Modern C++ Course

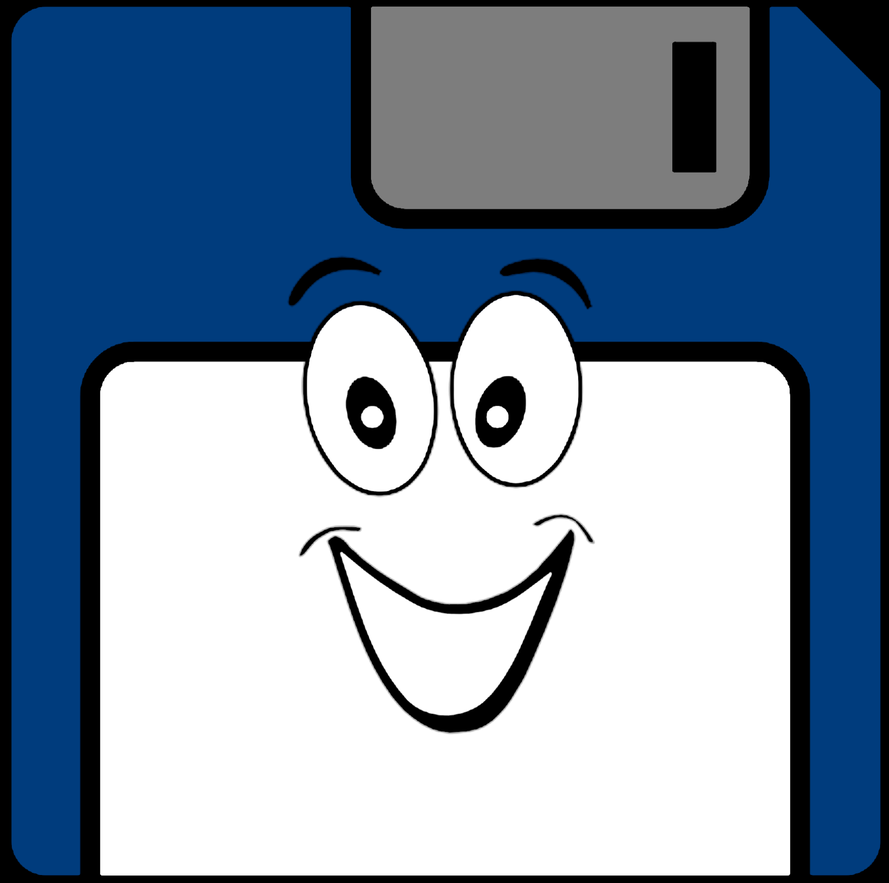
Who am I ?

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

```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

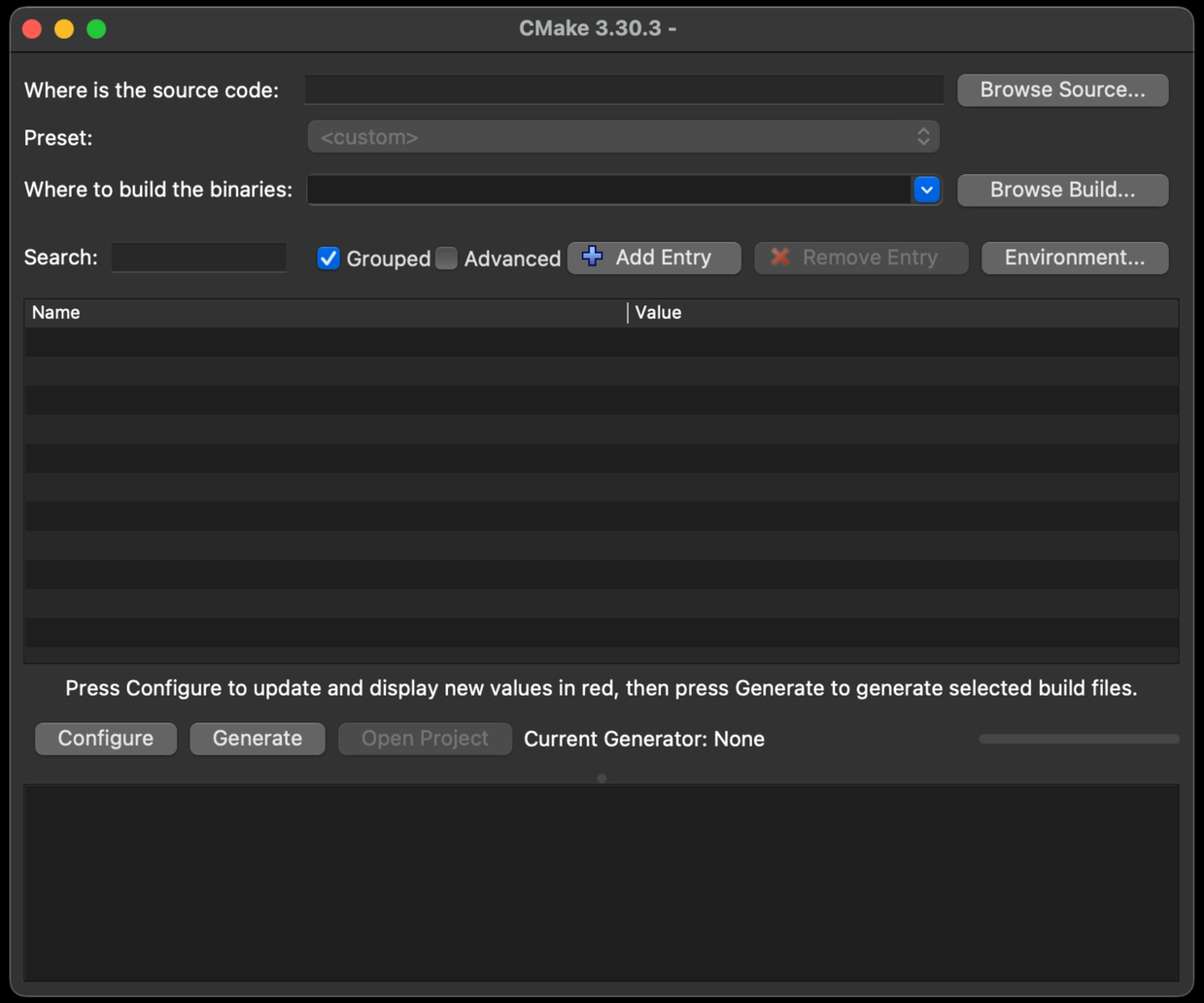
```cmake
cmake_minimum_required(VERSION 3.20)

