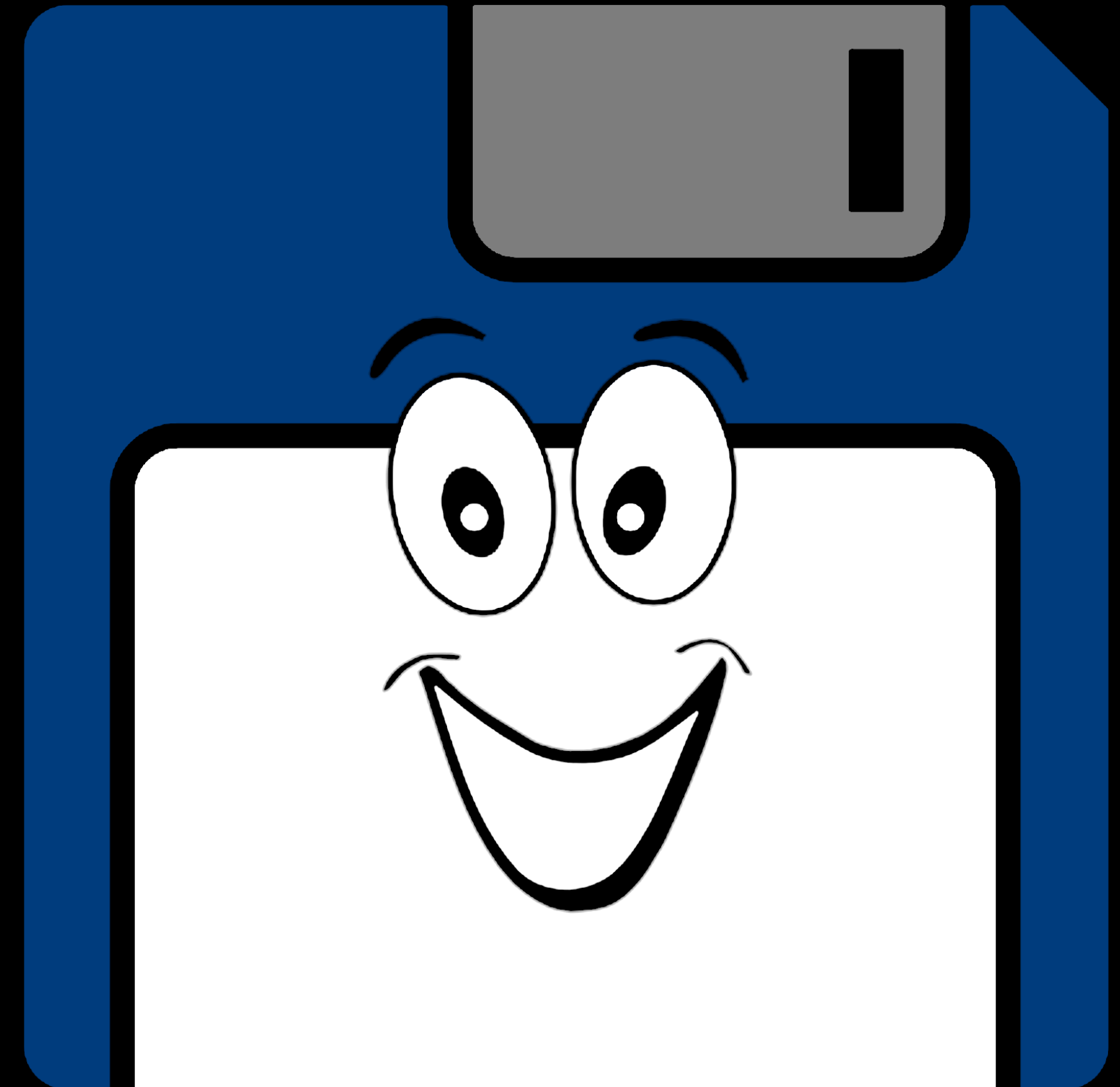# Who am I ?

