

# Modern C++ Course



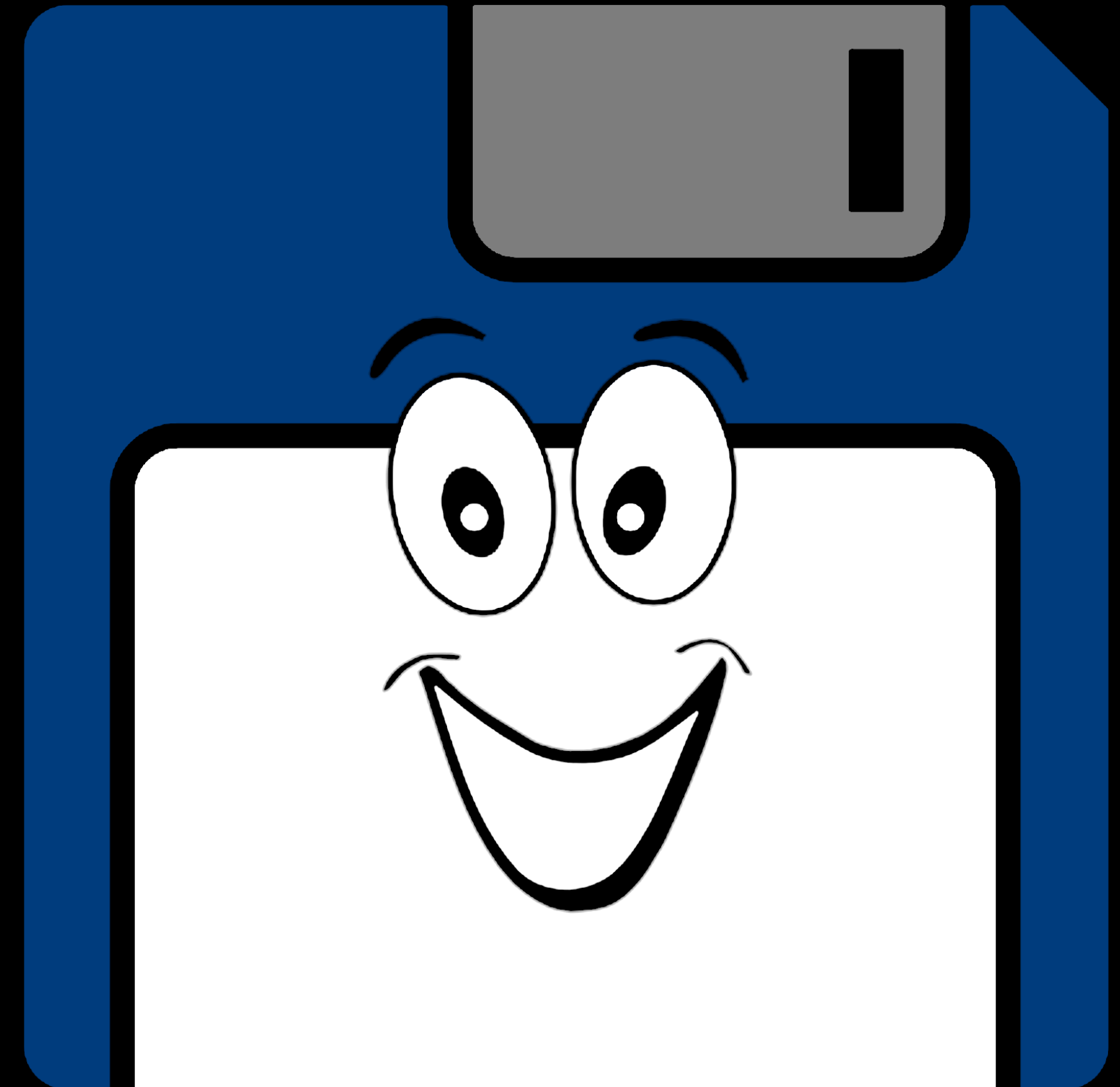
# Who am I ?

## 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 classic first application





# The classic first application

program.cpp




```
#include <iostream>

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



# The classic first application


## program.cpp



```
#include <iostream>

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

## CMakeLists.txt



```
cmake_minimum_required(VERSION 3.20)

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



# The classic first application

## program.cpp

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

## CMakeLists.txt

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

## Output

```
● ● ●  
  
> Hello, World!
```



# The classic first application

## program.cpp

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

## CMakeLists.txt

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

## Output

```
● ● ●  
  
> Hello, World!
```



# Main function



```
#include <iostream>

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



# Main function



```
#include <iostream>

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



```
#include <iostream>

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



# 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



```
#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



```
#include <cstddef> // (and others) defines:
```

```
int value = 42;  
std::size_t size = sizeof(value); // 4  
std::size_t size = sizeof(int);   // 4  
std::size_t size = sizeof(42);    // 4
```



# Pointer to integer



