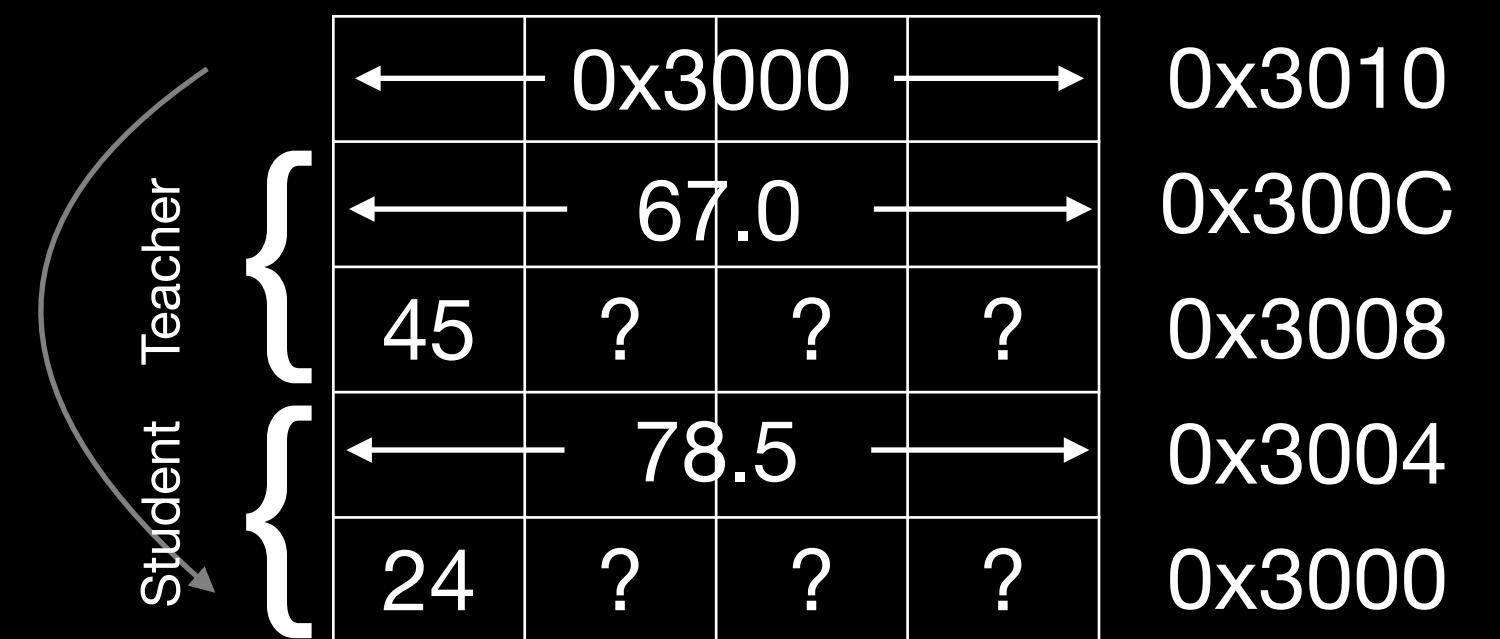
0x3000	0x3010			
67.0	0x300C			
45	?	?	?	0x3008
78.5	0x3004			
25	?	?	?	0x3000

Struct

“members” grouped together under one name

```
struct individual {  
    unsigned char age;  
    float weight;  
};  
  
individual student;  
student.age = 25;  
student.weight = 78.5f;  
  
individual teacher = {45, 67.0f};  
  
individual* ptr = &student;  
ptr->age = 24; // same as: (*ptr).age = 24;
```

Memory layout



Union

“members” packed together at same memory location

```
● ● ●

union duration {
    int seconds;
    short hours;
    char days;
};

duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3; // d1.seconds overwritten
int a = d1.seconds; // d1.seconds is garbage
```

Union

“members” packed together at same memory location

```
● ● ●  
union duration {  
    int seconds;  
    short hours;  
    char days;  
};  
  
duration d1, d2, d3;  
d1.seconds = 259200;  
d2.hours = 72;  
d3.days = 3;  
d1.days = 3; // d1.seconds overwritten  
int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

				0x300C
				0x3008
				0x3004
				0x3000

Union

“members” packed together at same memory location

```
● ● ●  
union duration {  
    int seconds;  
    short hours;  
    char days;  
};  
  
duration d1, d2, d3;  
d1.seconds = 259200;  
d2.hours = 72;  
d3.days = 3;  
d1.days = 3; // d1.seconds overwritten  
int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

				0x300C
?	?	?	?	0x3008
?	?	?	?	0x3004
?	?	?	?	0x3000

Union

“members” packed together at same memory location

```
● ● ●  
union duration {  
    int seconds;  
    short hours;  
    char days;  
};  
  
duration d1, d2, d3;  
d1.seconds = 259200;  
d2.hours = 72;  
d3.days = 3;  
d1.days = 3; // d1.seconds overwritten  
int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

				0x300C
?	?	?	?	0x3008
?	?	?	?	0x3004
?	?	?	?	0x3000
← 259200 →				

Union

“members” packed together at same memory location

```
● ● ●  
union duration {  
    int seconds;  
    short hours;  
    char days;  
};  
  
duration d1, d2, d3;  
d1.seconds = 259200;  
d2.hours = 72;  
d3.days = 3;  
d1.days = 3; // d1.seconds overwritten  
int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

?	?	?	?	0x300C
?	?	?	?	0x3008
← 72 →	?	?	?	0x3004
← 259200 →				0x3000

Union

“members” packed together at same memory location

```
● ● ●  
union duration {  
    int seconds;  
    short hours;  
    char days;  
};  
  
duration d1, d2, d3;  
d1.seconds = 259200;  
d2.hours = 72;  
d3.days = 3;  
d1.days = 3; // d1.seconds overwritten  
int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

				0x300C
3	?	?	?	0x3008
← 72 →	?	?	?	0x3004
← 259200 →				0x3000

Union

“members” packed together at same memory location

```
● ● ●  
union duration {  
    int seconds;  
    short hours;  
    char days;  
};  
  
duration d1, d2, d3;  
d1.seconds = 259200;  
d2.hours = 72;  
d3.days = 3;  
d1.days = 3; // d1.seconds overwritten  
int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