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 << "maint without arguments" << std::endl;
}
```

Main function

```cpp
#include <iostream>

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

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
  std::cout << "maint 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 a 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

```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 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::bfloat16_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

```cpp
#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

```cpp
#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 !
```

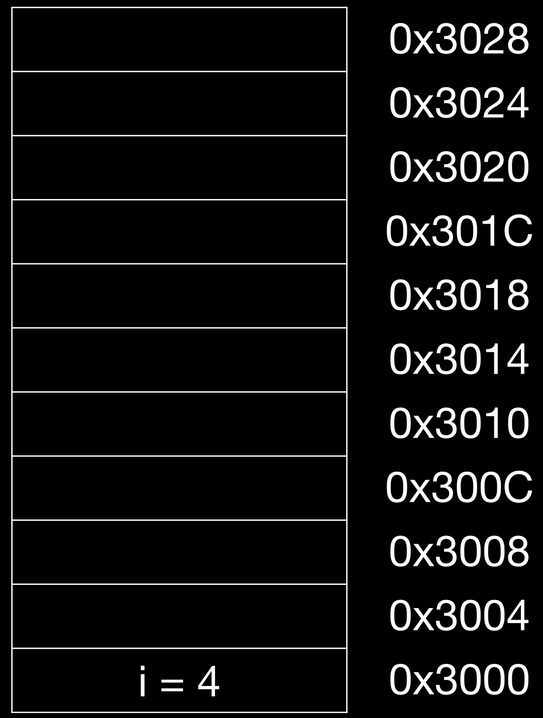
Pointers

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

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

// compile error
int* pak = k;

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

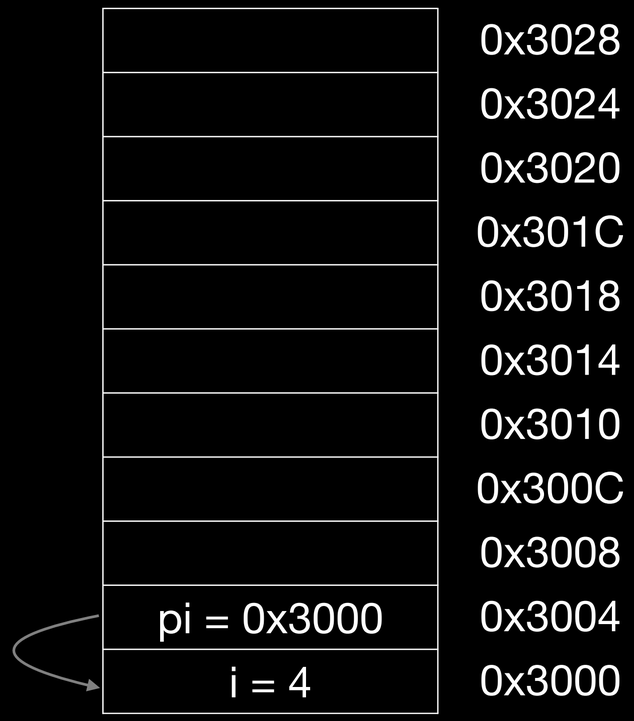
Memory layout



Pointers

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

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

// compile error
int* pak = k;

