Modern C++ Course

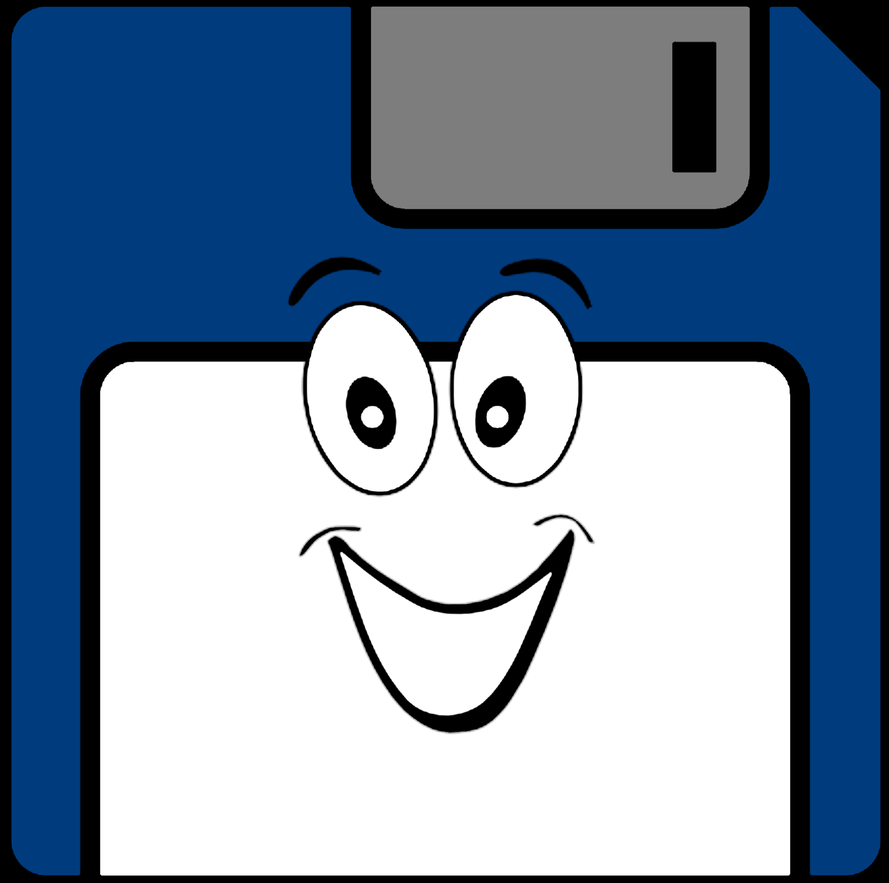
Who am I ?

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
- Struct 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
- Struct 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

```cpp
#include <iostream>

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

The first classic application

program.cpp

```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

```cpp
#include <iostream>

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

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.20)

project(hello_world)
add_executable(${PROJECT_NAME} program.cpp)
```

Output

```
> Hello, World!
```

The first classic application

program.cpp

```cpp
#include <print>

auto main() -> int {
  std::println("Hello, World!");
}
```

CMakeLists.txt

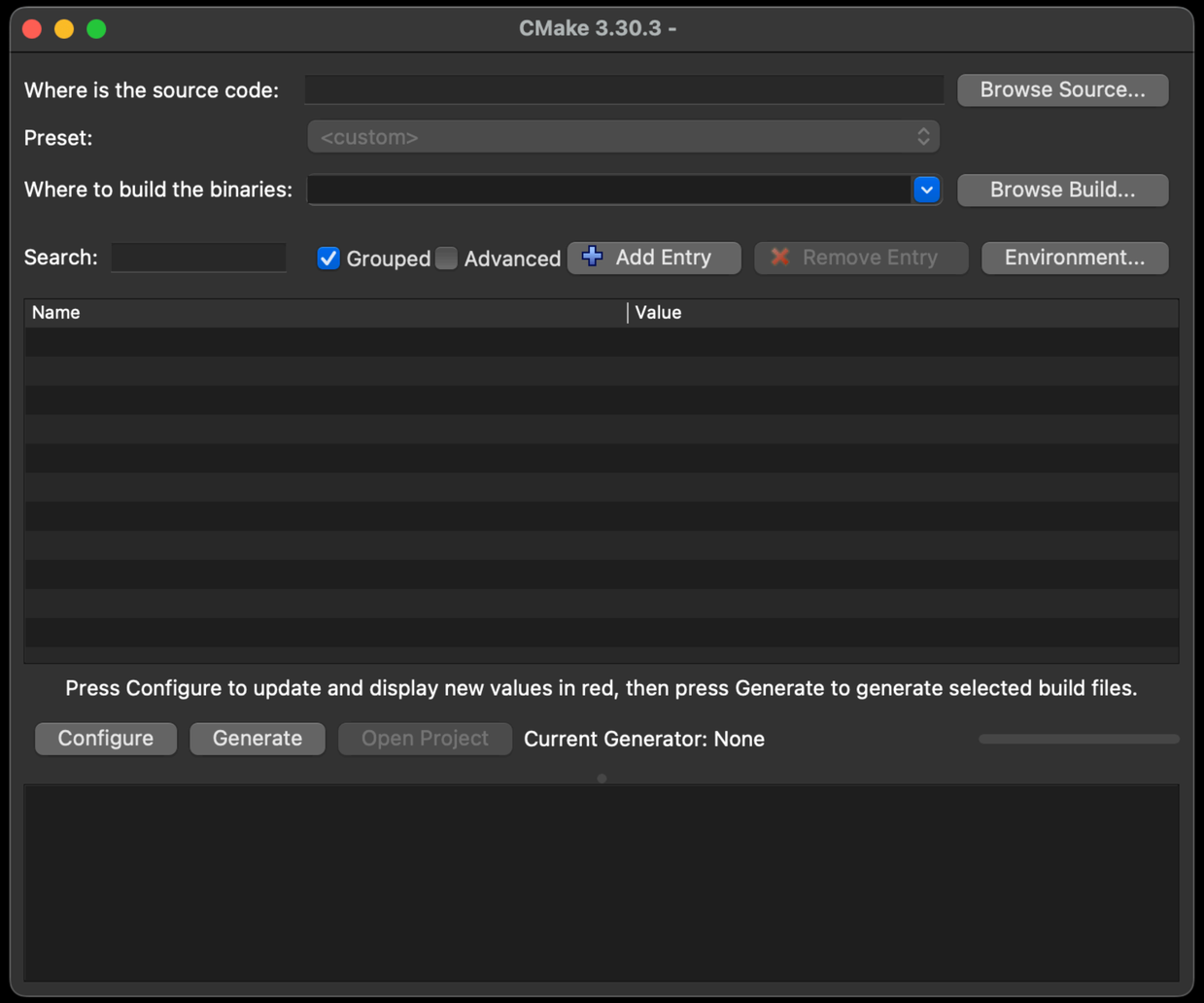
```cmake
cmake_minimum_required(VERSION 3.20)

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

```cpp
#include <iostream>

int main() {
  std::cout << "main function" << std::endl;
}
```

Main function

```cpp
#include <iostream>

int main() {
  std::cout << "main function" << std::endl;
}
```

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
  for (auto index = 0; index < argc; index++)
    std::cout << (index == 0 ? "main function with arguments [" : ", ") << argv[index];
  std::cout << "]" << 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
- Struct 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
- Struct and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions

Comments

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

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

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

Basic types

```cpp
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 ASCIl 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

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

long l = -111;               // min 32 bit signed integer
long int li = -121;          // int is optional
unsigned long ul = 13Ul;     // min 32 bit unsigned integer
unsigned long int uli = 14Ul; // int is optional

long long ll = -1511;               // min 64 bit signed integer
long long int lli = -161l;          // int is optional
unsigned long long ull = 17ull;     // min 64 bit unsigned integer
unsigned long long int ulli = 18ull; // int is optional
```

Basic types

```
1 float f = 0.19f;         // 32 (1+8+23) bit float
2 double d = 0.20;         // 64 (1+11+52) bit float
3 long double ld = 0.211; // min 64 bit float
4
5 const char* nstr = "native string"; // array of chars ended by \0
6 std::string str = "string";         // class provided by the std
```

Fixed width integer types

```cpp
#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 u132 = 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

```cpp
#include <stdfloat>

std:: float16_t value = 3.14f16;   // 16 (1+5+10) bit float
std:: float32_t value = 3.14732;   // like float (1+8+23)
                                   // but different type
std:: float64_t value = 3.14†64;   // like double (1+11+52)
                                   // but different type
std:: float128_t value = 3.14f128; // 128 (1+15+112) bit float
std: :bfloat16_t value = 3.14bf16; // 16 (1+8+7) bit float

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

Integer literals

```
 1 int value = 4284;                    // decimal (base 10)
 2 int value = 0b0001000010111100;      // binary (base 2) since C++14
 3 int value = 010274;                  // octal (base 8)
 4 int value = 0x10bc;                  // hexadecimal (base 16)
 5 int value = 0x10BC;                  // hexadecimal (base 16)
 6
 7 int value = 123'456'789;             // digit separators, since C++14
 8 int value = 0b0001'0000'1011'1100; // digit separators, since C++14
 9
10 4284             // int
11 4284u,   4284U   // unsigned int
12 42841,   4284L   // long
13 4284ul,  4284UL  // unsigned long
14 428411,  4284LL  // long long
15 4284ull, 4284ULL // unsigned long long
```

Floating-point literals

```
 1 double value = 12.34;
 2 double value = 12.;
 3 double value = .34;
 4 double value = 12e34;           // 12 * 10^34
 5 double value = 12E34;           // 12 * 10^34
 6 double value = 12e-34;          // 12 * 10^-34
 7 double value = 12.34e34;        // 12.34 * 10^34
 8
 9 double value = 123'456.789'101; // digit separators, C++14
10 double value = 0x4d2.4p3;       // hexfloat, 0x4d2.4 * 2^3
11                                 // = 1234.25 * 2^3 = 9874
12
13 3.14f, 3.14F, // float
14 3.14,  3.14,  // double
15 3.141, 3.14L, // long double
```

Sizeof