				0x300C
3	?	?	?	0x3008
← 72 →		?	?	0x3004
3	?	?	?	0x3000

Union

“members” packed together at same memory location

```
● ● ●  
union duration {  
    int seconds;  
    short hours;  
    char days;  
};  
  
duration d1, d2, d3;  
d1.seconds = 259200;  
d2.hours = 72;  
d3.days = 3;  
d1.days = 3; // d1.seconds overwritten  
int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

?	?	?	?	0x300C
3	?	?	?	0x3008
← 72 →	?	?	?	0x3004
3	?	?	?	0x3000

Enum

- use to declare a list of related constants (enumerators)
- has an underlying integral type
- enumerator names leak into enclosing scope



```
enum vehicle_type {  
    BIKE, // 0  
    CAR, // 1  
    BUS, // 2  
};  
  
vehicle_type t = CAR;
```



```
enum vehicle_type : int { // since C++11  
    BIKE = 3,  
    CAR = 5,  
    BUS = 7,  
};  
  
vehicle_type t = BUS;
```

Enum class

- scopes enumerator names, avoids name clashes
- strong typing, no automatic conversion to int



```
enum class vehicle_type {  
    bike, // 0  
    car, // 1  
    bus, // 2  
};  
  
vehicle_type t = vehicle_type::car;
```



```
enum class vehicle_type : int {  
    bike, // 0  
    car, // 1  
    bus, // 2  
};  
  
vehicle_type t = vehicle_type::car;
```

More concrete example



```
enum class shape_type {circle, rectangle};

struct rectangle {
    float width;
    float height;
};

struct shape {
    shape_type type;
    union {
        float radius;
        rectangle rect;
    };
};

shape circle1 {.type = shape_type::circle, .radius = 3.4};
shape rectangle1 {.type = shape_type::rectangle, .rect = {3, 4}};
```

typedef and using

C++98



```
typedef std::uint64_t myint;  
myint count = 17;  
typedef float position[3];
```

C++11



```
using myint = std::uint64_t;  
myint count = 17;  
using position = float[3];  
  
template<typename type_t>  
using myvec = std::vector<type_t>;  
myvec<int> myintvec;
```

Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



References

- References allow for direct access to another object
- They can be used as shortcuts / better readability
- They can be declared `const` to allow only read access



```
int i = 2;
int &iref = i; // access to i
iref = 3; // i is now 3

// const reference to a member:
struct a { int x; int y; } a;
const int &x = a.x; // direct read access to A's x
x = 4; // doesn't compile
a.x = 4; // fine
```

References vs pointers

- Natural syntax
- Cannot be null
- Must be assigned when defined, cannot be reassigned
- Prefer using references instead of pointers
- Mark references **const** to prevent modification

Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Functions



Functions



```
// without param and no return
void hello() {
    std::cout << "Hello, World";
}

// hello();
```

Functions



```
// without param and no return
void hello() {
    std::cout << "Hello, World";
}

// hello();
```



```
// without param and return
float pi() {
    return 3.14159;
}

// int result = pi();
```

Functions



```
// without param and no return
void hello() {
    std::cout << "Hello, World";
}

// hello();
```



```
// without param and return
float pi() {
    return 3.14159;
}

// int result = pi();
```



```
// with param and no return
void print(char* msg) {
    std::cout << msg;
}

// print("Hello, World");
```

Functions



```
// without param and no return
void hello() {
    std::cout << "Hello, World";
}

// hello();
```



```
// without param and return
float pi() {
    return 3.14159;
}

// int result = pi();
```



```
// with param and no return
void print(char* msg) {
    std::cout << msg;
}

// print("Hello, World");
```



```
// with param and return
int square(int a) {
    return a * a;
}

// int result = square(3);
```

Functions



```
// without param and no return  
void hello() {  
    std::cout << "Hello, World";  
}  
  
// hello();
```



```
// without param and return  
float pi() {  
    return 3.14159;  
}  
  
// int result = pi();
```



```
// with param and no return  
void print(char* msg) {  
    std::cout << msg;  
}  
  
// print("Hello, World");
```



```
// with param and return  
int square(int a) {  
    return a * a;  
}  
  
// int result = square(3);
```



```
// with params and return  
int multiply(int a, int b) {  
    return a * b;  
}  
  
// int result = multiply(3, 4);
```

Functions



```
// without param and no return
void hello() {
    std::cout << "Hello, World";
}

// hello();
```



```
// without param and return
float pi() {
    return 3.14159;
}

// int result = pi();
```



```
// with param and no return
void print(char* msg) {
    std::cout << msg;
}

// print("Hello, World");
```



```
// with param and return
int square(int a) {
    return a * a;
}

// int result = square(3);
```



```
// with params and return
int multiply(int a, int b) {
    return a * b;
}

// int result = multiply(3, 4);
```



```
// default argument
int add(int a, int b = 2) {
    return a + b;
}

// int result = add(3);
// int result = add(3, 1);
```

Functions



```
// without param and no return
void hello() {
    std::cout << "Hello, World";
}

// hello();
```



```
// without param and return
float pi() {
    return 3.14159;
}

// int result = pi();
```



```
// with param and no return
void print(char* msg) {
    std::cout << msg;
}

// print("Hello, World");
```



```
// with param and return
int square(int a) {
    return a * a;
}

// int result = square(3);
```



```
// with params and return
int multiply(int a, int b) {
    return a * b;
}

// int result = multiply(3, 4);
```



```
// default arguments
int add(int a = 4, int b = 2) {
    return a + b;
}

// int result = add();
// int result = add(3);
// int result = add(3, 1);
```

Functions



```
// without param and no return
void hello() {
    std::cout << "Hello, World";
}

// hello();
```



```
// without param and return
float pi() {
    return 3.14159;
}

// int result = pi();
```



```
// with param and no return
void print(char* msg) {
    std::cout << msg;
}

// print("Hello, World");
```



```
// with param and return
int square(int a) {
    return a * a;
}

// int result = square(3);
// int& r = square(2); // error
// int const& r = square(2);
```



```
// with params and return
int multiply(int a, int b) {
    return a * b;
}

// int result = multiply(3, 4);
```



```
// default arguments
int add(int a = 4, int b = 2) {
    return a + b;
}

// int result = add();
// int result = add(3);
// int result = add(3, 1);
```

Parameter are passed by value

```
● ● ●

struct big_struct {
    ...
};

big_struct s;

// parameter by value
void print_val(big_struct p) {
    ...
}
print_val(s); // copy