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

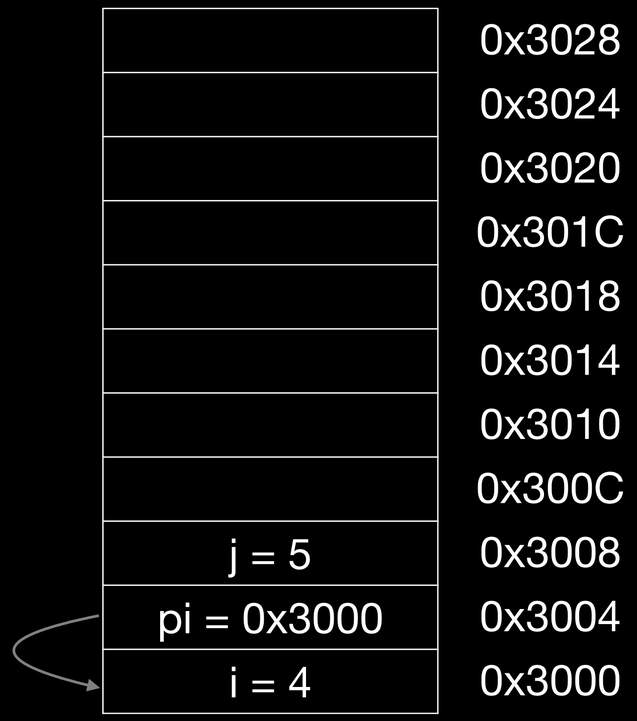
Memory layout



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;
```

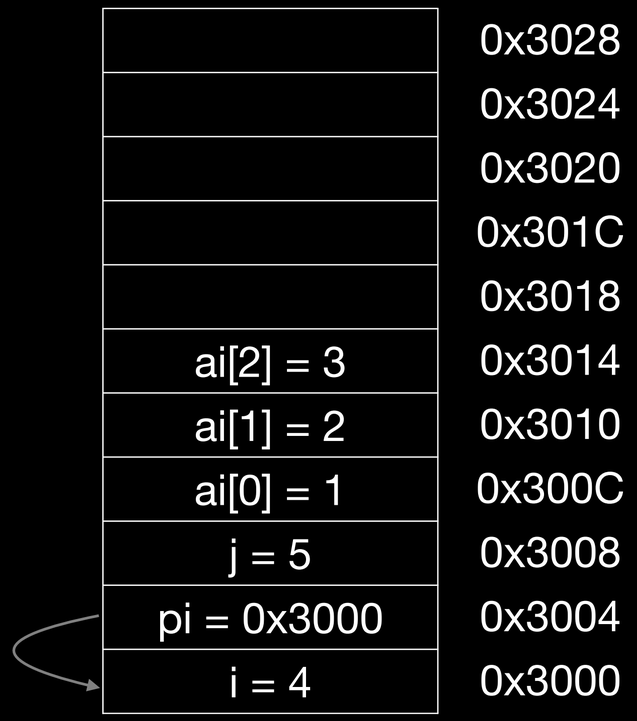
Memory layout



Pointers

```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;
```

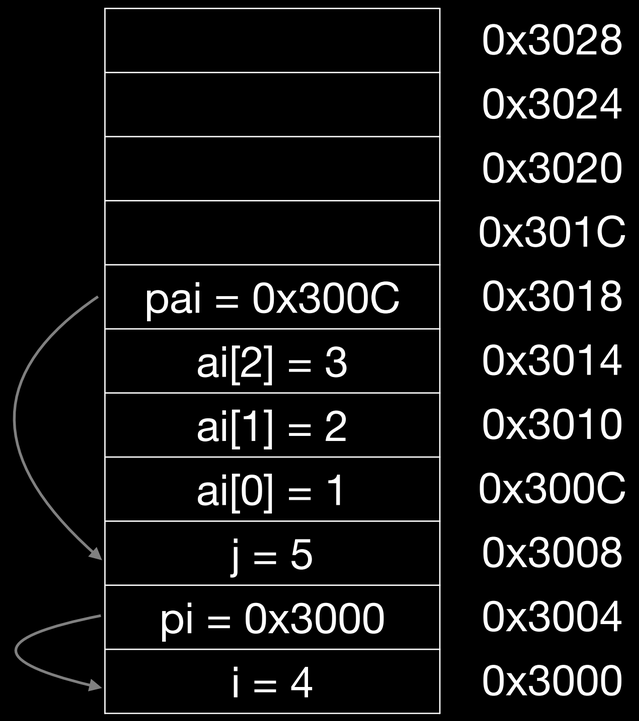
Memory layout



Pointers

```
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;
```

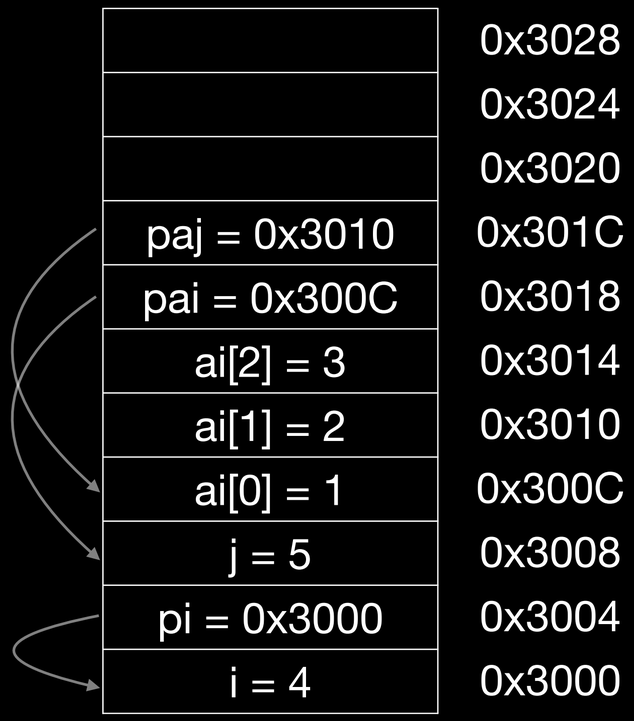
Memory layout



Pointers