```cpp
#include <cstddef>

int value = 42;

std::size_t size_1 = sizeof(value); // 4
std::size_t size_2 = sizeof(int);   // 4
std::size_t size_3 = sizeof(42);    // 4
```

Pointer to integer

```cpp
#include <cstdint>

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
- Struct 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
- Struct and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions

Static arrays

```
 1 int ints1[4] = {1, 2, 3, 4};
 2 int ints2[] = {1, 2, 3, 4}; // identical
 3
 4 char chars1[3] = {'a', 'b', 'c'};    // char array
 5 char chars2[4] = "abc";              // valid native string
 6 char chars3[4] = {'a', 'b', 'c', 0}; // same valid native string
 7
 8 int i1 = ints1[2];  // i1 = 3
 9 char c = chars1[8]; // at best garbage, may segfault
10 int i2 = ints1[4];  // also garbage !
```

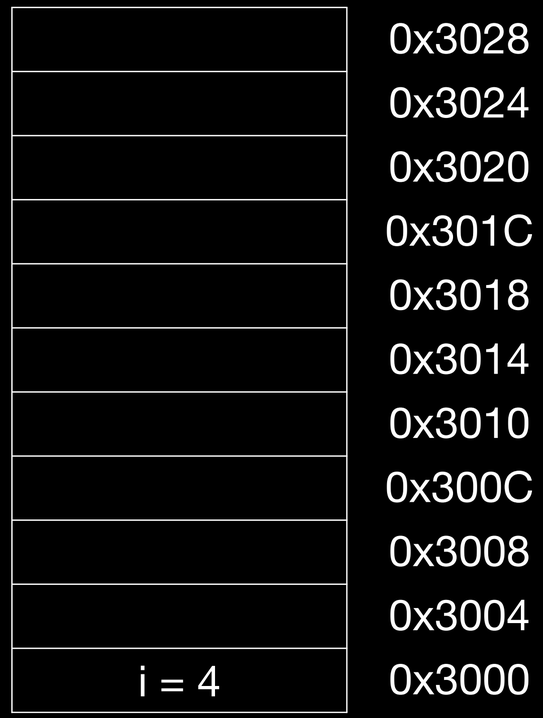
Pointers

```
 1 int i = 4;
 2 int* pi = &i;
 3 int j= *pi + 1;
 4
 5 int ai[] = {1, 2, 3};
 6 int* pai = ai; // decay to pointer
 7 int* paj = pai + 1;
 8 int k = *paj + 1;
 9
10 // compile error
11 int* pak = k;
12
13 // segmentation fault !
14 int* pak = (int*)k;
15 int l = *pak;
```

Pointers

```
1  int i = 4;
2  int* pi = &i;
3  int j= *pi + 1;
4
5  int ai[] = {1, 2, 3};
6  int* pai = ai; // decay to pointer
7  int* paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int* pak = k;
12
13 // segmentation fault !
14 int* pak = (int*)k;
15 int l = *pak;
```

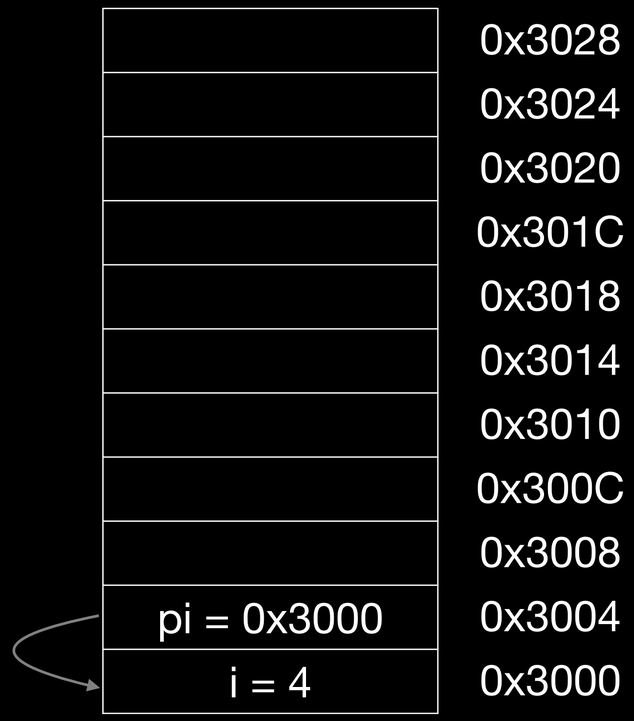
Memory layout



Pointers

```
1 int i = 4;
2 int* pi = &i;
3 int j= *pi + 1;
4
5 int ai[] = {1, 2, 3};
6 int* pai = ai; // decay to pointer
7 int* paj = pai + 1;
8 int k = *paj + 1;
9
10 // compile error
11 int* pak = k;
12
13 // segmentation fault !
14 int* pak = (int*)k;
15 int l = *pak;
```

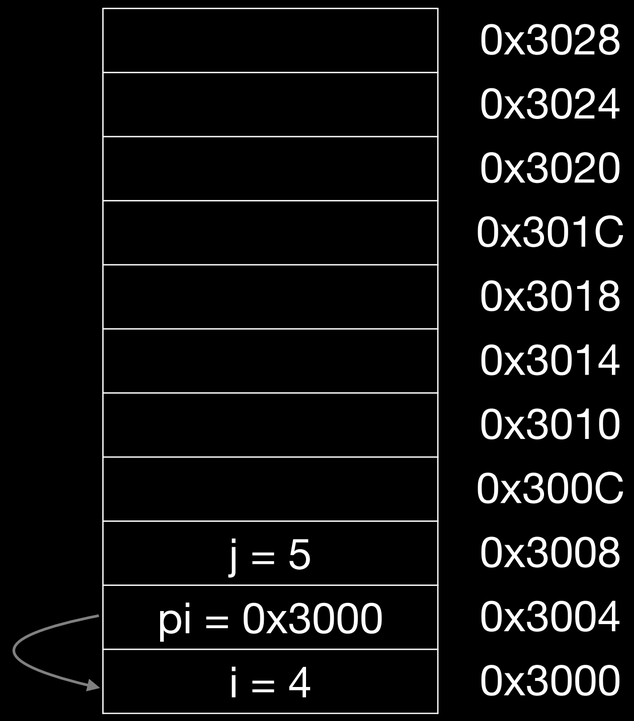
Memory layout



Pointers

```
1 int i = 4;
2 int* pi = &i;
3 int j= *pi + 1;
4
5 int ai[] = {1, 2, 3};
6 int* pai = ai; // decay to pointer
7 int* paj = pai + 1;
8 int k = *paj + 1;
9
10 // compile error
11 int* pak = k;
12
13 // segmentation fault !
14 int* pak = (int*)k;
15 int l = *pak;
```

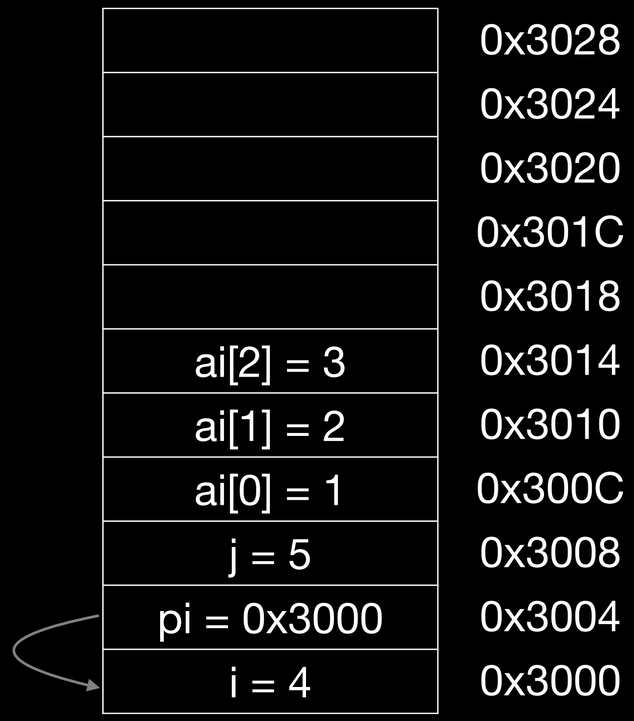
Memory layout



Pointers

```
1 int i = 4;
2 int* pi = &i;
3 int j= *pi + 1;
4
5 int ai[] = {1, 2, 3};
6 int* pai = ai; // decay to pointer
7 int* paj = pai + 1;
8 int k = *paj + 1;
9
10 // compile error
11 int* pak = k;
12
13 // segmentation fault !
14 int* pak = (int*)k;
15 int l = *pak;
```

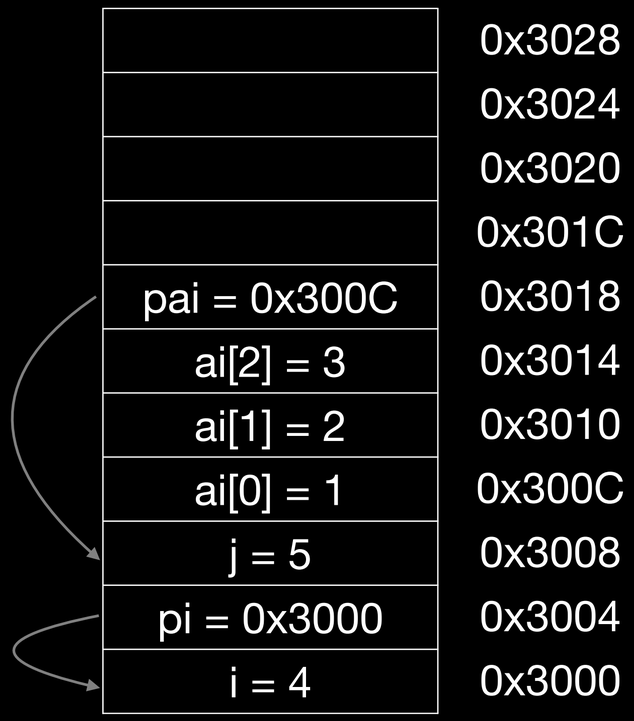
Memory layout



Pointers

```
1 int i = 4;
2 int* pi = &i;
3 int j= *pi + 1;
4
5 int ai[] = {1, 2, 3};
6 int* pai = ai; // decay to pointer
7 int* paj = pai + 1;
8 int k = *paj + 1;
9
10 // compile error
11 int* pak = k;
12
13 // segmentation fault !
14 int* pak = (int*)k;
15 int l = *pak;
```