// parameter by reference
void print_ref(big_struct &q) {
    ...
}
print_ref(s); // no copy
```



Parameter are passed by value



```
struct big_struct {  
    ...  
};  
  
big_struct s;  
  
// parameter by value  
void print_val(big_struct p) {  
    ...  
}  
print_val(s); // copy  
  
// parameter by reference  
void print_ref(big_struct &q) {  
    ...  
}  
print_ref(s); // no copy
```

Memory layout

	0x31E0
	0x3190
	0x3140
	0x30F0
	0x30A0
	0x3050
	0x3000

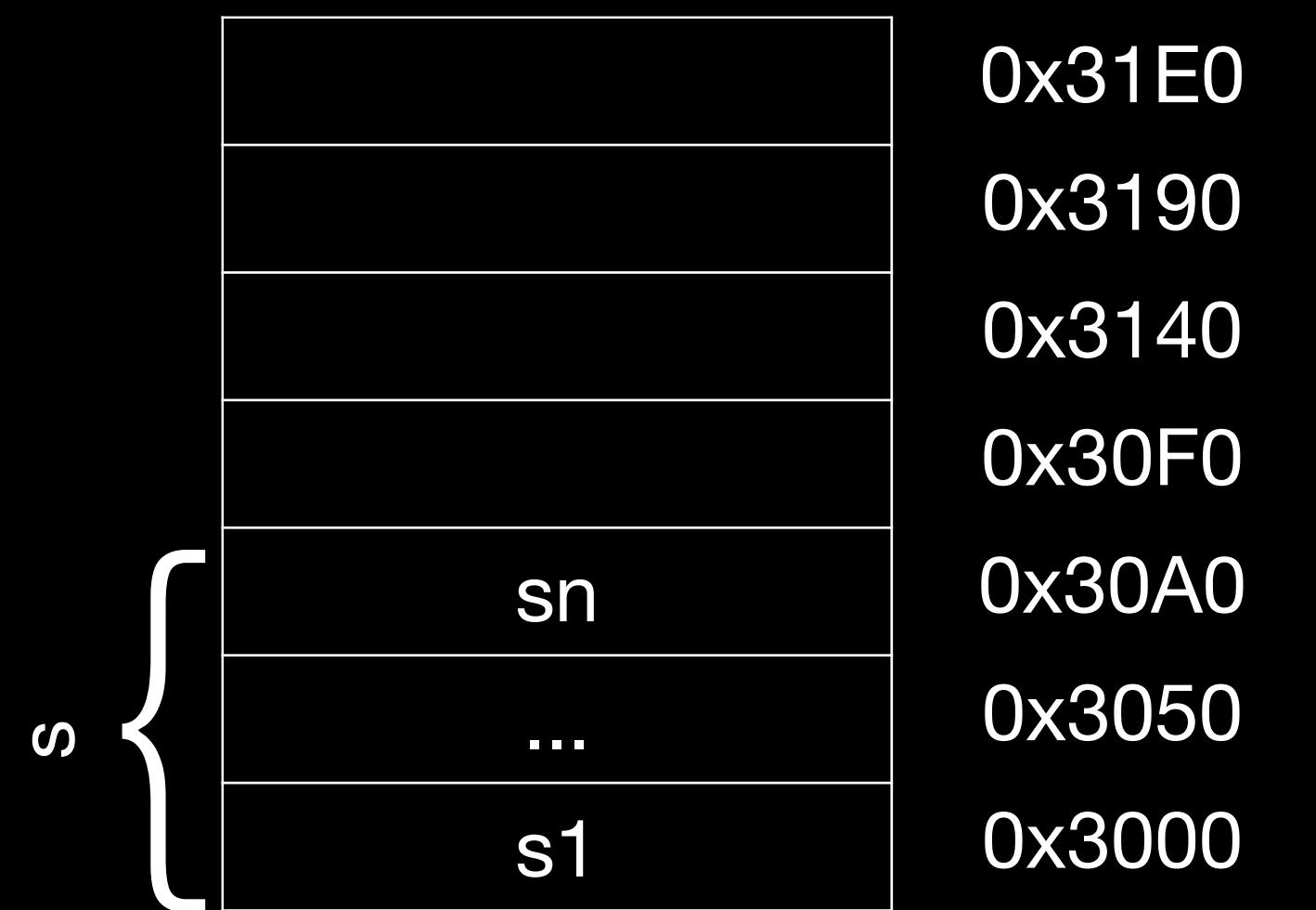


Parameter are passed by value



```
struct big_struct {  
    ...  
};  
  
big_struct s;  
  
// parameter by value  
void print_val(big_struct p) {  
    ...  
}  
print_val(s); // copy  
  
// parameter by reference  
void print_ref(big_struct &q) {  
    ...  
}  
print_ref(s); // no copy
```

Memory layout



Parameter are passed by value



```
struct big_struct {  
    ...  
};  
  
big_struct s;  
  
// parameter by value  
void print_val(big_struct p) {  
    ...  
}  
print_val(s); // copy  
  