```
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;
```

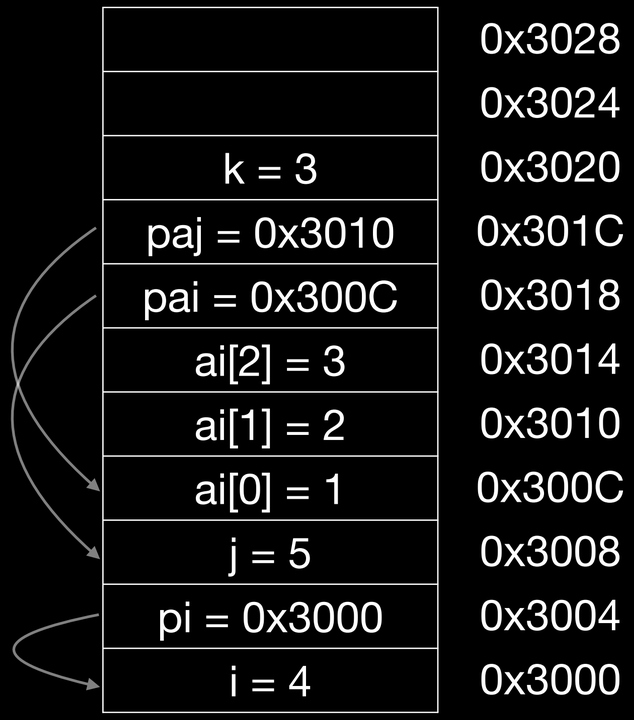
Memory layout



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;
```

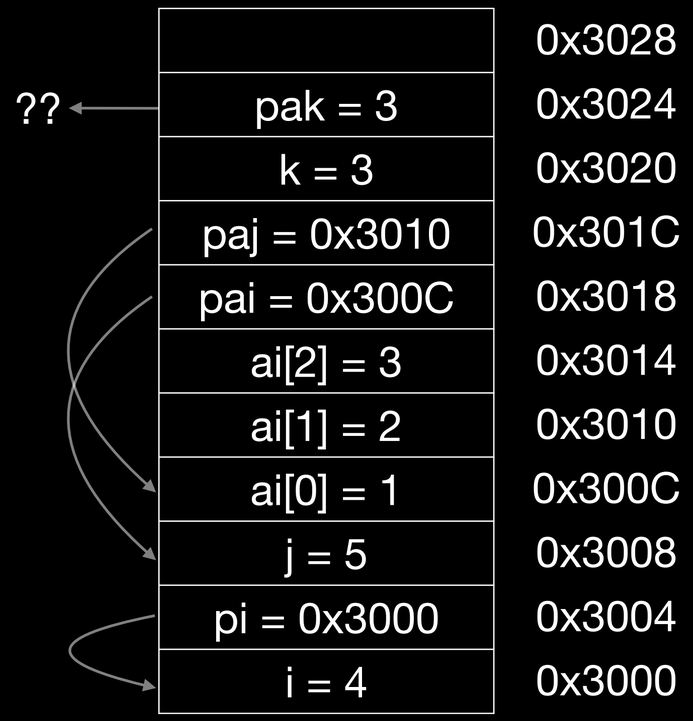
Memory layout



Pointers

```
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



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

```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*)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++

```cpp
#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

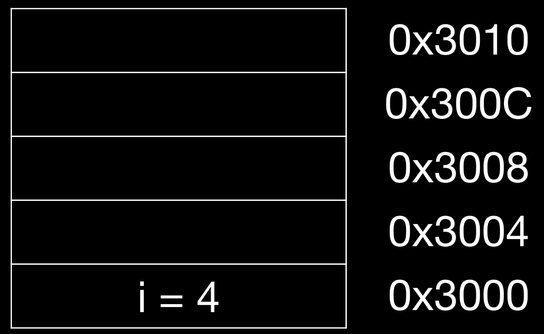
```
int a = 1;
{
  int b[4];
  b[0] = a;
}
// Doesn't compile here:
// b[1] = a + 1;
```

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



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

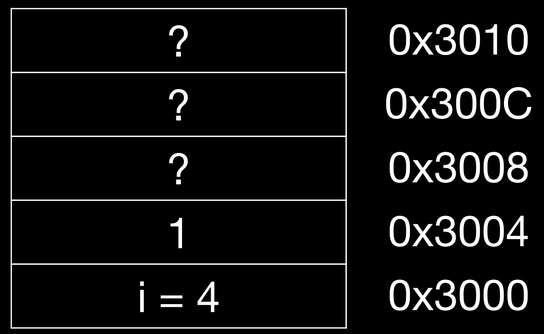
b[3] = ?	0x3010
b[2] = ?	0x300C
b[1] = ?	0x3008
b[0] = 1	0x3004
i = 4	0x3000

Scope and lifetime of variables

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

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

```
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 use 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 use 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 use 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 use 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

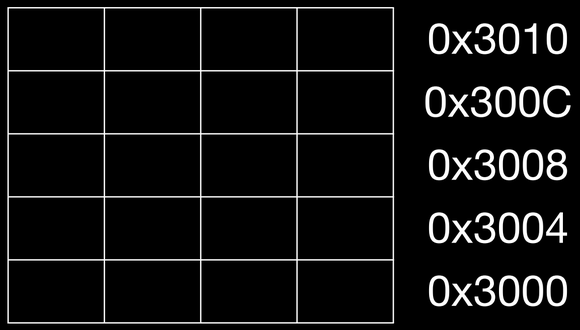
```
struct individual {
  unsigned char age;
  float weight;
};

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