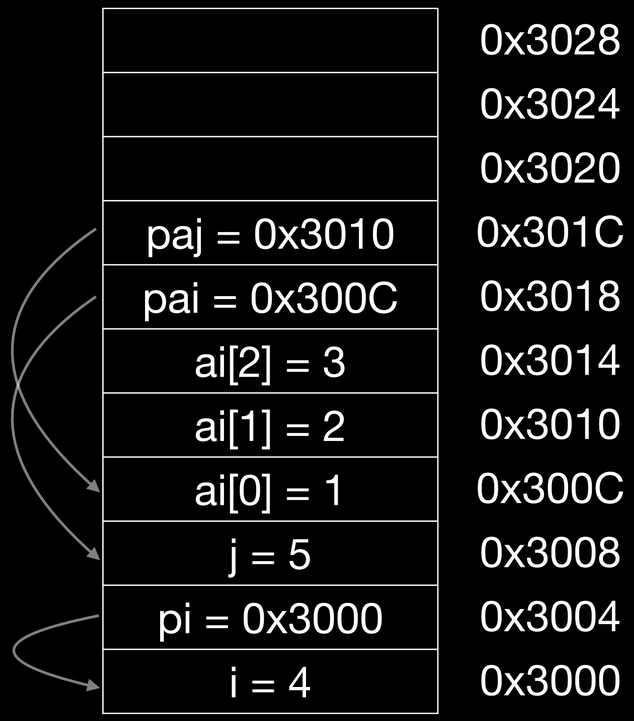
Memory layout



Pointers

```cpp
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;
```

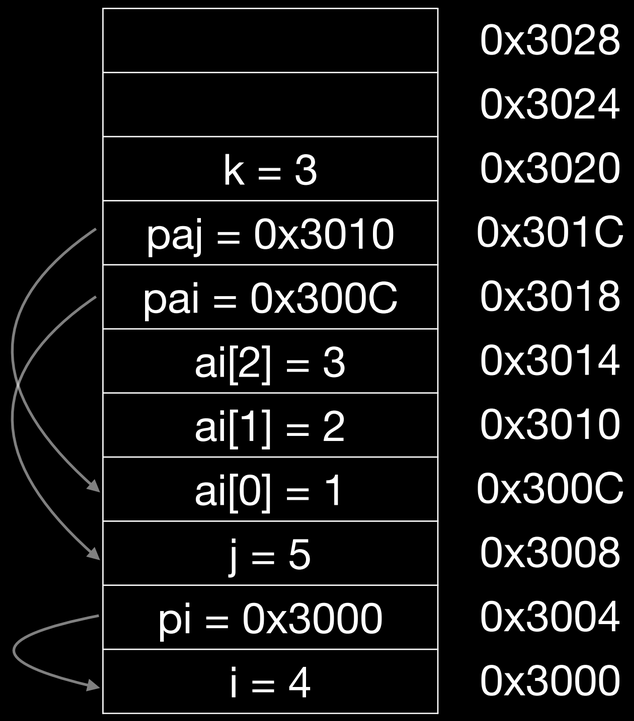
Memory layout



Pointers

```cpp
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;
```

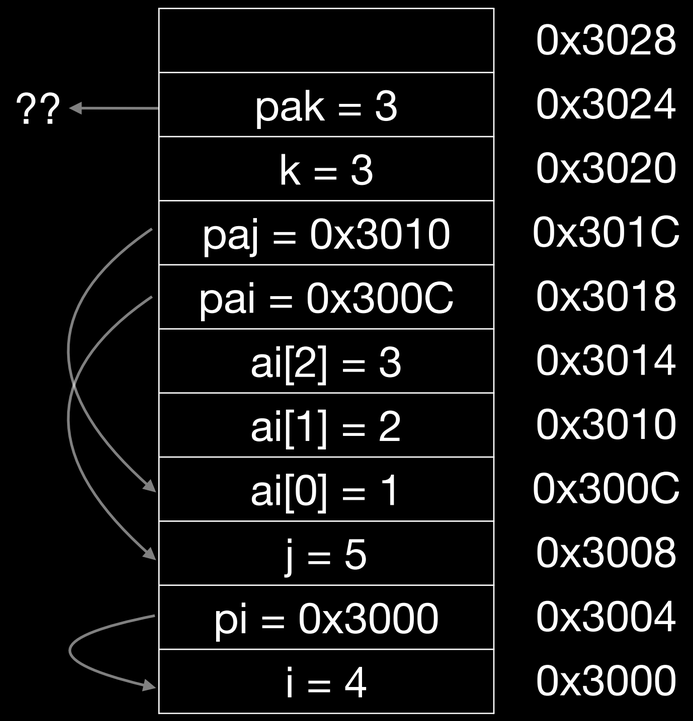
Memory layout



Pointers

```
1 int i = 4;
2 int* pi = &i;
3 int j= *pi + 1;
4
5 int ai[] = {1, 2, 3};
6 int* pai = ai; // decay to pointer
7 int* paj = pai + 1;
8 int k = *paj + 1;
9
10 // compile error
11 int* pak = k;
12
13 // segmentation fault !
14 int* pak = (int*)k;
15 int l = *pak;
```

Memory layout



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

```
1 int* ip = nullptr;
2 int i1 = NULL;    // compiles but bug
3 int i2 = nullptr; // ERROR
```

Dynamic arrays using C

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

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

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

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

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

Dynamic arrays using C++

```
// allocate array of ints
int* ai1 = new int[10];            // 10 ints, uninitialized
int* ai2 = new int[10] {};         // 10 ints, zero-initialized, since C++11
int* ai3 = new int[] {42, 21, 84}; // 3 ints, initialized, since C++11

// release arrays memory
delete[] ai1;
delete[] ai2;
delete[] ai3;

// allocate a single int
int* pi1 = new int;
int* pi2 = new int();    // zero-initialized
int* pi3 = new int {};   // zero-initialized, since C++11
int* pi4 = new int(42);  // initialized
int* pi5 = new int {42}; // initialized, since C++11

// release scalar memory
delete pi1;
delete pi2;
delete pi3;
delete pi4;
delete pi5;
```

Language Basics

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

Language Basics

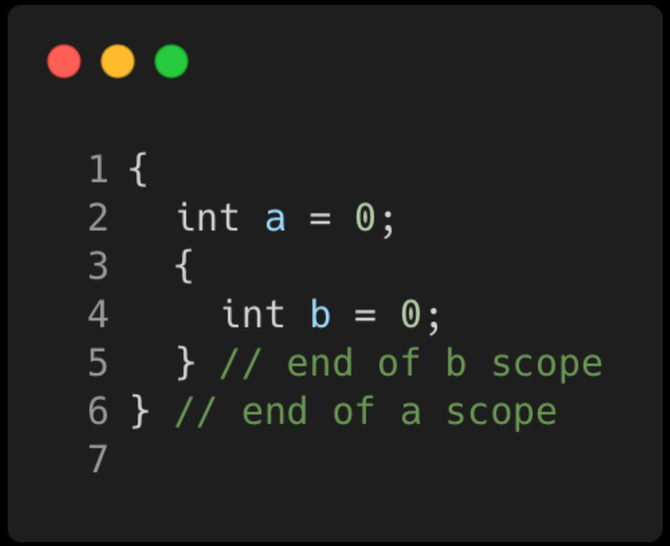
- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Struct and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions

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)



```
1 {
2   int a = 0;
3   {
4     int b = 0;
5   } // end of b scope
6 } // end of a scope
7
```

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

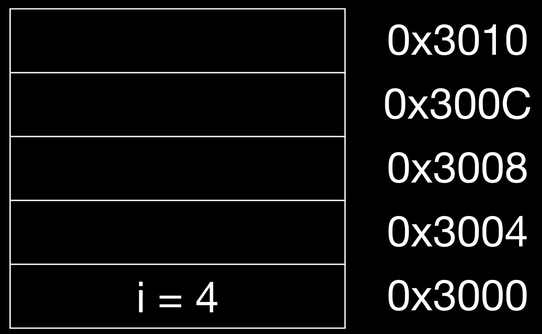
```
1 int a = 1;
2 {
3   int b[4];
4   b [0] = a;
5 }
6 // Doesn't compile here:
7 // b[1] = a + 1;
```

Scope and lifetime of variables

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

```
1 int a = 1;
2 {
3   int b[4];
4   b [0] = a;
5 }
6 // Doesn't compile here:
7 // b[1] = a + 1;
```

Memory layout

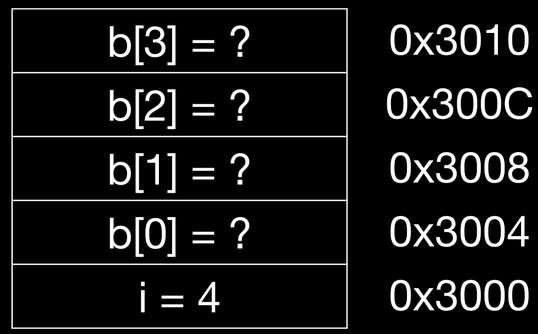


Scope and lifetime of variables

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

```
1 int a = 1;
2 {
3   int b[4];
4   b [0] = a;
5 }
6 // Doesn't compile here:
7 // b[1] = a + 1;
```

Memory layout

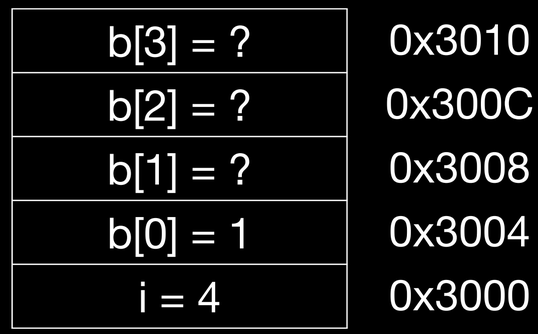


Scope and lifetime of variables

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

```
1 int a = 1;
2 {
3   int b[4];
4   b [0] = a;
5 }
6 // Doesn't compile here:
7 // b[1] = a + 1;
```