// parameter by reference  
void print_ref(big_struct &q) {  
    ...  
}  
print_ref(s); // no copy
```

Memory layout

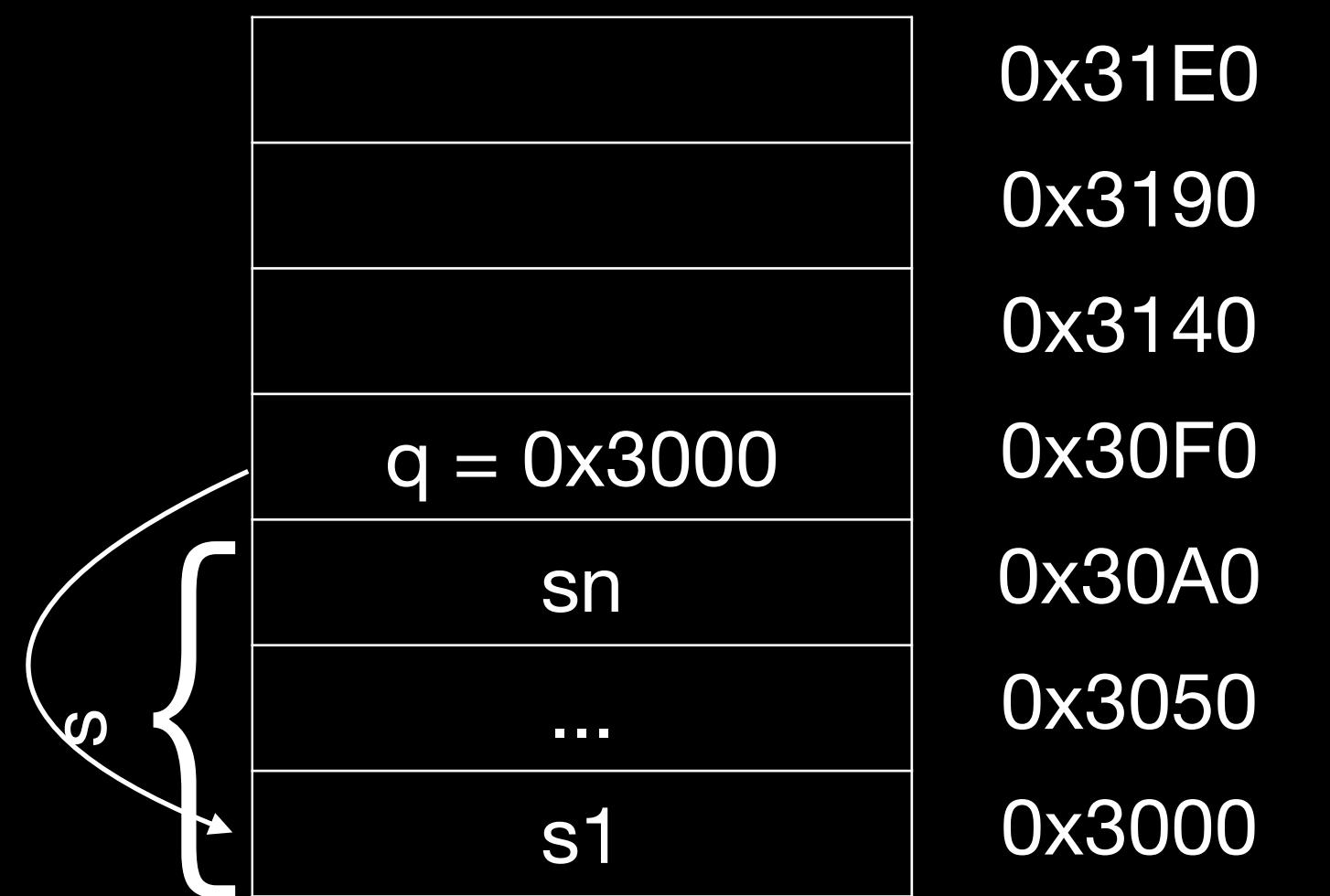
p	{	0x31E0
	pn	0x3190
	...	0x3140
	p1	0x30F0
s	{	0x30A0
	sn	0x3050
	...	0x3050
	s1	0x3000

Parameter are passed by value



```
struct big_struct {  
    ...  
};  
  
big_struct s;  
  
// parameter by value  
void print_val(big_struct p) {  
    ...  
}  
print_val(s); // copy  
  
// parameter by reference  
void print_ref(big_struct &q) {  
    ...  
}  
print_ref(s); // no copy
```

Memory layout



Pass by value or reference ?



```
struct small_struct {int a;};
small_struct s = {1};

void change_val(small_struct p) {
    p.a = 2;
}
change_val(s);
// s.a == 1

void change_ref(small_struct &q) {
    q.a = 2;
}
change_ref(s);
// s.a == 2
```



Pass by value or reference ?

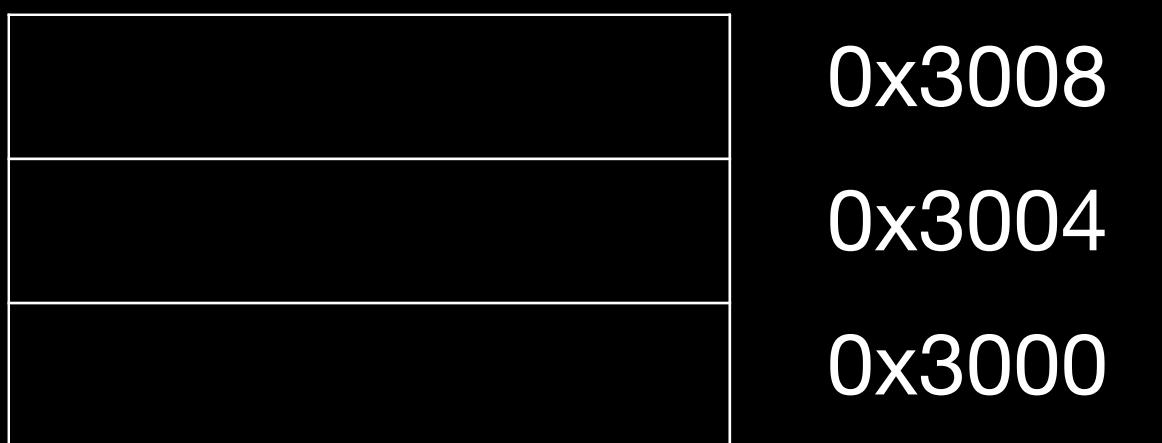


```
struct small_struct {int a;};
small_struct s = {1};

void change_val(small_struct p) {
    p.a = 2;
}
change_val(s);
// s.a == 1

void change_ref(small_struct &q) {
    q.a = 2;
}
change_ref(s);
// s.a == 2
```

Memory layout



Pass by value or reference ?

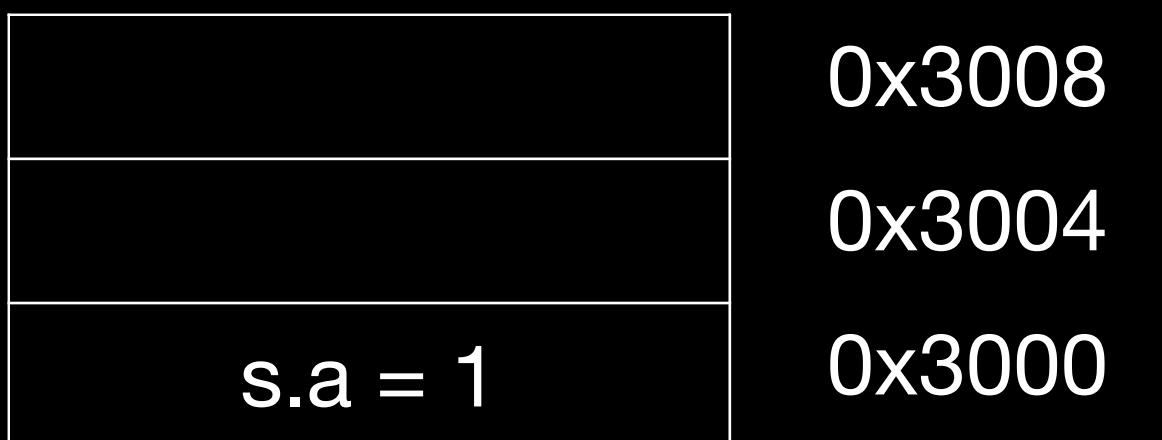