```
#include <cstdint> // defines:
```

```
int value1 = 42;
```

```
int value2 = 84;
```

```
// can hold any diff between two pointers
```

```
std::ptrdiff_t ptrdiff = &value2 - &value1;
```

```
// can hold any pointer value
```

```
std::intptr_t intptr = reinterpret_cast<intptr_t>(&value1);
```

```
std::uintptr_t uintptr = reinterpret_cast<uintptr_t>(&value2);
```



# Language Basics

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



# Language Basics

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



# Static arrays



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



# Pointers



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

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

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

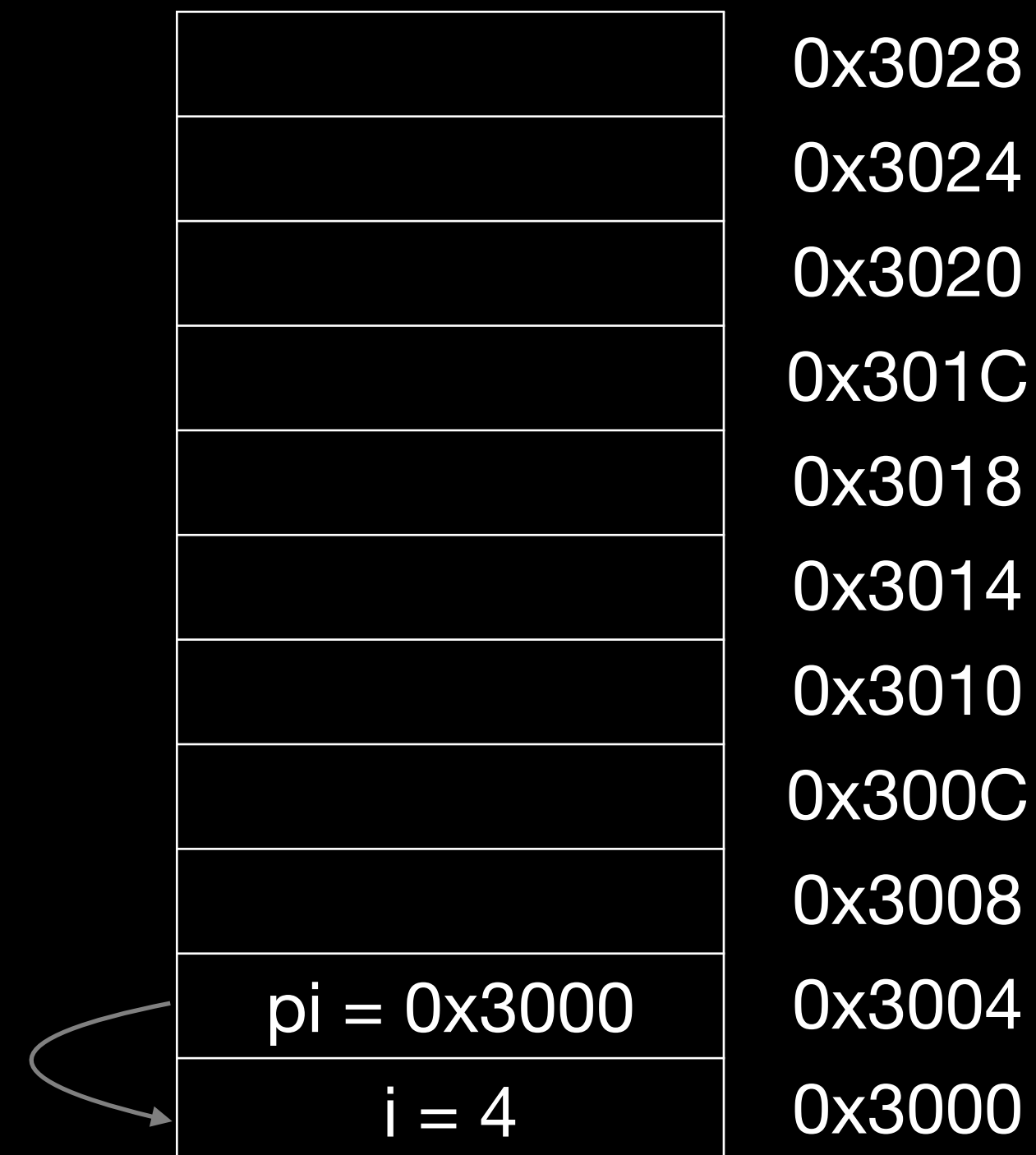


# Pointers