Memory layout

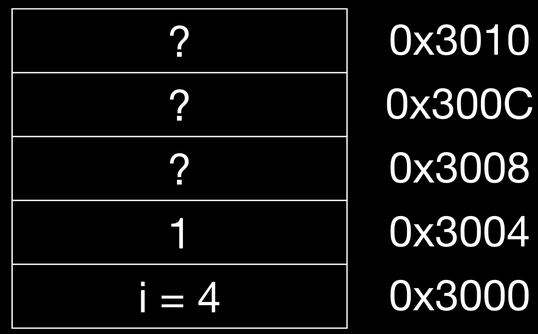


Scope and lifetime of variables

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

```
1 int a = 1;
2 {
3   int b[4];
4   b [0] = a;
5 }
6 // Doesn't compile here:
7 // b[1] = a + 1;
```

Memory layout



Namepaces

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

```
1 int value = 0;
2 namespace n {
3   int value = 0;
4   namespace p {
5     int value = 0;
6     namespace inner {
7       int value = 0;
8     }
9   }
10 }
11
12 void f() {
13   ::value = 42;
14   n::value = 84;
15   n::p::inner:: value = 21;
16 }
```

Nested namespaces

- Easier way to declare nested namespaces

C++98

```
1 namespace a {
2   namespace b {
3     namespace c {
4       // ...
5     }
6   }
7 }
```

C++17

```
1 namespace a::b::c {
2   // ...
3 }
```

namespace alias

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

```
1 namespace very_long_namespace {
2   int value = 0;
3 }
4
5 void f() {
6   very_long_namespace::value = 42;
7 }
```

namespace alias

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

```
1 namespace very_long_namespace {
2   int value = 0;
3 }
4
5 void f() {
6   namespace vln = very_long_namespace;
7   vln::value = 42;
8 }
```

namespace alias

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

```
1 namespace a::b::c::d {
2   int value = 0;
3 }
4
5 void f( ) {
6   a::b::c::d::value = 42;
7 }
```

namespace alias

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

```
1 namespace a::b::c::d {
2   int value = 0;
3 }
4
5 void f( ) {
6   namespace l = a::b::c::d;
7   l::value = 42;
8 }
```

Using namespace directive

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

```
1 namespace a {
2   int value = 0;
3 }
4
5 void f() {
6   a::value = 42;
7 }
```

Using namespace directive

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

```
1 namespace a {
2   int value = 0;
3 }
4
5 void f() {
6   using namespace a;
7   value = 42;
8 }
```

Using namespace directive

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

```
1 namespace a::b::c {
2   int value = 0;
3 }
4
5 void f() {
6   using namespace a::b::c;
7   value = 42;
8 }
```

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.

```
1 namespace a::b::c {
2   int value = 0;
3 }
4
5 void f() {
6   using namespace a::b;
7   c::value = 42;
8 }
```

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

```
1 namespace {
2   int local_variable = 0;
3 }
```

equivalent

```
1 static int local_variable = 0;
```

Language Basics

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

Struct

“members” grouped together under one name

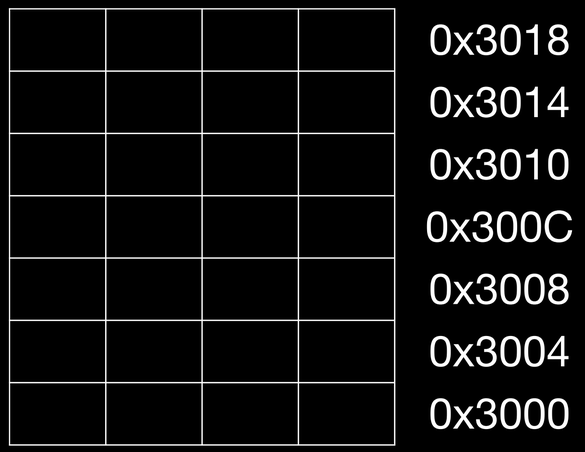
```
1 struct individual {
2   unsigned char age;
3   float weight;
4 };
5
6 individual student;
7 student.age = 25;
8 student.weight = 78.5f;
9
10 individual teacher = {45, 67.0f};
11
12 individual director = {.weight = 80.8f, .age = 60};
13
14 individual* ptr = &student;
15 ptr->age = 24; // same as: (*ptr).age = 24;
```

Struct

"members" grouped together under one name

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

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

individual teacher = {45, 67.0f};

individual director = {.weight = 80.8f, .age = 60};

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

Memory layout

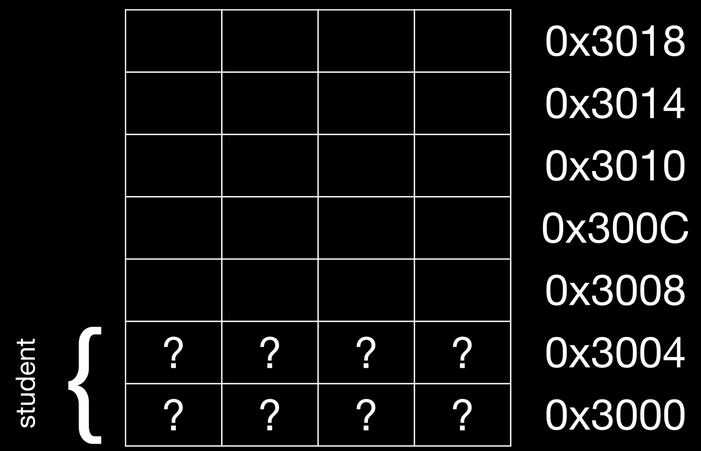


Struct

"members" grouped together under one name

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

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

individual teacher = {45, 67.0f};

individual director = {.weight = 80.8f, .age = 60};

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

Memory layout

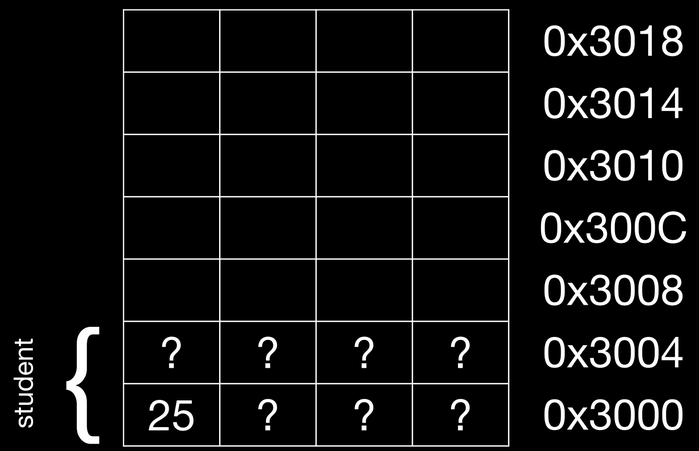


Struct

"members" grouped together under one name

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

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

individual teacher = {45, 67.0f};

individual director = {.weight = 80.8f, .age = 60};

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

Memory layout

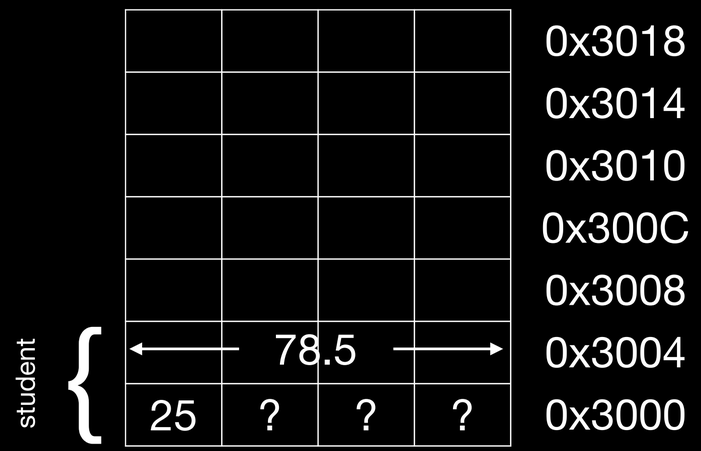


Struct

"members" grouped together under one name

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

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

individual teacher = {45, 67.0f};

individual director = {.weight = 80.8f, .age = 60};

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

Memory layout

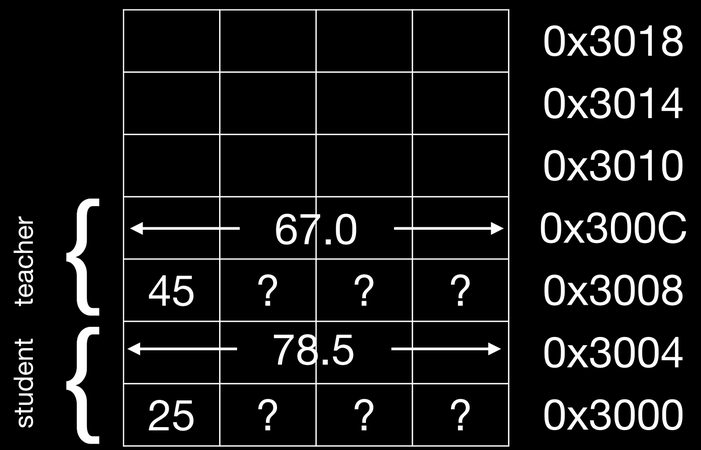


Struct

"members" grouped together under one name

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

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

individual teacher = {45, 67.0f};

individual director = {.weight = 80.8f, .age = 60};

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

Memory layout

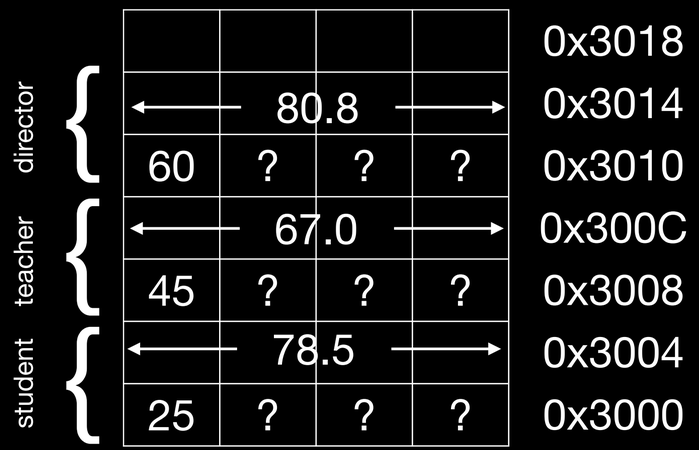


Struct

"members" grouped together under one name

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

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

individual teacher = {45, 67.0f};

individual director = {.weight = 80.8f, .age = 60};