```
struct small_struct {int a;};
small_struct s = {1};

void change_val(small_struct p) {
    p.a = 2;
}
change_val(s);
// s.a == 1

void change_ref(small_struct &q) {
    q.a = 2;
}
change_ref(s);
// s.a == 2
```

Memory layout



Pass by value or reference ?



```
struct small_struct {int a;};
small_struct s = {1};

void change_val(small_struct p) {
    p.a = 2;
}
change_val(s);
// s.a == 1

void change_ref(small_struct &q) {
    q.a = 2;
}
change_ref(s);
// s.a == 2
```

Memory layout

	0x3008
p.a = 1	0x3004
s.a = 1	0x3000



Pass by value or reference ?



```
struct small_struct {int a;};
small_struct s = {1};

void change_val(small_struct p) {
    p.a = 2;
}
change_val(s);
// s.a == 1

void change_ref(small_struct &q) {
    q.a = 2;
}
change_ref(s);
// s.a == 2
```

Memory layout

	0x3008
p.a = 2	0x3004
s.a = 1	0x3000



Pass by value or reference ?

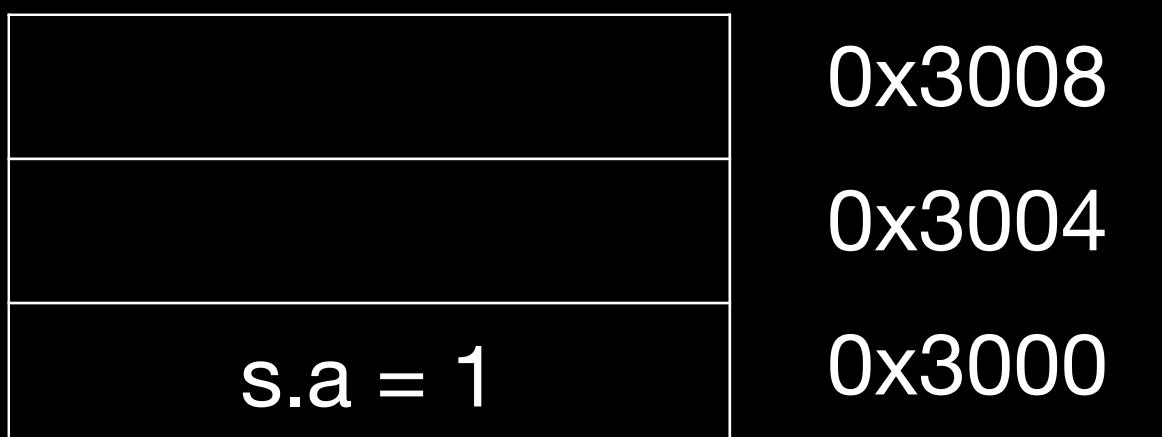


```
struct small_struct {int a;};
small_struct s = {1};

void change_val(small_struct p) {
    p.a = 2;
}
change_val(s);
// s.a == 1
```

```
void change_ref(small_struct &q) {
    q.a = 2;
}
change_ref(s);
// s.a == 2
```

Memory layout



Pass by value or reference ?

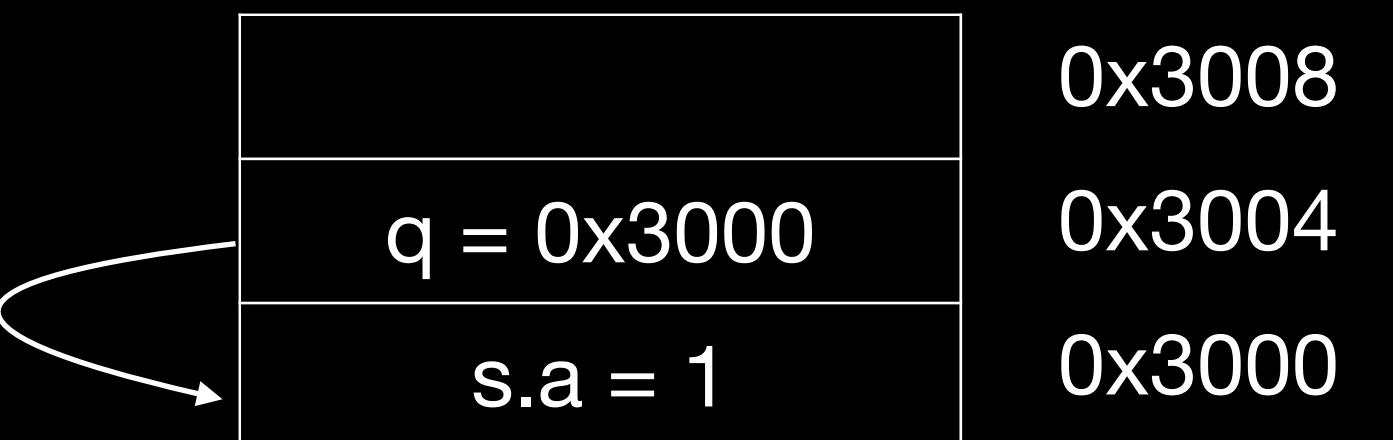