```
int i = 4;  
int* pi = &i;  
int j = *pi + 1;  
  
int ai[] = {1, 2, 3};  
int* pai = ai; // decay to pointer  
int* paj = 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

	0x3028
	0x3024
	0x3020
	0x301C
	0x3018
	0x3014
	0x3010
	0x300C
	0x3008
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000



# Pointers



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

```
int ai[] = {1, 2, 3};
```

```
int* pai = ai; // decay to pointer  
int* paj = pai + 1;  
int k = *paj + 1;
```

```
// compile error  
int* pak = k;
```

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

## Memory layout

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



# Pointers



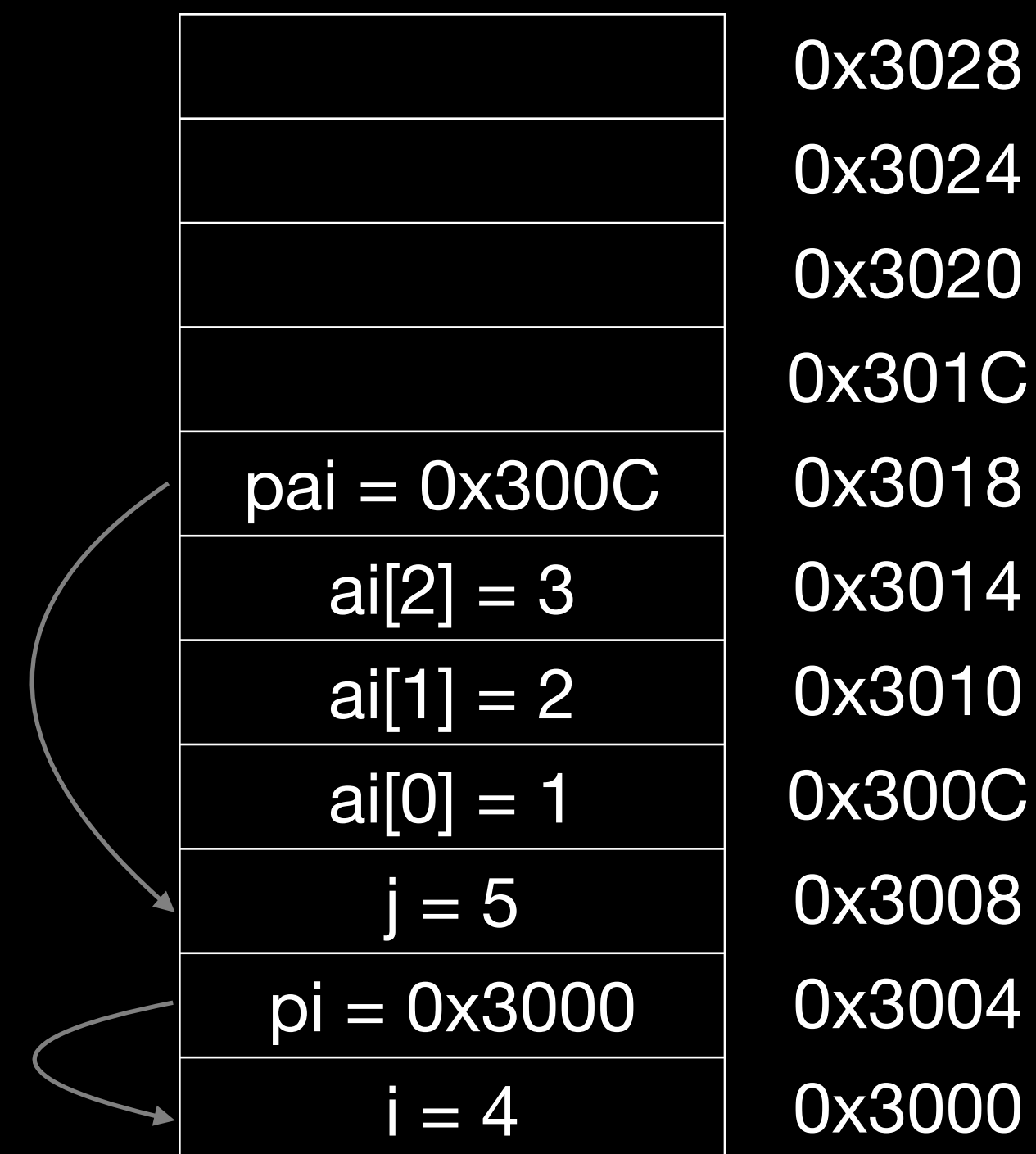
```
int i = 4;
int* pi = &i;
int j = *pi + 1;

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

// compile error
int* pak = k;

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

## Memory layout



# Pointers



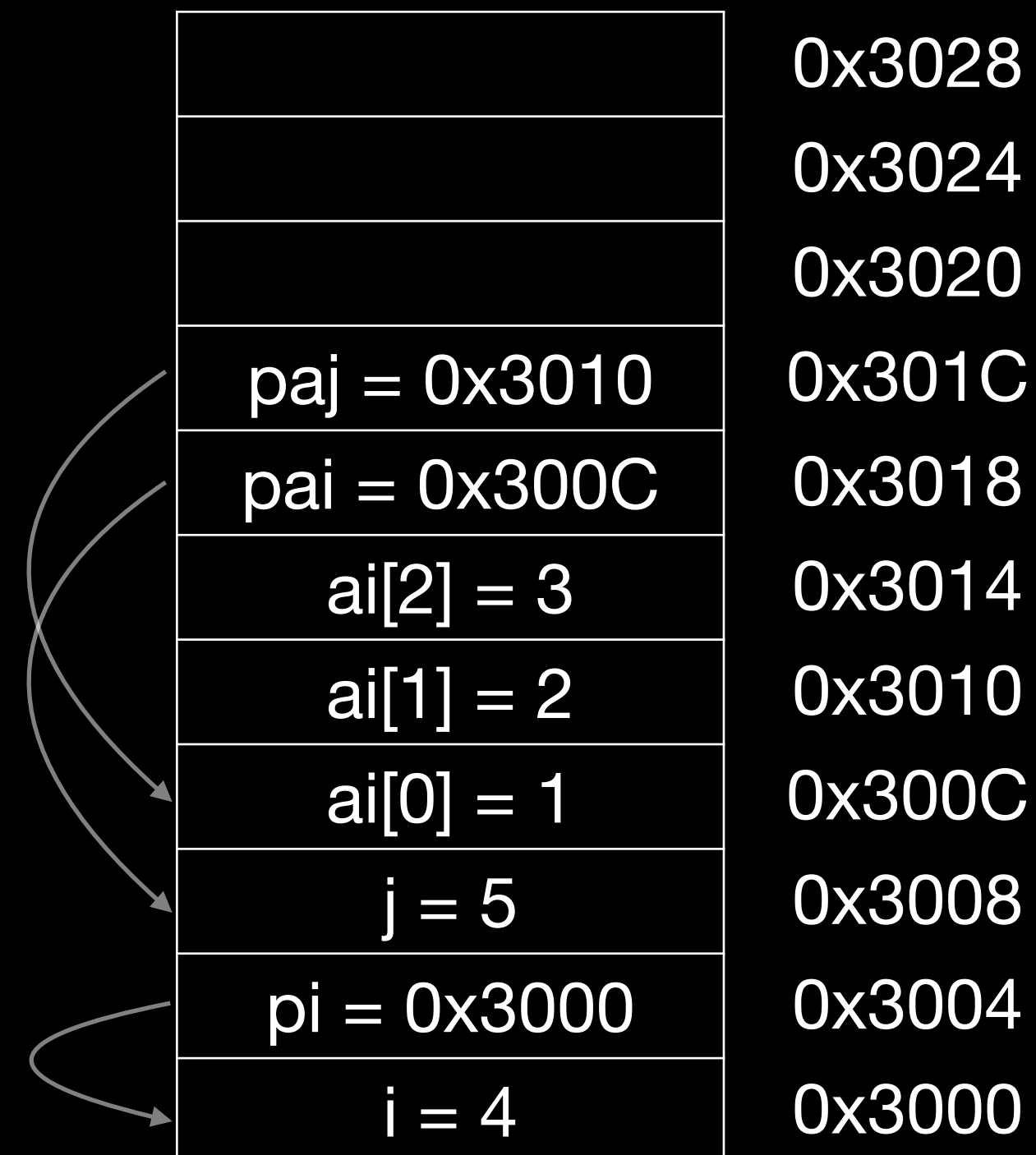
```
int i = 4;
int* pi = &i;
int j = *pi + 1;