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

Memory layout

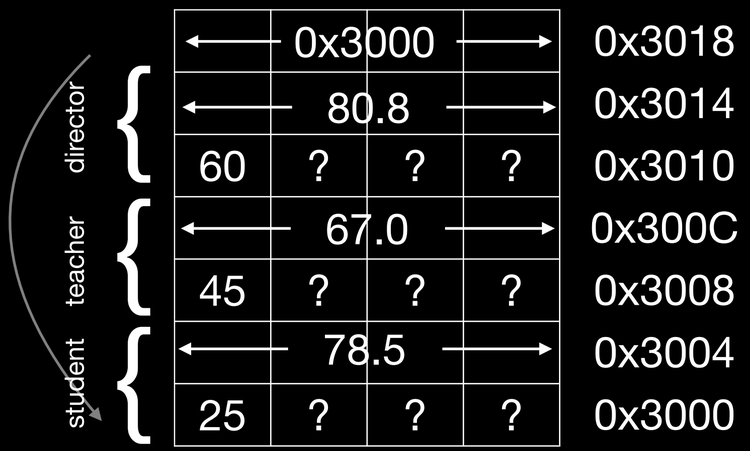


Struct

"members" grouped together under one name

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

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

individual teacher = {45, 67.0f};

individual director = {.weight = 80.8f, .age = 60};

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

Memory layout

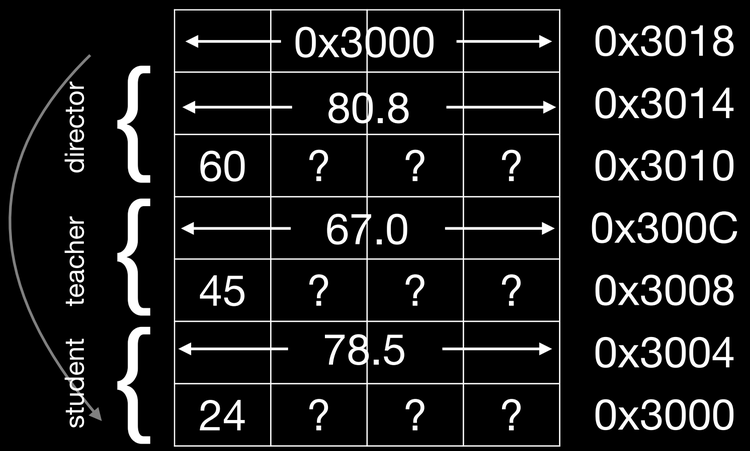


Struct

"members" grouped together under one name

```
 1 struct individual {
 2   unsigned char age;
 3   float weight;
 4 };
 5
 6 individual student;
 7 student.age = 25;
 8 student.weight = 78.5f;
 9
10 individual teacher = {45, 67.0f};
11
12 individual director = {.weight = 80.8f, .age = 60};
13
14 individual* ptr = &student;
15 ptr->age = 24; // same as: (*ptr).age = 24;
```

Memory layout



Union

“members” packed together at same memory location

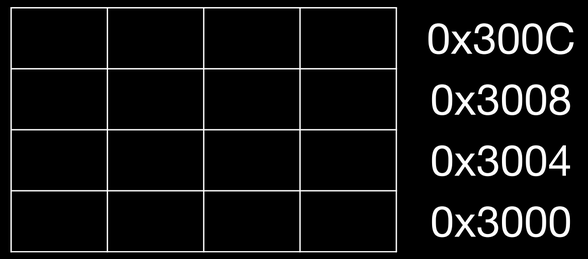
```
union duration {
  int seconds;
  short hours;
  char days;
};

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

```
 1 union duration {
 2   int seconds;
 3   short hours;
 4   char days;
 5 };
 6
 7 duration d1, d2, d3;
 8 d1.seconds = 259200;
 9 d2.hours = 72;
10 d3.days = 3;
11 d1. days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

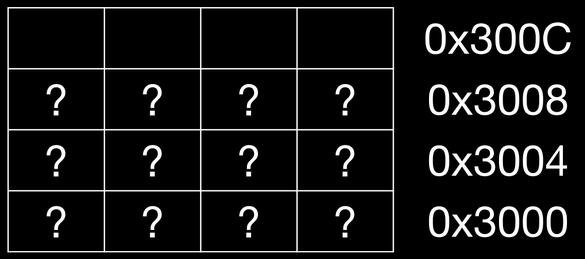


Union

"members" packed together at same memory location

```
 1 union duration {
 2   int seconds;
 3   short hours;
 4   char days;
 5 };
 6
 7 duration d1, d2, d3;
 8 d1.seconds = 259200;
 9 d2.hours = 72;
10 d3.days = 3;
11 d1. days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

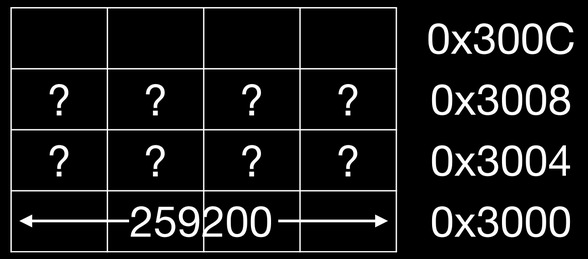


Union

"members" packed together at same memory location

```cpp
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

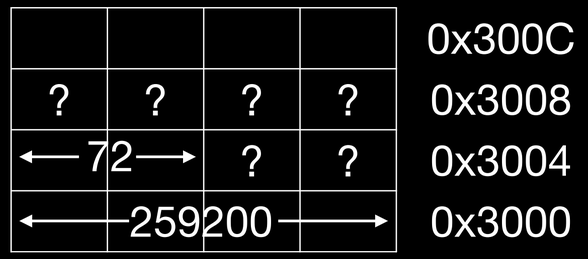


Union

"members" packed together at same memory location

```
 1 union duration {
 2   int seconds;
 3   short hours;
 4   char days;
 5 };
 6
 7 duration d1, d2, d3;
 8 d1.seconds = 259200;
 9 d2.hours = 72;
10 d3.days = 3;
11 d1. days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

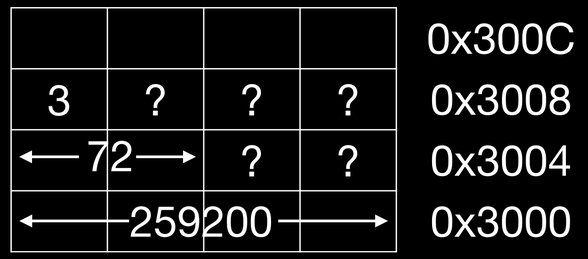


Union

"members" packed together at same memory location

```cpp
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

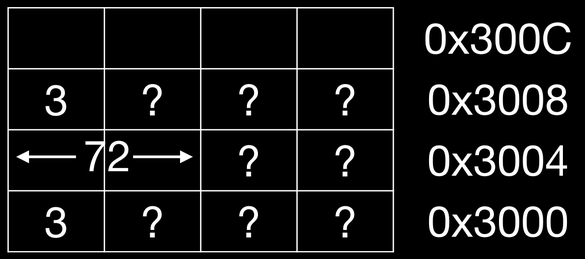


Union

"members" packed together at same memory location

```
 1 union duration {
 2   int seconds;
 3   short hours;
 4   char days;
 5 };
 6
 7 duration d1, d2, d3;
 8 d1.seconds = 259200;
 9 d2.hours = 72;
10 d3.days = 3;
11 d1. days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

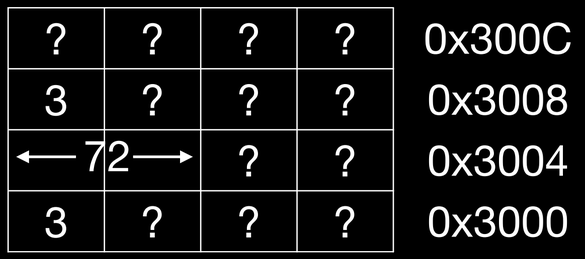


Union

"members" packed together at same memory location

```cpp
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



Enum

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

```
1 enum vehicle_type {
2   BIKE, // 0
3   CAR,  // 1
4   BUS,  // 2
5 };
6
7 vehicle_type t = CAR;
```

```
1 enum vehicle_type : int { // Since C++11
2   BIKE = 3,
3   CAR = 5,
4   BUS = 7,
5 };
6
7 vehicle_type t = BUS;
```

Enum class

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

```
1 enum class vehicle_type {
2   bike, // 0
3   car,  // 1
4   bus,  // 2
5 };
6
7 vehicle_type t = vehicle_type::car;
```

```
1 enum class vehicle_type : int {
2   bike = 3,
3   car = 5,
4   bus = 7,
5 };
6
7 vehicle_type t = vehicle_type::bus;
```

More concrete example

```
1 enum class shape_type {circle, rectangle};
2 struct rectangle {
3   float width;
4   float height;
5 };
6
7 struct shape {
8   shape_type type;
9   union {
10    float radius;
11    rectangle rect;
12  } ;
13 };
14
15 shape circle1 {.type = shape_type::circle, .radius = 3.45};
16 shape rectangle1 {.type = shape_type::rectangle, .rect = 13, 4};
```

typedef and using

C++98

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

C++11

```
1 using myint = std::uint64_t;
2 myint count = 17;
3 using position = float[3];
4
5 template<typename type_t>
6 using myvec = std::vector<type_t>;
7 myvec<int> myintvec;
```

Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Struct 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
- Struct 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

```
1 int i = 2;
2 int& iref = i; // access to i
3 iref = 3;      // i is now 3
4
5 // const reference to a member:
6 struct a_struct {int x; int y;} a;
7 const int& x = a.x; // direct read access to a_struct's x
8 x = 4;              // doesn't compile
9 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
- Struct 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
- Struct and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions

Function without param and no return