```
struct small_struct {int a;};
small_struct s = {1};

void change_val(small_struct p) {
    p.a = 2;
}
change_val(s);
// s.a == 1
```

```
void change_ref(small_struct &q) {
    q.a = 2;
}
change_ref(s);
// s.a == 2
```

Memory layout



Pass by value or reference ?

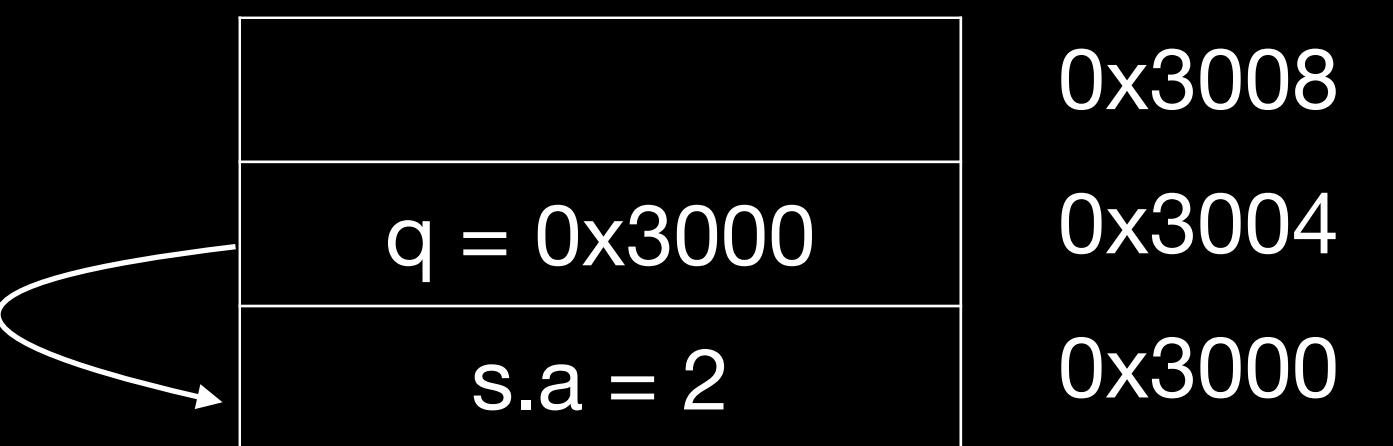


```
struct small_struct {int a;};
small_struct s = {1};

void change_val(small_struct p) {
    p.a = 2;
}
change_val(s);
// s.a == 1

void change_ref(small_struct &q) {
    q.a = 2;
}
change_ref(s);
// s.a == 2
```

Memory layout



Pass by value or reference ?

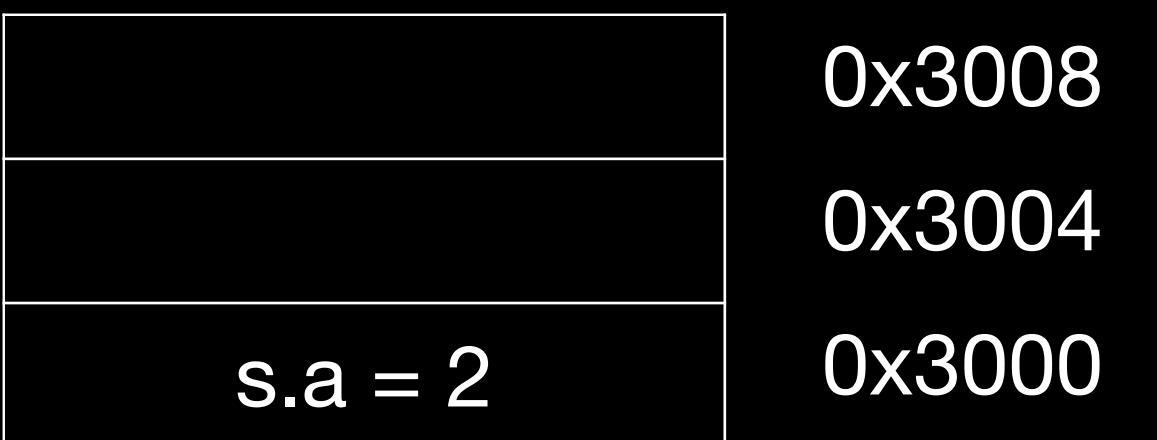


```
struct small_struct {int a;};
small_struct s = {1};

void change_val(small_struct p) {
    p.a = 2;
}
change_val(s);
// s.a == 1

void change_ref(small_struct &q) {
    q.a = 2;
}
change_ref(s);
// s.a == 2
```

Memory layout



Different ways to pass arguments to a function

- By default, arguments are passed by value (= copy) => good for small types, e.g. numbers
- Use references for parameters to avoid copies => good for large types, e.g. objects
- Use const for safety and readability whenever possible



```
struct foo {  
    ...  
};  
foo bar;  
  
void fct_val(foo value);  
fct_val(bar); // by value  
  
void fct_ref(const foo& value);  
fct_ref(bar); // by reference  
  
void fct_ptr(const foo* value);  
fct_ptr(&bar); // by pointer  
  
void fct_write(T &value);  
fct_write(bar); // non-const ref
```

Overloading

- We can have multiple functions with the same name
 - Must have different parameter lists
 - A different return type alone is not allowed
 - Form a so-called “overload set”
- Default arguments can cause ambiguities



```
int sum(int b); // 1
int sum(int b, int c); // 2, ok, overload
// float sum(int b, int c); // disallowed

sum(42); // calls 1
sum(42, 43); // calls 2

int sum(int b, int c, int d = 4); // 3, overload

sum(42, 43, 44); // calls 3
sum(42, 43); // error: ambiguous, 2 or 3
```

Exercise : functions

Familiarise yourself with pass by value / pass by reference.

- Go to exercises/functions
- Look at functions.cpp
- Compile it (make) and run the program (./functions)
- Work on the tasks that you find in functions.cpp



Good practices

- Write readable functions
- Keep functions short
- Do one logical thing (single-responsibility principle)
- Use expressive names
- Document non-trivial functions



```
/// @brief Count number of dilepton events in data.  
/// @param d Dataset to search.  
/// @return The number of dilepton events.  
unsigned int count_dileptons(data &d) {  
    select_events_with_muons(d);  
    select_events_with_electrons(d);  
    return d.size();  
}
```

Good practices

Don't! Everything in one long function

```
unsigned int run_job() {
    // Step 1: data
    data d;
    d.resize(123456);
    d.fill(...);

    // Step 2: muons
    for (...) {
        if (...) {
            d.erase(...);
        }
    }

    // Step 3: electrons
    for (...) {
        if (...) {
            d.erase(...);
        }
    }

    // Step 4: dileptons
    int counter = 0;
    for (...) {
        if (...) {
            counter++;
        }
    }

    return counter;
}
```



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Binary and Assignment Operators



```
int i = 1 + 4 - 2; // 3
i *= 3;           // short for: i = i * 3;
i /= 2;           // 4
i = 23 % i;      // modulo => 3
```

Increment / Decrement Operators