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



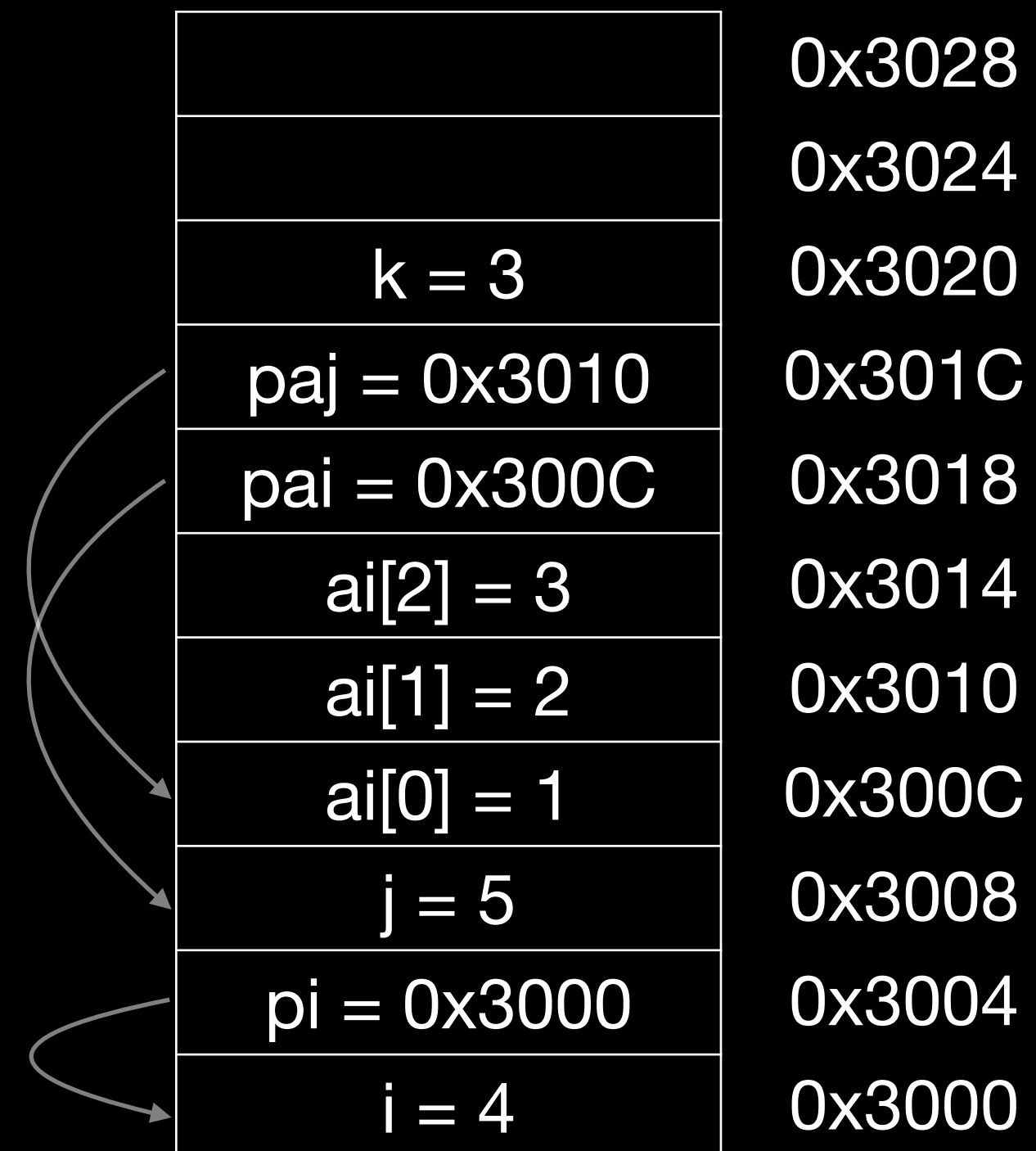
```
int i = 4;
int* pi = &i;
int j = *pi + 1;