```cpp
#include <iostream>

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

int main() {
  hello();
}
```

Function without param and return

```cpp
#include <iostream>

// function without param and return
float get_pi() {
  return 3.14159;
}

int main() {
  float pi = get_pi();
  std::cout << "pi = " << pi << std::endl;
}
```

Function with param and no return

```cpp
#include <iostream>
#include <string>

// function with param and no return
void println(const std::string& msg) {
  std::cout << msg << std::endl;
}

int main() {
  println("Hello, World!");
}
```

Function with param and return

```cpp
#include <iostream>

// function with param and return
int square(int value) {
  return value * value;
}

int main() {
  int result = square(3);
  std::cout << "3^2 = " << result << std::endl;
}
```

Function with params and return

```cpp
#include <iostream>

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

int main() {
  int result = multiply(3, 4);
  std::cout << "3 * 4 = " << result << std::endl;
}
```

Function with default argument

```cpp
#include <iostream>

// function with default argument
int add(int a, int b = 1) {
  return a + b;
}

int main() {
  int result = add(3);
  std::cout << "result = " << result << std::endl;

  result = add(3, 4);
  std::cout << "result = " << result << std::endl;
}
```

Function with default arguments

```cpp
#include <iostream>

// function with default arguments
int add(int a = 4, int b = 1) {
  return a + b;
}

int main() {
  int result = add();
  std::cout << "result = " << result << std::endl;

  result = add(3);
  std::cout << "result = " << result << std::endl;

  result = add(3, 4);
  std::cout << "result = " << result << std::endl;
}
```

Functions and references to returned values

```cpp
#include <iostream>

int square(int value) {
  return value * value;
}

int main() {
  int result = square(3);
  std::cout << "3^2 = " << result << std::endl;

  // Not allowed
  int& result_ref = square(3); // build error
  std::cout << "3^2 = " << result_ref << std::endl;

  // Ok
  const int& result_const_ref = square(3); // Ok
  std::cout << "3^2 = " << result_const_ref << std::endl;
}
```

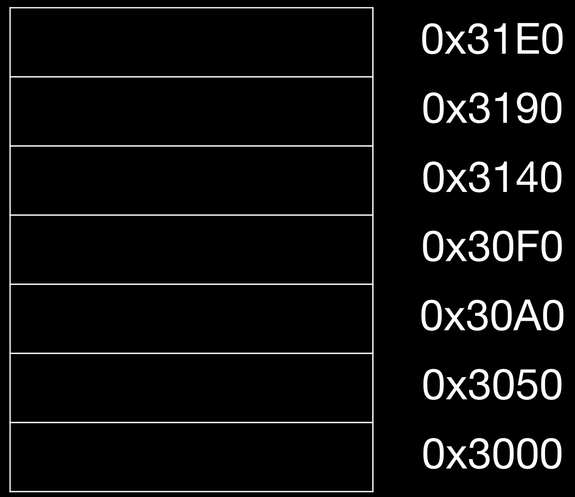
Parameters are passed by value

```
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
```

Parameters are passed by value

```cpp
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
```

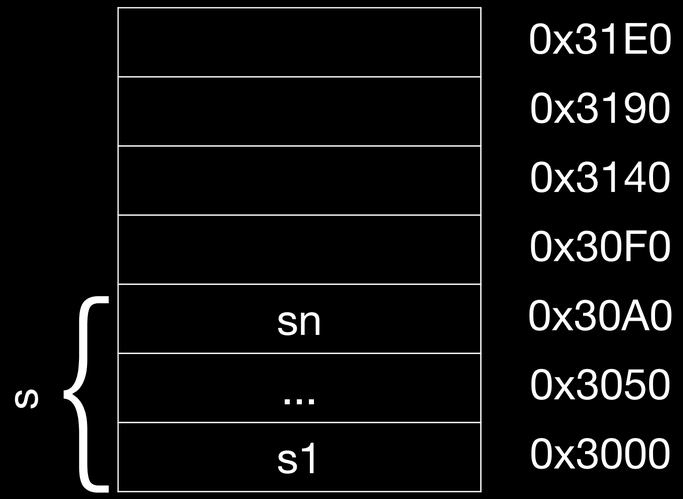
Memory layout



Parameters are passed by value

```
1 struct big_struct {
2   // ...
3 };
4
5 big_struct s;
6
7 // parameter by value
8 void print_val(big_struct p) {
9   // ...
10 }
11 print_val(s); // copy
12
13 // parameter by reference
14 void print_ref(big_struct& q) {
15   // ...
16 }
17 print_ref(s); // no copy
```

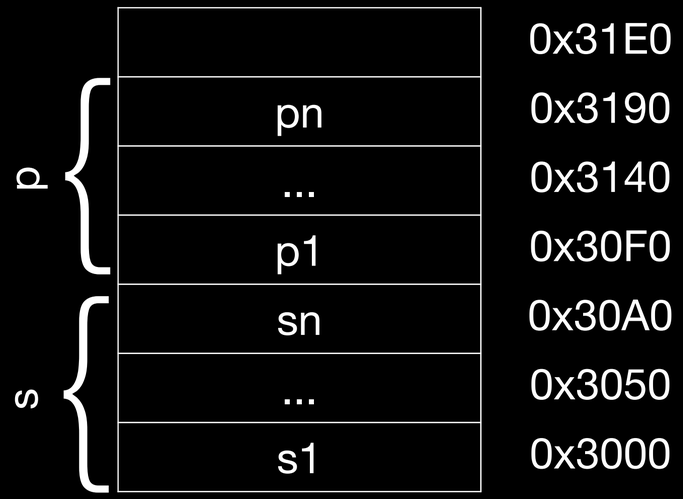
Memory layout



Parameters are passed by value

```
1 struct big_struct {
2   // ...
3 };
4
5 big_struct s;
6
7 // parameter by value
8 void print_val(big_struct p) {
9   // ...
10 }
11 print_val(s); // copy
12
13 // parameter by reference
14 void print_ref(big_struct& q) {
15   // ...
16 }
17 print_ref(s); // no copy
```

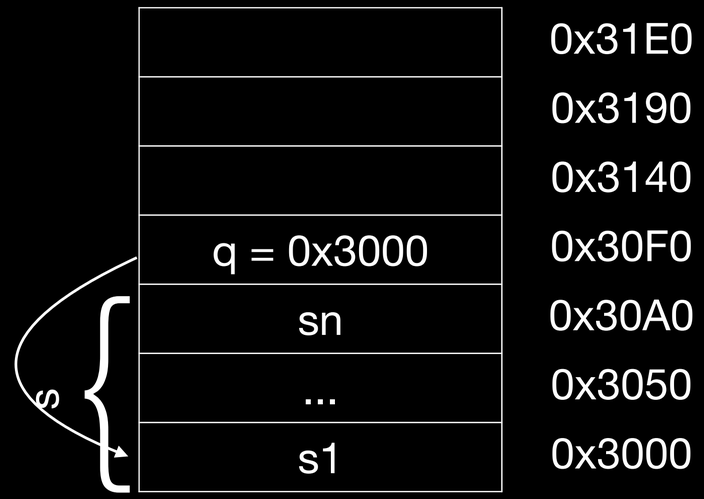
Memory layout



Parameters are passed by value

```cpp
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 ?

```
1 struct small_struct {int a;};
2 small_struct s = {1};
3
4 void change_val(small_struct p) {
5   p.a = 2;
6 }
7 change_val(s);
8 // s.a == 1
9
10 void change_ref(small_struct& g) {
11   q.a = 2;
12 }
13 change_ref(s);
14 // s.a == 2
```

Pass by value or reference ?

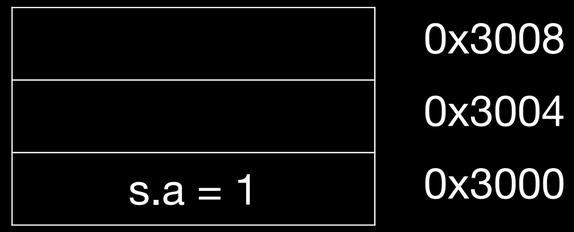
```
1  struct small_struct {int a;};
2  small_struct s = {1};
3
4  void change_val(small_struct p) {
5    p.a = 2;
6  }
7  change_val(s);
8  // s.a == 1
9
10 void change_ref(small_struct& g) {
11   q.a = 2;
12 }
13 change_ref(s);
14 // s.a == 2
```

Memory layout

Pass by value or reference ?

```
1 struct small_struct {int a;};
2 small_struct s = {1};
3
4 void change_val(small_struct p) {
5   p.a = 2;
6 }
7 change_val(s);
8 // s.a == 1
9
10 void change_ref(small_struct& g) {
11   q.a = 2;
12 }
13 change_ref(s);
14 // s.a == 2
```

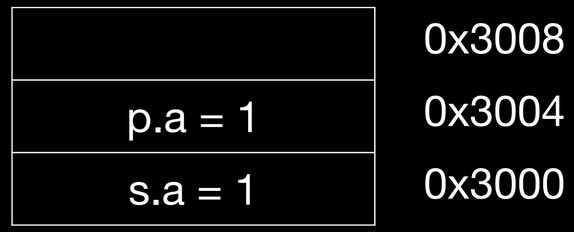
Memory layout



Pass by value or reference ?

```
1 struct small_struct {int a;};
2 small_struct s = {1};
3
4 void change_val(small_struct p) {
5   p.a = 2;
6 }
7 change_val(s);
8 // s.a == 1
9
10 void change_ref(small_struct& g) {
11   q.a = 2;
12 }
13 change_ref(s);
14 // s.a == 2
```

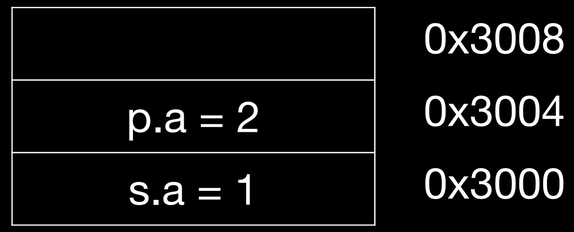
Memory layout



Pass by value or reference ?

```
1 struct small_struct {int a;};
2 small_struct s = {1};
3
4 void change_val(small_struct p) {
5   p.a = 2;
6 }
7 change_val(s);
8 // s.a == 1
9
10 void change_ref(small_struct& g) {
11   q.a = 2;
12 }
13 change_ref(s);
14 // s.a == 2
```