```
int i = 0; i++; // i = 1
int j = ++i;    // i = 2, j = 2
int k = i++;    // i = 3, k = 2
int l = --i;    // i = 2, l = 2
int m = i--;    // i = 1, m = 2
```

Bitwise and Assignment Operators



```
unsigned i = 0xee & 0x55; // 0x44
i |= 0xee;                // 0xee
i ^= 0x55;                // 0xbb
unsigned j = ~0xee;        // 0xffffffff11
unsigned k = 0x1f << 3;    // 0xf8
unsigned l = 0x1f >> 2;    // 0x7
```

Logical Operators



```
bool a = true;
bool b = false;
bool c = a && b; // false
bool d = a || b; // true
bool e = !d;     // false
```

Comparison Operators



```
bool a = (3 == 3); // true
bool b = (3 != 3); // false
bool c = (4 < 4); // false
bool d = (4 <= 4); // true
bool e = (4 > 4); // false
bool f = (4 >= 4); // true
auto g = (5 <=> 5); // Since C++20
```

Operator Precedence

Avoid writing a line of operators as in the example on the right

- It's better to decompose the expression
- and to use parentheses

```
c &= 1+(++b) | (a--) * 4 % 5 ^ 7; // ???
```

Details can be found on [cppreference](#)

Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



if - syntax

- The else and else if clauses are optional
- The else if clause can be repeated
- Braces are optional if there is a single statement



```
if (condition1) {  
    statement1; statement2;  
} else if (condition2)  
    only_one_statement;  
else {  
    statement3;  
    statement4;  
}
```

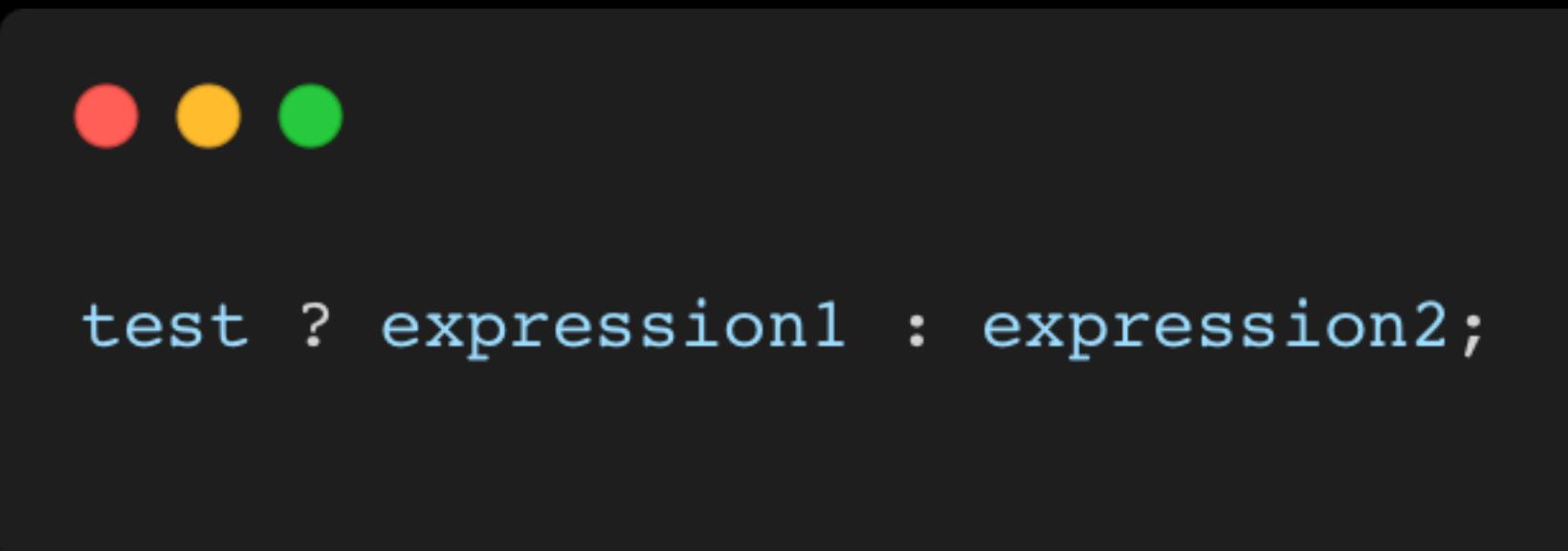
if - example

```
int collatz(int a) {
    if (a <= 0) {
        std::cout << "not supported\n";
        return 0;
    } else if (a == 1) {
        return 1;
    } else if (a%2 == 0) {
        return collatz(a/2);
    } else {
        return collatz(3*a+1);
    }
}
```



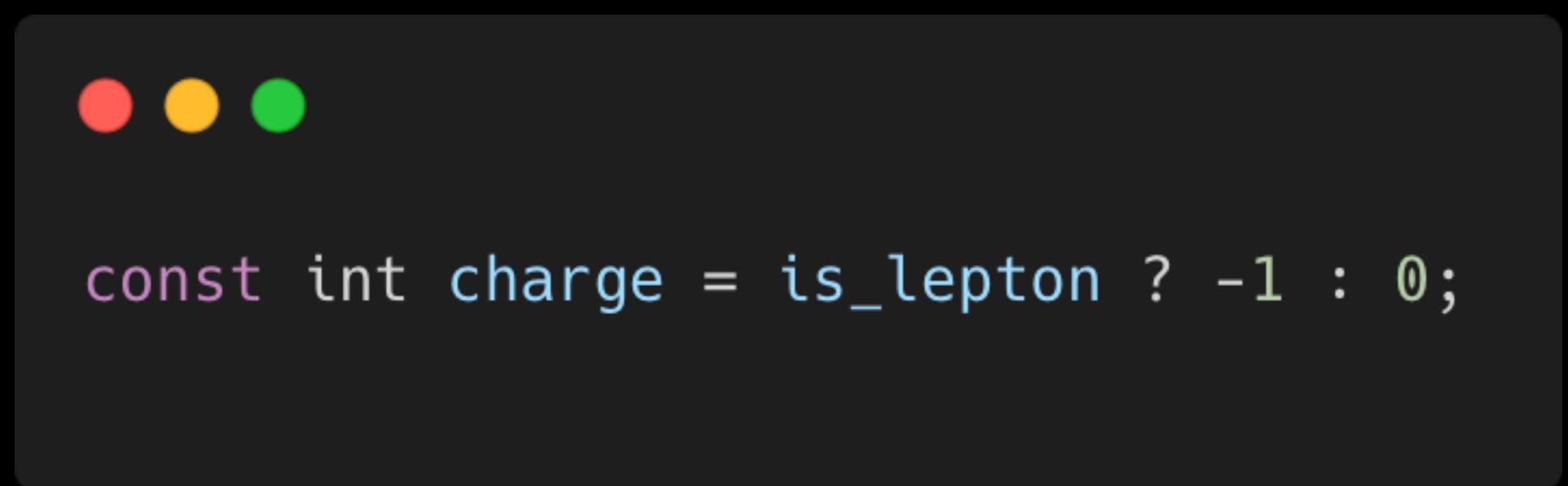
Conditional operator - syntax

- If **test** is true **expression1** is returned
- Else, **expression2** is returned