```cpp
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

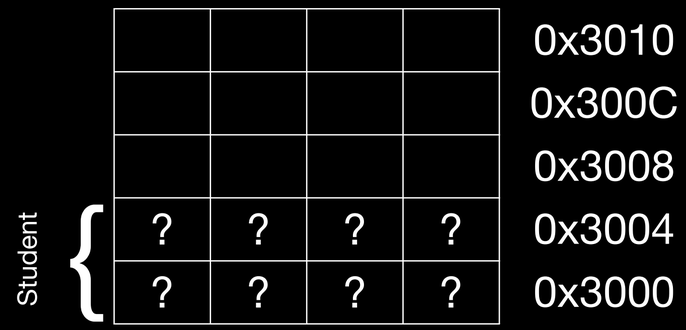


Struct

"members" grouped together under one name

```cpp
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

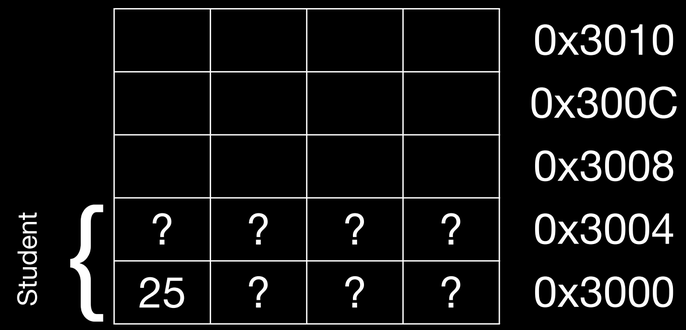


Struct

"members" grouped together under one name

```cpp
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

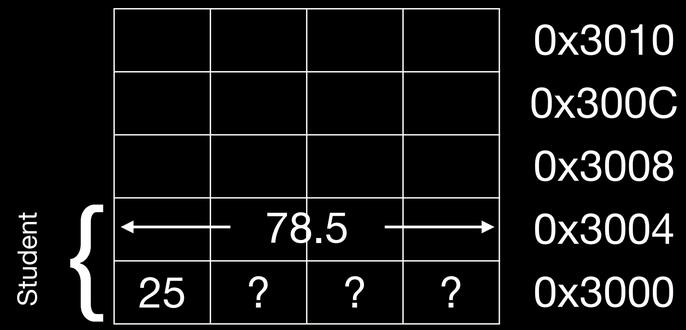


Struct

"members" grouped together under one name

```cpp
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

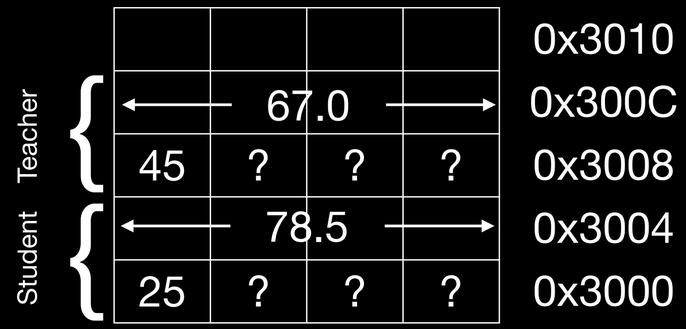


Struct

"members" grouped together under one name

```cpp
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

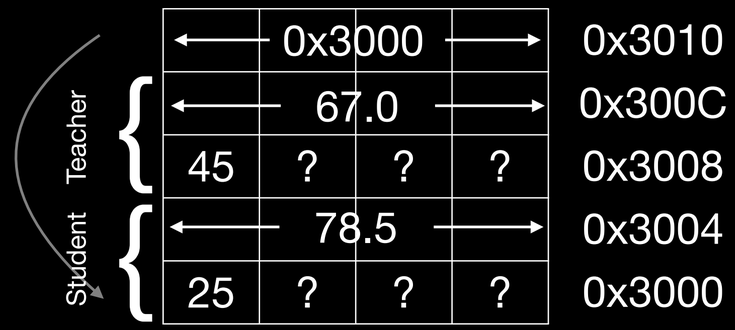


Struct

"members" grouped together under one name

```cpp
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

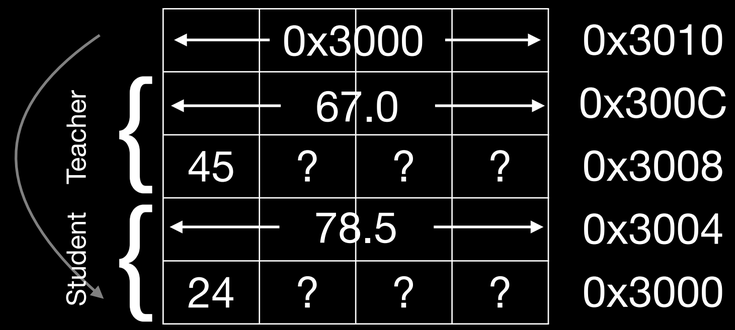


Struct

"members" grouped together under one name