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



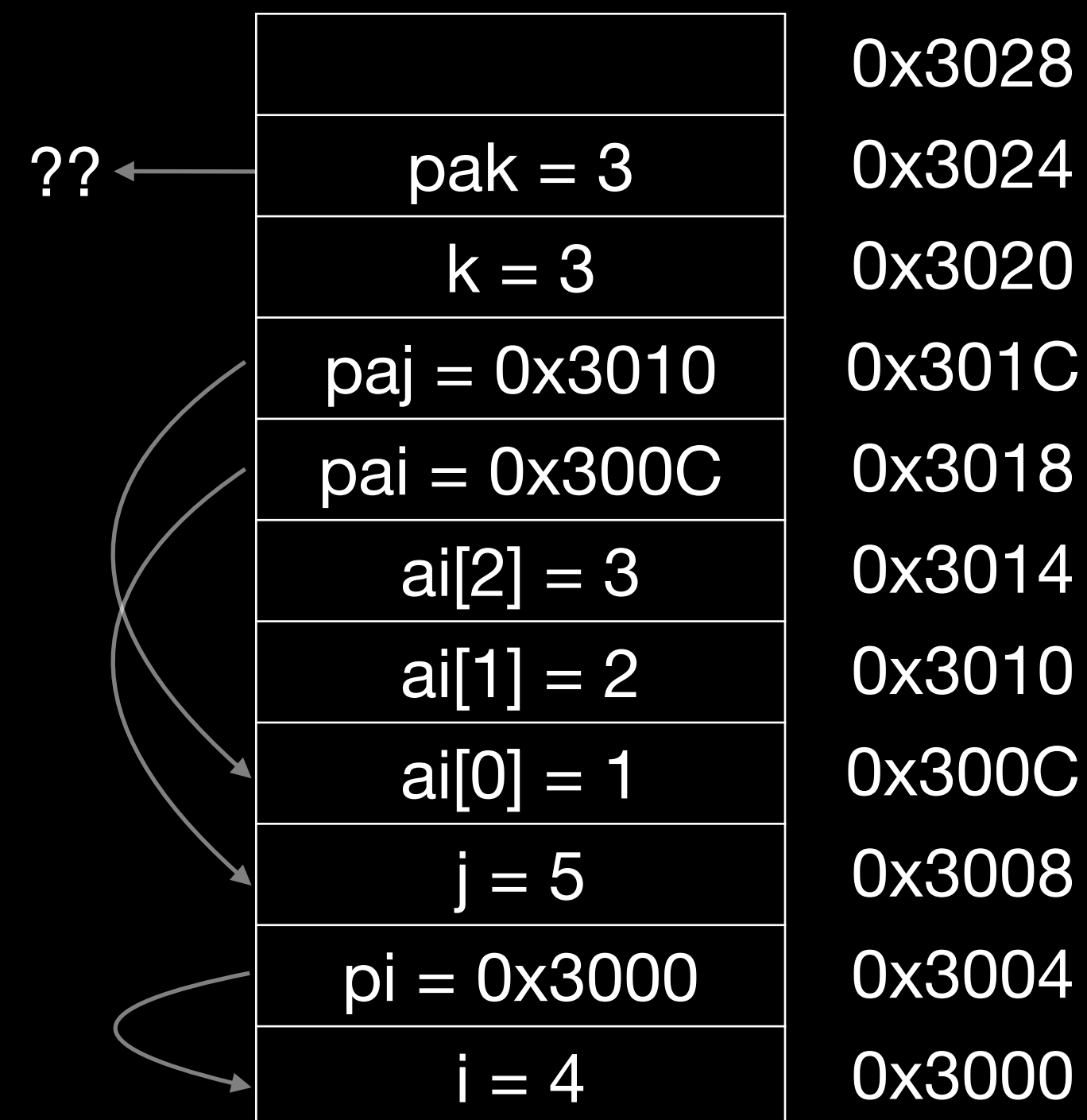
```
int i = 4;
int* pi = &i;
int j = *pi + 1;