```
test ? expression1 : expression2;
```

Conditional operator - example



A screenshot of a dark-themed macOS terminal window. In the top-left corner, there are three colored window control buttons: red, yellow, and green. The main text area contains a single line of C++ code:

```
const int charge = is_lepton ? -1 : 0;
```

Conditional operator - best practice

Avoid writing a line of conditional operators as in the example on the right

- Explicit `ifs` are generally easier to read
- Use the ternary operator with short conditions and expressions
- Avoid nesting



```
int collatz(int a) {  
    return a==1 ? 1 : collatz(a%2==0 ? a/2 : 3*a+1);  
}
```

switch - syntax

- The **break** statement is not mandatory but...
- Cases are entry points, not independent pieces
- Execution “falls through” to the next case without a **break**!
- The **default** case may be omitted
- Avoid **switch** statements with fall-through cases



```
switch(identifier) {  
    case c1 : statements1; break;  
    case c2 : statements2; break;  
    case c3 : statements3; break;  
    ...  
    default : statementsn; break;  
}
```

switch - example



```
enum class lang {french, german, english, other};  
lang language = ...;  
switch (language) {  
    case lang::french:  
        std::cout << "Bonjour";  
        break;  
    case lang::german:  
        std::cout << "Guten Tag";  
        break;  
    case lang::english:  
        std::cout << "Good morning";  
        break;  
    default:  
        std::cout << "I do not speak your language";  
}
```

[[fallthrough]] attribute

- New compiler warning
- Since C++ 17, compilers are encouraged to warn on fall-through



```
switch (c) {  
    case 'a':  
        f(); // Warning emitted  
    case 'b': // Warning probably suppressed  
    case 'c':  
        g();  
        [[fallthrough]]; // Warning suppressed  
    case 'd':  
        h();  
}
```

Init-statements for if and switch

Purpose

- Allows to limit variable scope in if and switch statements



```
if (value val = get_value(); condition(val)) {  
    f(val); // ok  
} else  
    g(val); // ok  
h(val); // error, no `val` in scope here
```

Init-statements for if and switch

Don't confuse with a variable declaration as condition!



```
if (value* val = get_value_ptr( ))  
    f(*val);
```

for loop - syntax

- Multiple initializations / increments are comma separated
- Initializations can contain declarations
- Braces are optional if loop body is a single statement



```
for (initializations; condition; increments) {  
    statements;  
}
```

for loop - example



```
for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {  
    std::cout << i << "^2 is " << j << '\n';  
}
```

Range-based loop - syntax

- Simplifies loops over “ranges” tremendously
- Especially with std containers and ranges



```
for (type iteration_variable : range) {  
    // body using iteration_variable  
}
```

Range-based loop - example

```
● ● ●  
int v[ ] = {1,2,3,4};  
int sum = 0;  
for (int a : v) {  
    sum += a;  
}
```

Init-statements for range-based loop

Purpose

- Allows to limit variable scope in range-based loops



```
std::array data = {"hello", ",","world"};
for (std::size_t i = 0; auto& d : data) {
    std::cout << i++ << ' ' << d << '\n';
}
```

while loop - syntax

- Braces are optional if loop body is a single statement



```
while (condition) {  
    statements;  
}  
  
do {  
    statements;  
} while (condition);
```

while loop - example



```
int j = 2;
while (j < 9) {
    std::cout << j << ' ';
    j += 2;
}
std::cout << '\n';
```



```
int j = 2;
do {
    std::cout << j << ' ';
    j += 2;
}
while (j < 9);
std::cout << '\n';
```

Jump statements



Jump statements

- **break** Exits the loop and continues after it



```
stream_data sd = get_stream_data();
while (!sd.end_of_stream()) {
    data d;
    errno err = sd.read_data(d);
    if (errno != 0)
        break;
    consume_data(d);
}
```

Jump statements

- **break** Exits the loop and continues after it
- **continue** Goes immediately to next loop iteration



```
stream_data sd = get_stream_data();
while (!sd.end_of_stream()) {
    data d;
    errno err = sd.read_data(d);
    if (errno != 0)
        continue;
    consume_data(d);
}
```

Jump statements

- **break** Exits the loop and continues after it
- **continue** Goes immediately to next loop iteration
- **return** Exits the current function



```
void println(const char* value) {  
    if (value == nullptr) return;  
  
    std::cout << value << std::endl;  
}
```

Jump statements

- **break** Exits the loop and continues after it
- **continue** Goes immediately to next loop iteration
- **return** Exits the current function



```
bool is_odd(int value) {  
    if (value % 2) return true;  
    return false;  
}
```

Jump statements

- **break** Exits the loop and continues after it
- **continue** Goes immediately to next loop iteration
- **return** Exits the current function
- **goto** Can jump anywhere inside a function, **avoid!**



```
void println(const char* value) {  
    if (value == nullptr) goto end;  
  
    std::cout << value;  
  
end:  
    std::cout << std::endl;  
}
```

Exercise : control

Familiarise yourself with different kinds of control structures.

Re-implement them in different ways.

- Go to exercises/control
- Look at control.cpp
- Compile it (make) and run the program (./control)
- Work on the tasks that you find in README.md



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- **auto keyword**
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- **auto keyword**
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- **inline keyword**
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- **inline keyword**
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- **inline keyword**
- Assertions



Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions



Outline

1. Introduction
2. Language Basics
3. Object Oriented Programming (OOP)
4. Core Modern C++
5. Modern C++ Expert
6. Advanced Programming



End