```cpp
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

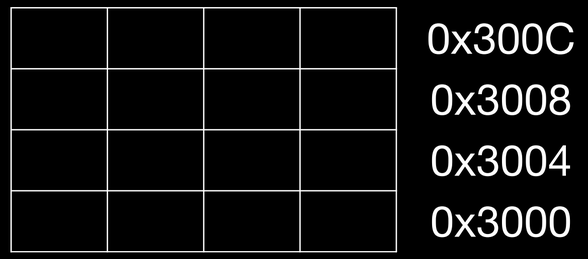
```
union duration {
  int seconds;
  short hours;
  char days;
};

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

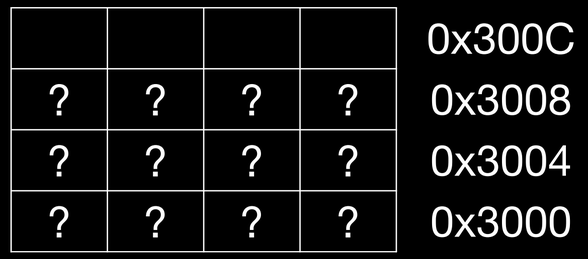


Union

"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

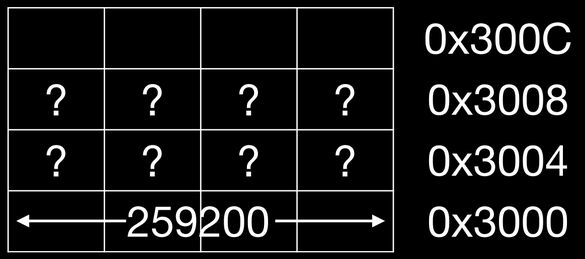


Union

"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

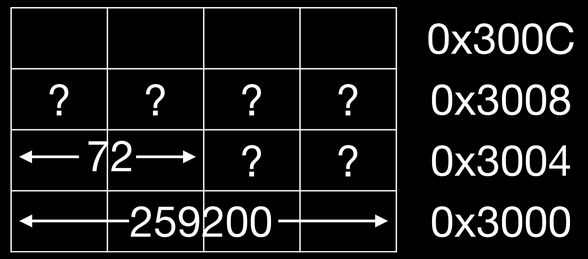


Union

"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

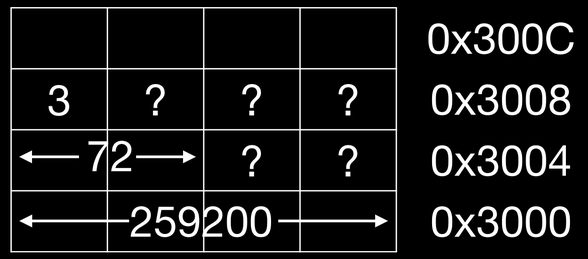


Union

"members" packed together at same memory location

```
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

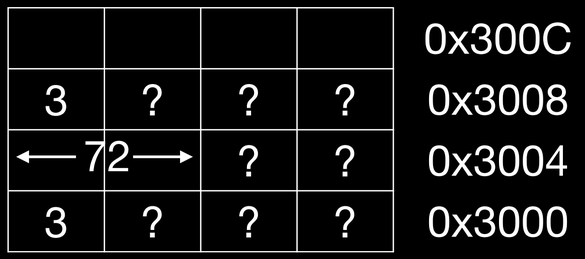


Union

"members" packed together at same memory location

```
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

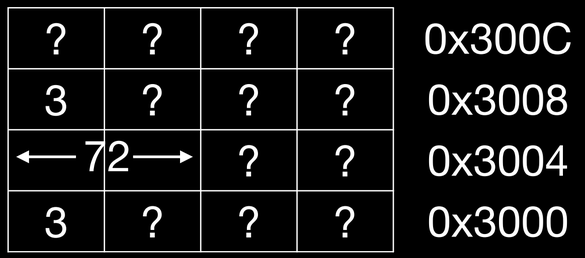


Union

"members" packed together at same memory location

```
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

```
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

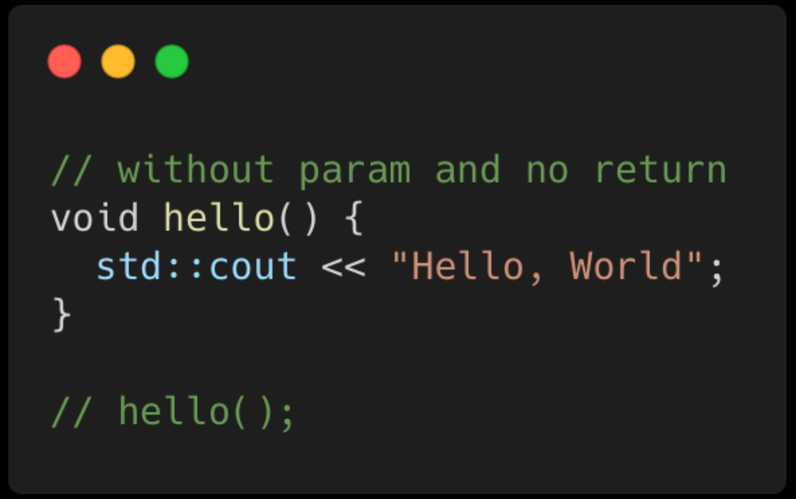
- Hello World
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- auto keyword
- inline keyword
- Assertions

Functions

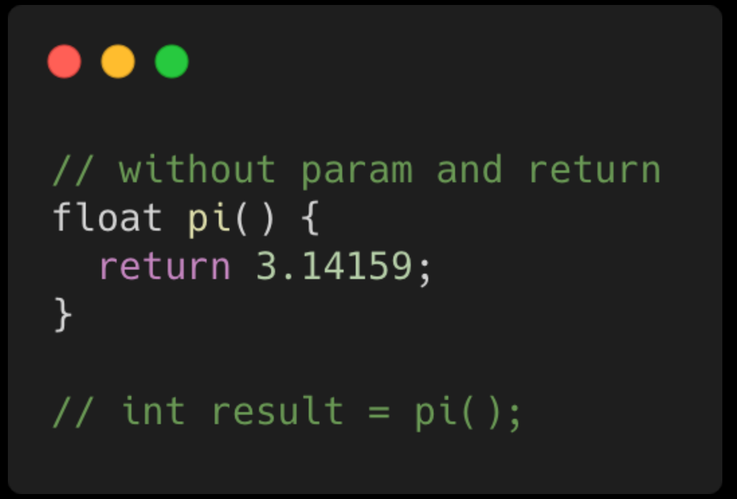
Functions

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

// hello();
```

Functions



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

// hello();
```



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

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

Functions

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

// hello();
```

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

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

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

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

Functions

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

// hello();
```

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

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

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

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

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

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

Functions

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

// hello();
```