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



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

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

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

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

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



# Dynamic arrays using C++



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

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

delete[] ai; // release array memory

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

delete pi; // release scalar memory
```



# Language Basics

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



# Language Basics

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



# Scope

Portion of the source code where a given name is valid

Typically :

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



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




# Scope and lifetime of variables

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



# Scope and lifetime of variables

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



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



# Scope and lifetime of variables

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

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


## Memory layout

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



# Scope and lifetime of variables

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




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

## Memory layout

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

# Scope and lifetime of variables

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



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

## Memory layout

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

# Scope and lifetime of variables

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

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

## Memory layout

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

# Namespaces

- 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 {  
    // ...  
}
```



# Anonymous namespace

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



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

equivalent



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



# Language Basics

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



# Language Basics


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





# Struct

“members” grouped together under one name



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



# Struct

“members” grouped together under one name

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

## Memory layout

				0x3010
				0x300C
				0x3008
				0x3004
				0x3000

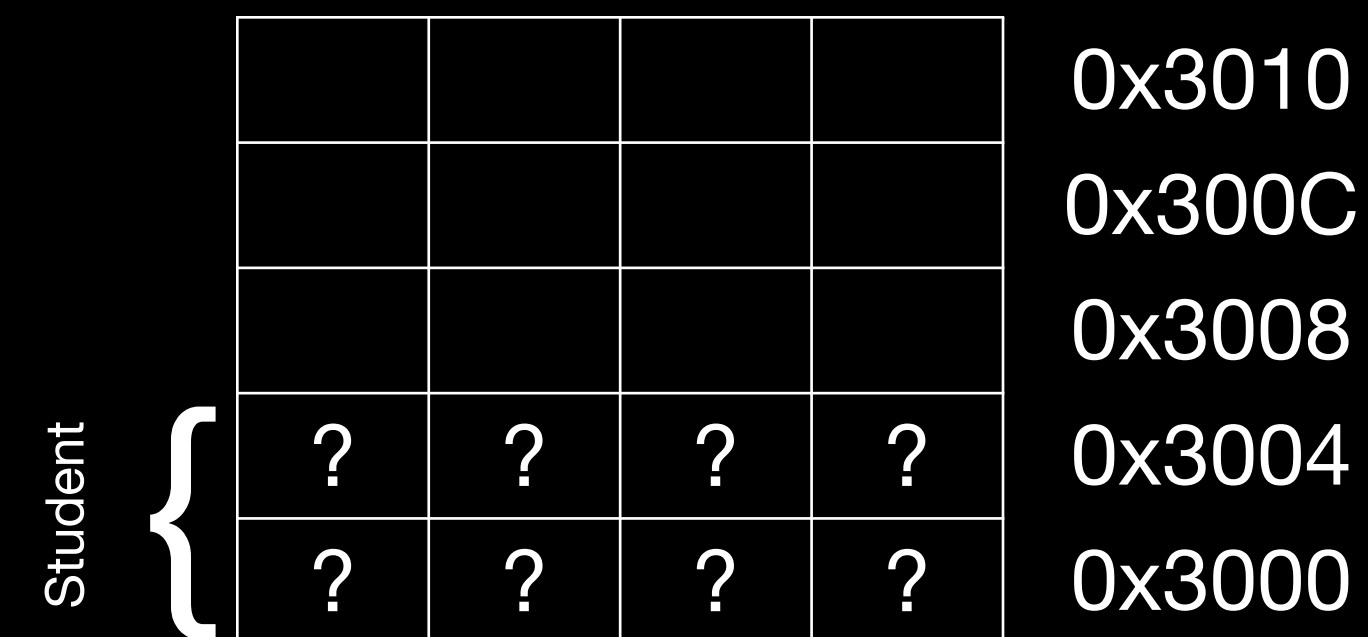


# Struct

“members” grouped together under one name

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

## Memory layout



# Struct

“members” grouped together under one name

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

## Memory layout

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



# Struct

“members” grouped together under one name

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

## Memory layout



# Struct

“members” grouped together under one name

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

## Memory layout

Student Teacher	{					0x3010
		←	67.0	→		0x300C
	{	45	?	?	?	0x3008
		←	78.5	→		0x3004
		25	?	?	?	0x3000