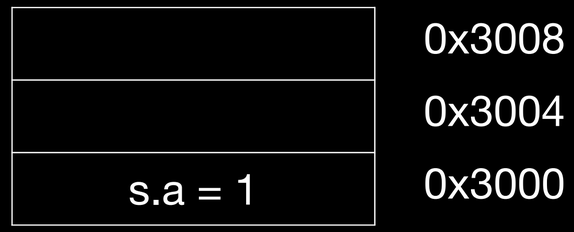
Memory layout



Pass by value or reference ?

```
1 struct small_struct {int a;};
2 small_struct s = {1};
3
4 void change_val(small_struct p) {
5   p.a = 2;
6 }
7 change_val(s);
8 // s.a == 1
9
10 void change_ref(small_struct& g) {
11   q.a = 2;
12 }
13 change_ref(s);
14 // s.a == 2
```

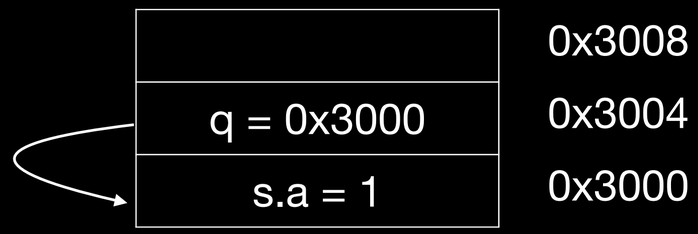
Memory layout



Pass by value or reference ?

```
1 struct small_struct {int a;};
2 small_struct s = {1};
3
4 void change_val(small_struct p) {
5   p.a = 2;
6 }
7 change_val(s);
8 // s.a == 1
9
10 void change_ref(small_struct& g) {
11   q.a = 2;
12 }
13 change_ref(s);
14 // s.a == 2
```

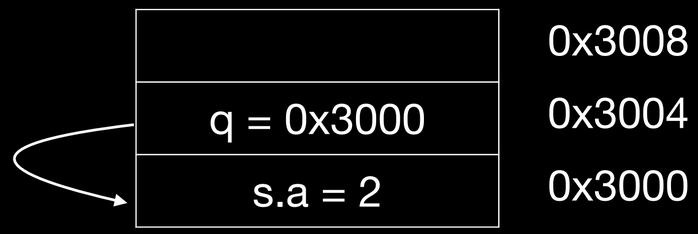
Memory layout



Pass by value or reference ?

```
1 struct small_struct {int a;};
2 small_struct s = {1};
3
4 void change_val(small_struct p) {
5   p.a = 2;
6 }
7 change_val(s);
8 // s.a == 1
9
10 void change_ref(small_struct& g) {
11   q.a = 2;
12 }
13 change_ref(s);
14 // s.a == 2
```

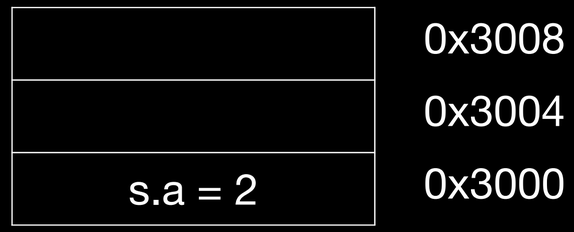
Memory layout



Pass by value or reference ?

```
1 struct small_struct {int a;};
2 small_struct s = {1};
3
4 void change_val(small_struct p) {
5   p.a = 2;
6 }
7 change_val(s);
8 // s.a == 1
9
10 void change_ref(small_struct& g) {
11   q.a = 2;
12 }
13 change_ref(s);
14 // 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

```
1  struct foo {
2    // ...
3  };
4  foo bar;
5
6  void fct_val(foo value);
7  fct_val(bar);   // by value
8
9  void fct_ref(const foo& value);
10 fct_ref(bar);   // by reference
11
12 void fct_ptr(const foo* value);
13 fct_ptr (&bar); // by pointer
14
15 void fct_write(foo& value);
16 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

```
1  int sum(int b);                 // 1
2  int sum(int b, int c);          // 2, ok, overload
3  // float sum(int b, int c); // disallowed
4
5  sum(42);      // calls 1
6  sum(42, 43); // calls 2
7
8  int sum(int b, int c, int d = 4); // 3, overload
9
10 sum(42, 43, 44); // calls 3
11 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

```
1 /// @brief Counts number of dilepton events in data.
2 /// @param a Dataset to search.
3 /// @return The number of dilepton events.
4 unsigned int count_dileptons(data& d) {
5   select_events_with_muons(d);
6   select_events_with_electrons(d);
7   return d.size();
8 }
```

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;
}
```

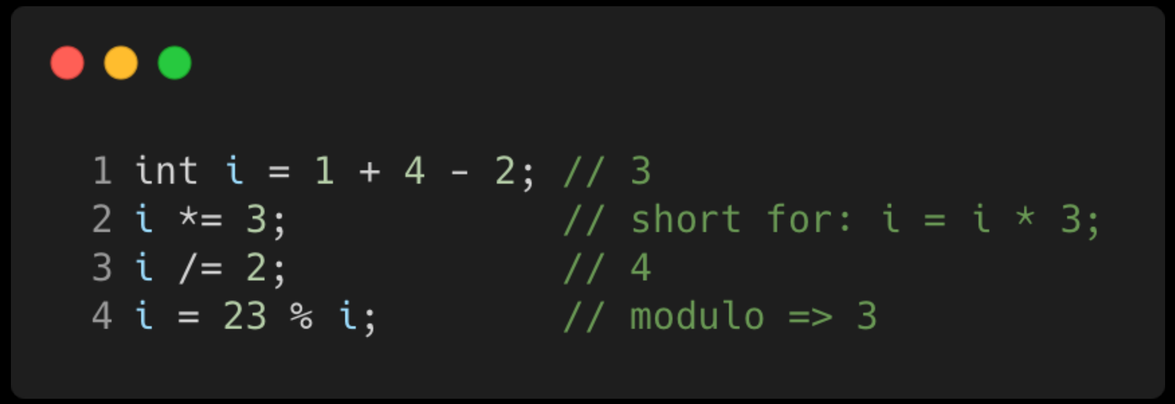
Language Basics

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

Binary and Assignment Operators



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

Increment / Decrement Operators

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

Bitwise and Assignment Operators

```
1 unsigned i = 0xee & 0x55; // 0x44
2 i |= 0xee;                // 0xee
3 i ^= 0x55;                // 0xbb
4 unsigned j = ~0xee;       // 0xffffff11
5 unsigned k = 0x1f << 3;   // 0xf8
6 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

```cpp
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); // std::strong_ordering::equivalent, Since C++20
```

Operator Precedences

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

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

```
1 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
- Struct 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
- Struct 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

```
1 if (condition1) {
2   statement1; statement2;
3 } else if (condition2)
4   only_one_statement;
5 else {
6   statement3;
7   statement4;
8 }
```

if - example

```cpp
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

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

Conditional operator - example

```
1 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

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

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

```
1 switch (identifier) {
2   case c1: statements1; break;
3   case c2: statements2; break;
4   case c3: statements3; break;
5   ...
6   default: statements3; break;
7 }
```

switch - example

```cpp
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

```
1 switch (c) {
2   case 'a':
3     f();    // Warning emitted
4   case 'b': // Warning probably suppressed
5   case 'c':
6     g();
7     [[fallthrough]]; // Warning suppressed
8   case 'd':
9     h();
10 }
```

Init-statements for if and switch

Purpose

- Allows to limit variable scope in if and switch statements

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

Init-statements for if and switch

Don’t confuse with a variable declaration as condition!

```
1 if (value* val = get_value_ptr())
2   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

```
1 for (initializations; condition; increments) {
2   statements;
3 }
```

for loop - example

```
1 for (int i = 0, j = 0; i < 10; i++, j = i * i)
2   std::cout << i << "^2 is " << j << std::endl;
```

Range-based loop - syntax

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

```
1 for (type iteration_vairiable : range) {
2   // body using iteration_riable
3 }
```

Range-based loop - example

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

Init-statements for range-based loop

Purpose

- Allows to limit variable scope in range-based loops

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

while loop - syntax

- Braces are optional if loop body is a single statement

```
1   while (condition) {
2     statements;
3   }
4
5   do {
6     statements;
7   } while (condition);
```

while loop - example

```
1 int j = 2;
2 while (j < 9) {
3   std:: cout << j << ' ';
4   j += 2;
5 }
6 std::cout << '\n';
```

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

Jump statements

Jump statements

- break Exits the loop and continues after it

```
1 data_stream ds = get_data_stream();
2 while (!ds.eof()) {
3   data d;
4   ds.read(&d, 1);
5   if (d == data_breaker)
6     break;
7   consume_data(d);
8 }
```

Jump statements

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

```
1 data_stream ds = get_data_stream();
2 while (!ds.eof()) {
3   data d;
4   ds.read(&d, 1);
5   if (!is_data_valid(d))
6     continue;
7   consume_data(d);
8 }
```

Jump statements

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

```
1 void println (const char* value) {
2   if (value == nullptr) return;
3   std::cout << value << std::endl;
4 }
```

Jump statements

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

```
1 bool is_odd(int value) {
2   if (value % 2) return true;
3   return false;
4 }
```

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!

```
1 void println (const char* value) {
2   if (value == nullptr) goto end;
3   std::cout << value;
4
5   end:
6   std::cout << std::endl;
7 }
```

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
- Struct 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
- Struct and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions

Interface

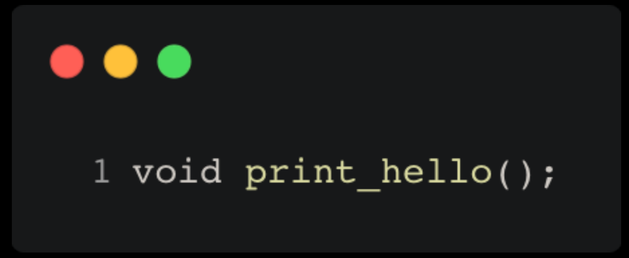
Set of declarations defining some functionality