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

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

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

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

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

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

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

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

Functions

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

// hello();
```

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

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

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

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

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

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

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

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

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

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

Functions

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

// hello();
```

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

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

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

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

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

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

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

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

```cpp
// 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

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

// hello();
```

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

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

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

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

```cpp
// 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);
```

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

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

```cpp
// 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);
```

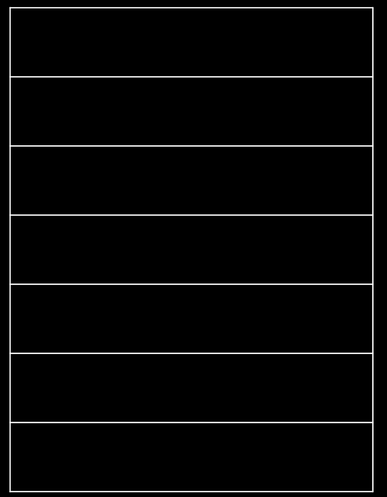
Parameter are passed by value

```
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
```

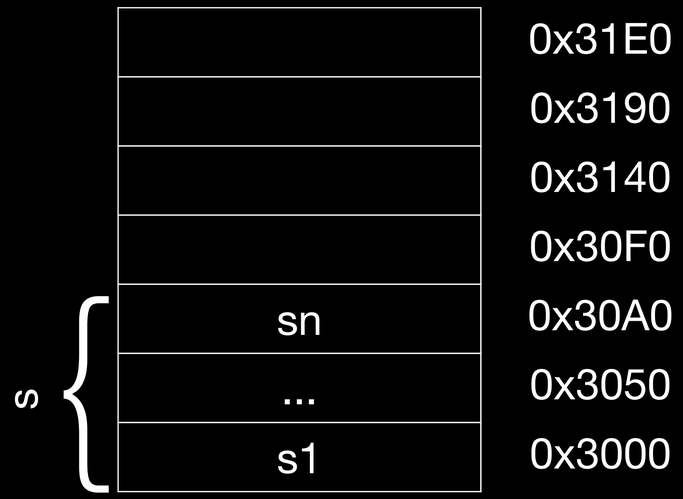
Memory layout



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
```

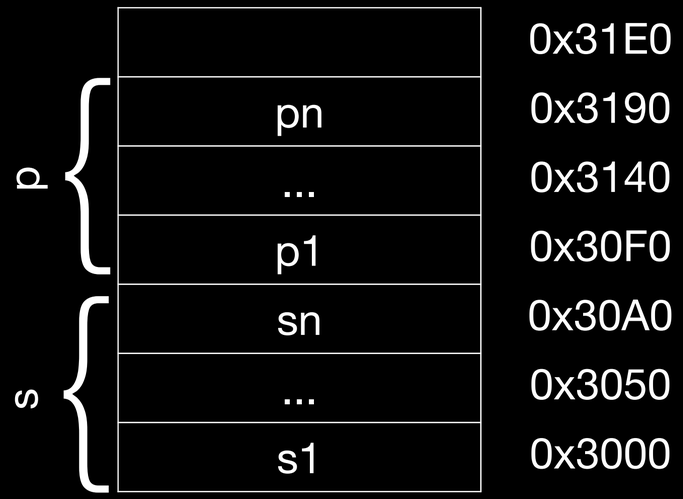
Memory layout



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
```

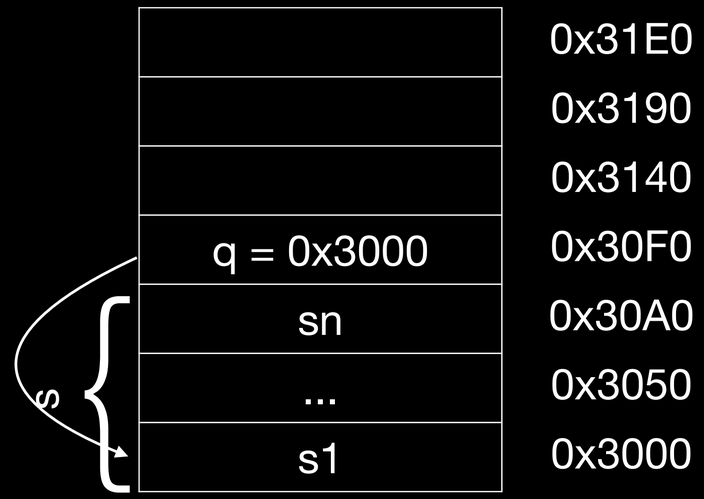
Memory layout



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 ?

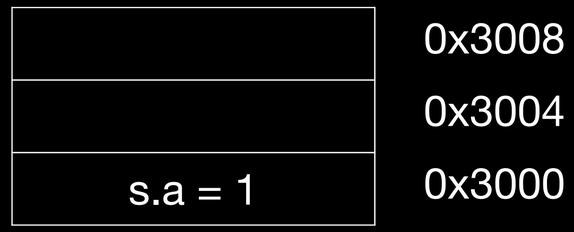
```
struct small_struct {int a;};
small_struct s = {1};

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 ?