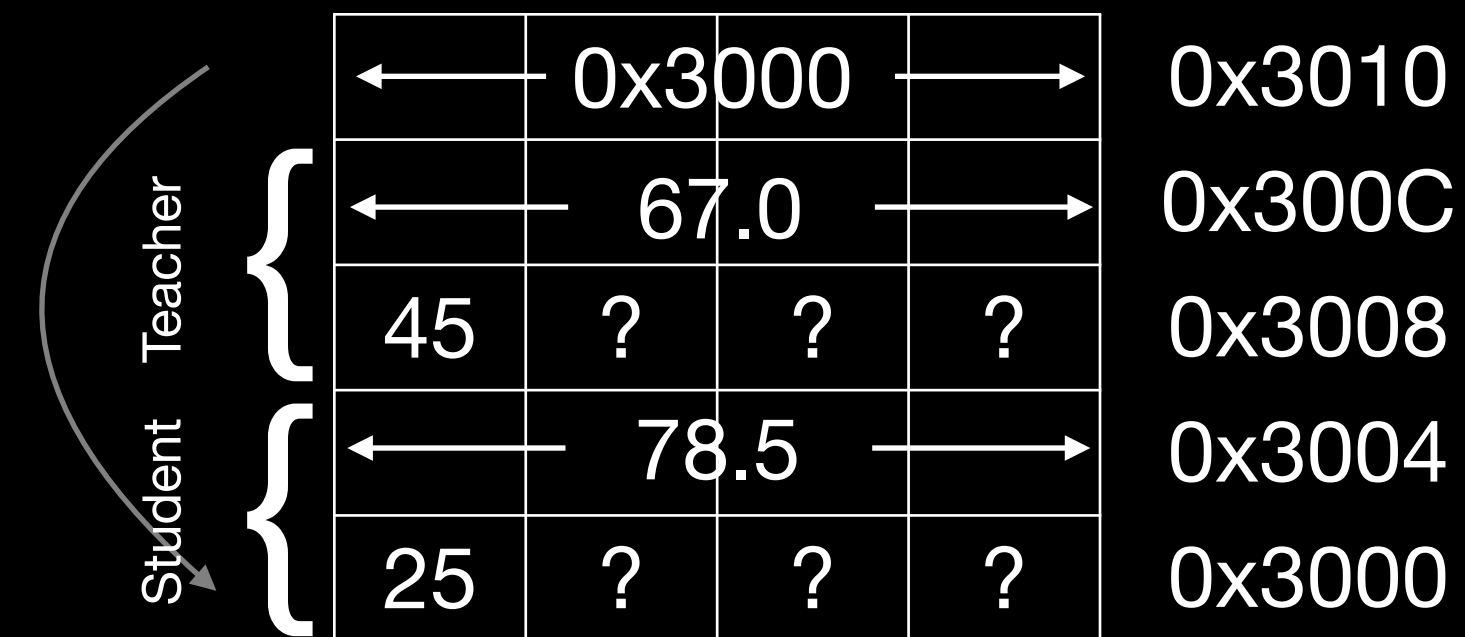


# Struct

“members” grouped together under one name

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

## Memory layout

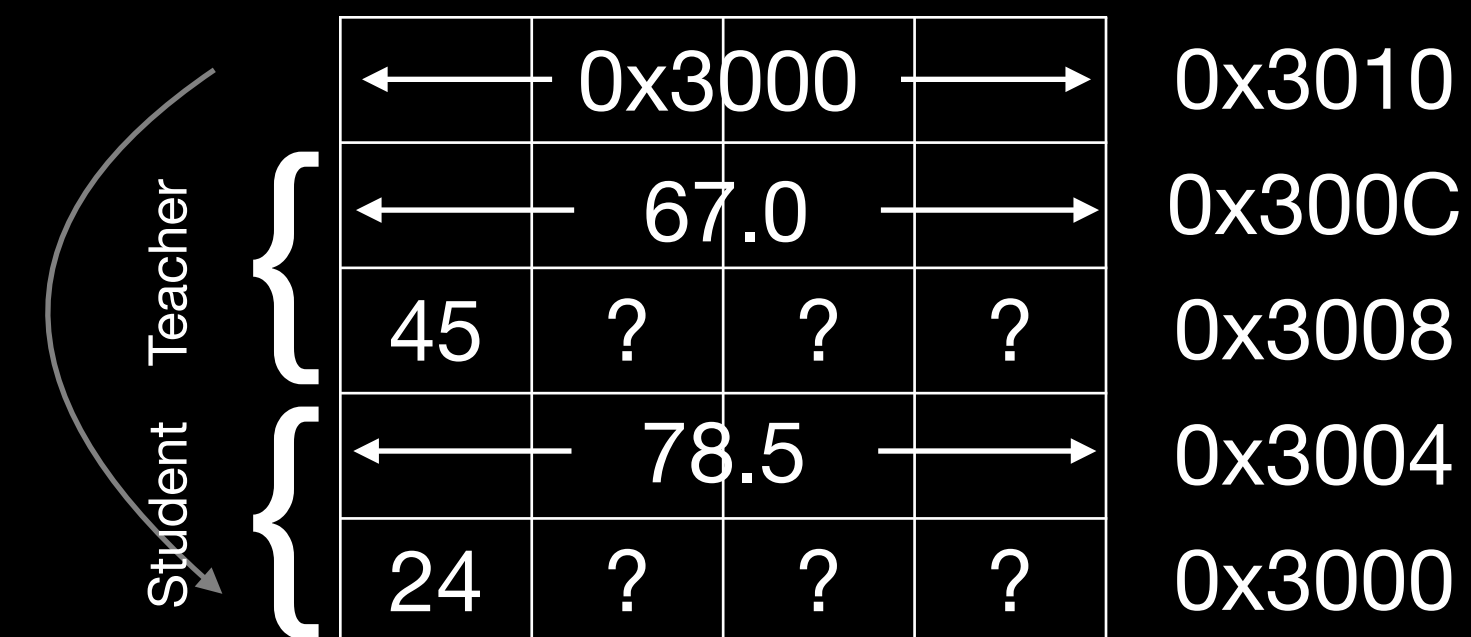


# Struct

“members” grouped together under one name

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

## Memory layout





# Union

“members” packed together at same memory location



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



# Union

“members” packed together at same memory location

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

## Memory layout

				0x300C
				0x3008
				0x3004
				0x3000



# Union

“members” packed together at same memory location

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

## Memory layout

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



# Union

“members” packed together at same memory location

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

## Memory layout

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



# Union

“members” packed together at same memory location

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

## Memory layout

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

# Union

“members” packed together at same memory location



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

## Memory layout

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





# Union

“members” packed together at same memory location



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

## Memory layout

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



# Union

“members” packed together at same memory location



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

## Memory layout

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

# Enum

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

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

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



# Enum class

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

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

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



# More concrete example

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

struct rectangle {
    float width;
    float height;
};

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

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



# typedef and using

## C++98



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

## C++11



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



# Language Basics

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



# Language Basics

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





# Language Basics

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



# Language Basics

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



# Language Basics

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



# Language Basics

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



# Language Basics

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



# Language Basics

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



# Language Basics

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



# Language Basics

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





# Language Basics

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



# Language Basics

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



# Language Basics

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



# 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