Interface

Set of declarations defining some functionality

- Put in a so-called “header file”

hello.hpp



Interface

Set of declarations defining some functionality

- Put in a so-called “header file”
- The implementation can be found in the "source file"

hello.cpp

```cpp
#include <iostream>

using namespace std;

void print_hello() {
  cout << "Hello, World!" << endl;
}
```

Interface

print_hello.hpp

```hpp
void print_hello();
```

print_hello.cpp

```cpp
#include <iostream>

using namespace std;

void print_hello() {
  cout << "Hello, World!" << endl;
}
```

main.cpp

```cpp
#include "print_hello.hpp"

int main() {
  print_hello();
}
```

Preprocessor

Preprocessor

- Constant macros

```
1 // constant macro
2 #define THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE 42
```

Preprocessor

- Constant macros
- Function-style macros

```
1 // function-style macro
2 #define CHECK_THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE(x) \
3   if ((x) != THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE)    \
4     std::cerr << #x " was not the response\n";
```

Preprocessor

- Constant macros
- Function-style macros
- Checks

```
1 // compile time or platform specific configuration check
2 #if defined(USE64BITS) || defined (__GNUG__)
3   using my_int = std::uint64_t;
4 #elif
5   using my_int = std::uint32_t;
6 #endif
```

Preprocessor

- Constant macros
- Function-style macros
- Checks

Use preprocessor only in very restricted cases

- Conditional inclusion of headers
- Customization for specific compilers/platforms

Preprocessor

- Constant macros
- Function-style macros
- Checks

Use preprocessor only in very restricted cases

- Conditional inclusion of headers
- Customization for specific compilers/platforms

```
1 #define THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE 42
2
3 constexpr int THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE = 42;
```

Preprocessor

- Constant macros
- Function-style macros
- Checks

Use preprocessor only in very restricted cases

- Conditional inclusion of headers
- Customization for specific compilers/platforms

```
#define IS_THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE(x) \
  ((x) == THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE)

template < typename type_t>
bool IS_THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE (type_t x) {
  return x == static_cast<type_t>(THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE);
}
```

Header include guards

Problem: redefinition by accident

- Headers may define new names (e.g. types)
- Multiple (transitive) inclusions of a header would define those
- names multiple times, which is a compile error

Header include guards

Problem: redefinition by accident

- Headers may define new names (e.g. types)
- Multiple (transitive) inclusions of a header would define those
- names multiple times, which is a compile error
- Solution: guard the content of your headers!

Include guards

```
1 #ifndef MY_HEADER_NAME
2 #define MY_HEADER_NAME
3
4 ... // header file content
5
6 #endif
```

Header include guards

Problem: redefinition by accident

- Headers may define new names (e.g. types)
- Multiple (transitive) inclusions of a header would define those
- names multiple times, which is a compile error
- Solution: guard the content of your headers!

Include guards

```
1 #ifndef MY_HEADER_NAME
2 #define MY_HEADER_NAME
3
4 ... // header file content
5
6 #endif
```

pragma once

```
1 #pragmatism once
2
3 ... // header file content
```

Header / source separation

- Headers should contain declarations of functions / classes
    - Only create them if interface is used somewhere else
- Might be included/compiled many times
- Good to keep them short
- Minimise #include statements
- Put long code in implementation files. Exceptions:
    - Templates and constexpr functions
    - Short functions

Language Basics

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

Benefits

- Robustness: If the expression's type is changed—including when a function return type is changed—it just works.
- Performance: You're guaranteed that there's no conversion.
- Usability: You don't have to worry about type name spelling difficulties and typos.
- Efficiency: Your coding can be more efficient.

Benefits

- Robustness: If the expression's type is changed—including when a function return type is changed—it just works.
- Performance: You're guaranteed that there's no conversion.
- Usability: You don't have to worry about type name spelling difficulties and typos.
- Efficiency: Your coding can be more efficient.

```
1 std::vector<int> v = std::vector<int> {42, 84, 21, 65};
2 float a = v[3];   // conversion intended?
3 int b = v.size(); // bug? unsigned to signed
```

Benefits

- Robustness: If the expression's type is changed—including when a function return type is changed—it just works.
- Performance: You're guaranteed that there's no conversion.
- Usability: You don't have to worry about type name spelling difficulties and typos.
- Efficiency: Your coding can be more efficient.

```
1 std::vector<int> v = std::vector<int> {42, 84, 21, 65};
2 float a = v[3];   // conversion intended?
3 int b = v.size(); // bug? unsigned to signed
```

```
1   auto v = std::vector<int> {42, 84, 21, 65};
2   auto a = v[3];
3   auto b = v.size();
```

Declarations

```
1 auto value = ...
2 auto& value_reference = ...
3 auto* value_pointer = ...
4 const auto constant_value = ...
5 const auto& constant_value_reference = ...
6 const auto* constant_ value_pointer = ...
7 auto [valuel, value2, valuen] = ...
```

for range loop

```cpp
using namespace std;

vector<tuple<int, double>> id_usages = vector<tuple<int, double>> {
  make_tuple(8634, 0.07),
  make_tuple(1482, 0.2),
  make_tuple(4821, 0.15),
  make_tuple(2563, 0.4),
  make_tuple(3920, 0.18)
};

for (const tuple<int, double>& item : id_usages)
  cout << "id: " << get<0>(item) << " => " << get<1>(item) * 100 << "%" << endl;
```

for range loop

```cpp
using namespace std;

auto id_usages = vector {
  make_tuple(8634, 0.07),
  make_tuple(1482, 0.2),
  make_tuple(4821, 0.15),
  make_tuple(2563, 0.4),
  make_tuple(3920, 0.18)
};

for (auto [id, percent] : id_usages)
  cout << "id: " << id << " => " << percent * 100 << "%" << endl;
```

Always initialized

```cpp
using namespace std;

struct my_struct {
  int value;
  float percent;
};

my_struct v1;
cout << "v1.value = " << v1.value << ", v1.percent = " << v1.percent << endl;
// v1.value = 86589520, v1.percent = 2.8026e-45

my_struct v2 {};
cout << "v2.value = " << v2.value << ", v2.percent = " << v2.percent << endl;
// v2.value = 0, v2.percent = 0

auto v3 = my_struct {};
cout << "v3.value = " << v3.value << ", v3.percent = " << v3.percent << endl;
// v3.value = 0, v3.percent = 0
```

Exercise : loops, references, auto

Familiarise yourself with range-based for loops and references.

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

Language Basics

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

Inline functions originally

- Applies to a function to tell the compiler to inline it
    - That is, replace function calls by the function's content (similar to how a macro works)
- Only a hint, compiler can still choose to not inline
- Avoids call overhead at the cost of increasing binary size

```
inline int mult(int a, int b) {
  return a * b;
}
```

Major side effect

- The linker reduces the duplicated functions into one
- An inline function definition can thus live in header files

```
1 inline int mult(int a, int b) {
2   return a * b;
3 }
```

Inline functions nowadays

- Compilers can judge far better when to inline or not
- thus primary purpose is gone
- Putting functions into headers became main purpose
- Many types of functions are marked inline by default:
  - function templates
  - constexpr functions
  - class member functions

```
1 inline int mult(int a, int b) {
2   return a * b;
3 }
```

Inline variables

- Global or static member variable specified as inline
- Same side effect, linker merges all occurrences into one
- Allows to define global variables/ constants in headers
- Avoid global variables and global constants! static member are fine.

my_struct.hpp

```cpp
#pragma once
#include <string>

inline int count_item = 0;
inline constexpr std::string file_name = "output.txt";

struct my_struct {
  inline static int count_item = 0;
  inline static constexpr std::string file_name = "input.txt";

  int value_a;
  int value_b;
};
```

Inline variables

my_struct.hpp

```cpp
#pragma once
#include <string>

inline int count_item = 0;
inline const std::string file_name = "output.txt";

struct my_struct {
  inline static int count_item = 0;
  inline static const std::string file_name = "input.txt";

  int value_a;
  int value_b;
};
```

program.cpp

```cpp
#include "my_struct.hpp"
#include <iostream>

int main() {
  using namespace std;

  count_item += 42;
  cout << "count_item = " << count_item << endl;
  cout << "file_name = " << file_name << endl << endl;

  my_struct::count_item += 21;
  cout << "my_struct::count_item = " << my_struct::count_item << endl;
  cout << "my_struct::file_name = " << my_struct::file_name << endl;
}
```

Language Basics

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

Checking invariants in a program

- An invariant is a property that is guaranteed to be true during certain phases of a program, and the program might crash or yield wrong results if it is violated
    - “Here, ‘a’ should always be positive”
- This can be checked using assert
- The program will be aborted if the assertion fails

```cpp
#include <cassert>

double f(double a) {
  // [...] do stuff with a
  // [...] that should leave it positive
  assert(a > .0);
  return std::sqrt(a);
}
```

Good practice

- Assertions are mostly for developers and debugging
- Use them to check important invariants of your program
- Prefer handling user-facing errors with helpful error messages/exceptions
- Assertions can impact the speed of a program
  - Assertions are disabled when the macro NDEBUG is defined
  - Decide if you want to disable them when you release code

```
1 #define NDEBUG
2 #include <cassert>
3
4 double f(double a) {
5   assert(a > .0); // no effect
6   return std::sqrt(a);
7 }
```

Static assert

- Checking invariants at compile time
  - To check invariants at compile time, use static_assert
  - The assertion can be any constant expression (see later)
  - The message argument is optional in C++ 17 and later

```
1 double f(user_type_t a) {
2   static_assert(std::is_floating_point<user_type_t>::value,
3                 "This function expects floating-point types.");
4   return std::sqrt(a);
5 }
6
```

```
$> static_assert.cpp:8:17 Static assertion failed due to requirement
'std::is_floating_point<int>::value': This function expects floating-point
types.
```

Language Basics

- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Struct 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
- Struct 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