```cpp
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
```

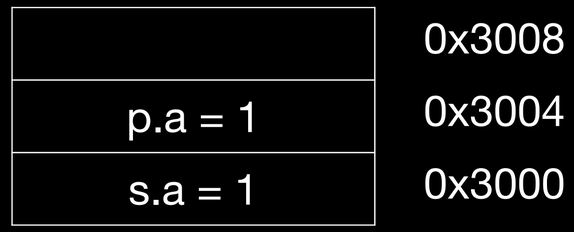
Memory layout



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
```

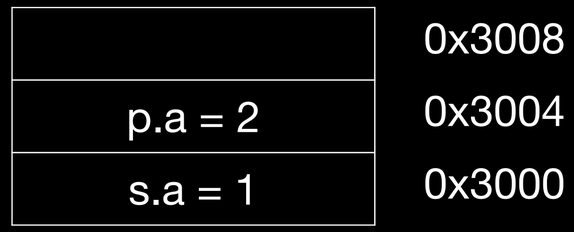
Memory layout



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
```

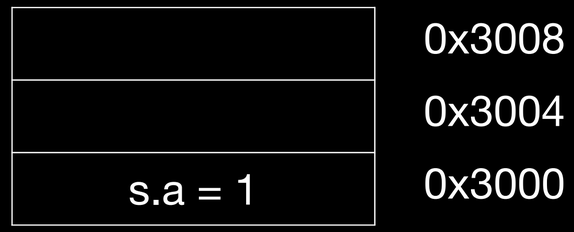
Memory layout



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
```

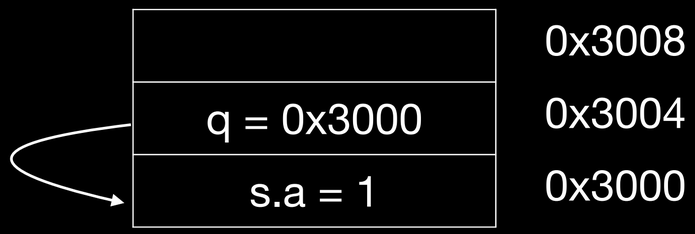
Memory layout



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
```

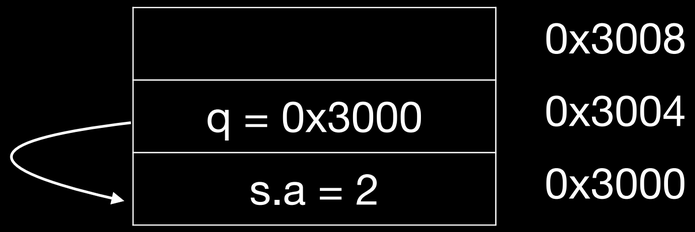
Memory layout



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
```

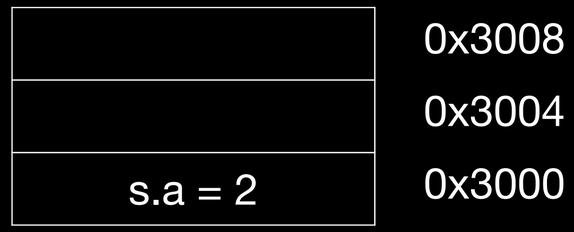
Memory layout



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;       // 0xffffff11
unsigned k = 0x1f << 3;   // 0xf8
unsigned l = 0x1f >> 2;   // 0x7
```

Logical Operators

```cpp
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); // 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

```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

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

Conditional operator - example

```
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

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

```cpp
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

Interface

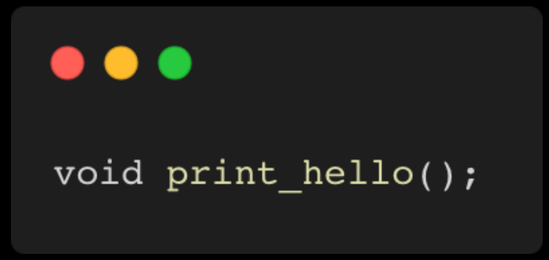
Set of declarations defining some functionality

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 "hello.hpp"
#include <iostream>

using namespace std;

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

Interface

hello.hpp

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

hello.cpp

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

using namespace std;

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

main.cpp

```cpp
#include "hello.hpp"

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

Preprocessor

Preprocessor

- Constant macros

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

Preprocessor

- Constant macros
- Function-style macros

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

Preprocessor

- Constant macros
- Function-style macros
- Checks

```
// compile time or platform specific configuration check
#if defined(USE64BITS) || defined(__GNUG__)
  using my_int = std::uint64_t;
#elif
  using my_int = std::uint32_t;
#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

```
#define THE_ANSWER_TO_THE_ULTIMATE_QUESTION_OF_LIFE 42

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!

```
#ifndef MY_HEADER_NAME
#define MY_HEADER_NAME

... // header file content

#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

```
#ifndef MY_HEADER_NAME
#define MY_HEADER_NAME

... // header file content

#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

```
#ifndef MY_HEADER_NAME
#define MY_HEADER_NAME

... // header file content

#endif
```

pragma once

```
#pragma once

... // 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:
    - Short functions
    - Templates and constexpr functions

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

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.

```
std::vector<int> v = std::vector<int>(42, 84, 21, 65);
float a = v[3];   // conversion intended?
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.

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

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

Declarations

```
auto value = ...
auto& value_reference = ...
auto* value_pointer = ...
const auto constant_value = ...
const auto& constant_value_reference = ...
const auto* constant_value_pointer = ...
auto [value1, 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)
  std::cout << "id : " << id << " => " << percent * 100 << "%"  << std::endl;
```

Always initialized

```cpp
struct my_struct {
  int value;
  float percent;
};

my_struct v1;
std::cout << "v1.value = " << v1.value << ", f1.percent = " << v1.percent << std::endl;
// v1.value = -389758896, v1.percent = 1.4013e-45

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

auto v3 = my_struct {};
std::cout << "v3.value = " << v3.value << ", v3.percent = " << v3.percent << std::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
- 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