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

Gammasoft

Modern C++ Course

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

Gammasoft

Modern C++ Course

The classic first application

Gammasoft

Modern C++ Course

The classic first application

program.cpp

```cpp
#include <iostream>

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

Gammasoft   Modern C++ Course

The classic first 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)
```

The classic first 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 classic first 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!
```

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;
}
```

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

Gammasoft

Modern C++ Course

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

```cpp
// 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

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

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

```cpp
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

```cpp
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
```

Gammasoft   Modern C++ Course

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

```cpp
#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

```cpp
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

```cpp
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

```cpp
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

```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;
```

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

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

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

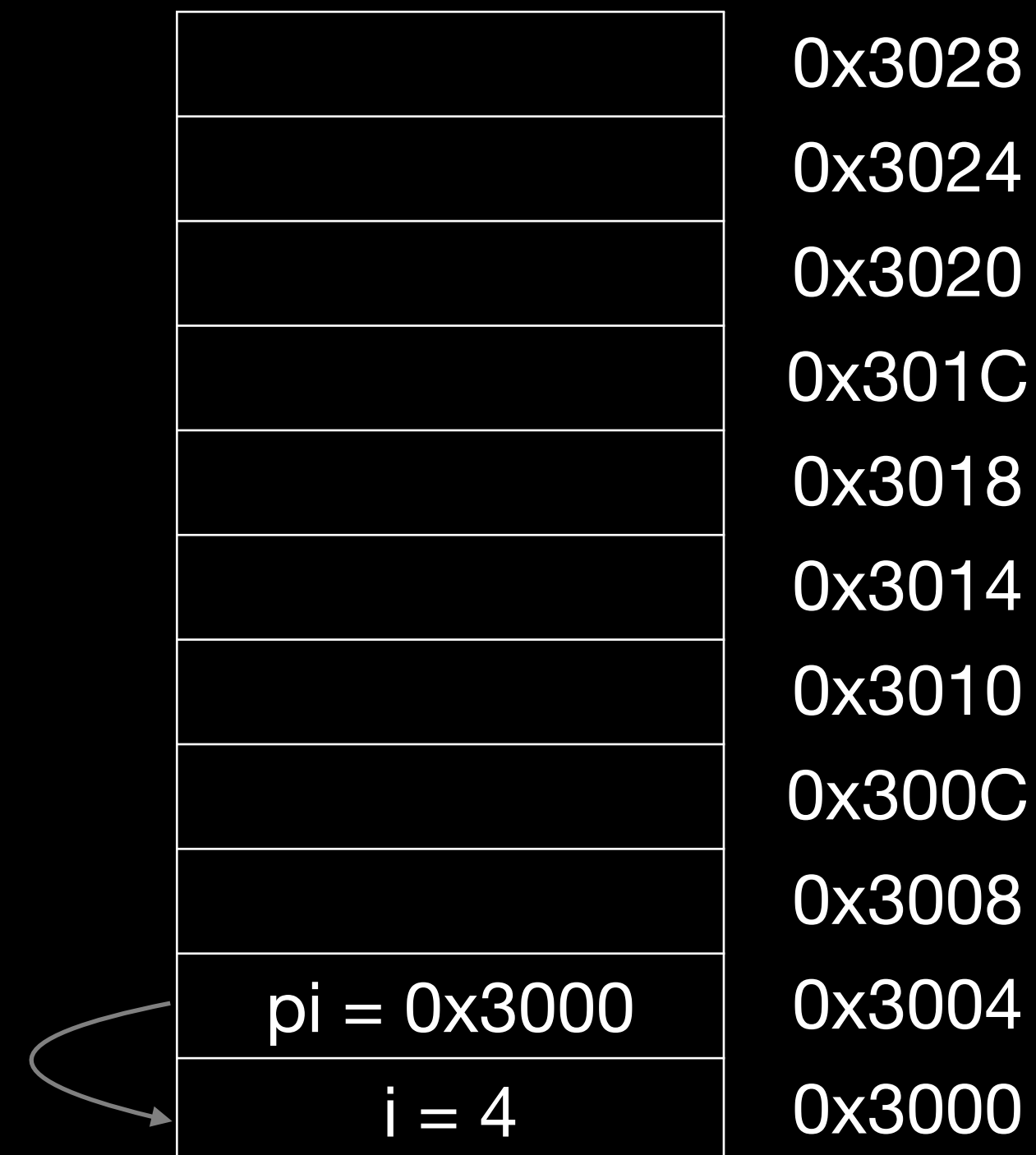|	
	0x3028
	0x3024
	0x3020
	0x301C
	0x3018
	0x3014
	0x3010
	0x300C
	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

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

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

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

	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

```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

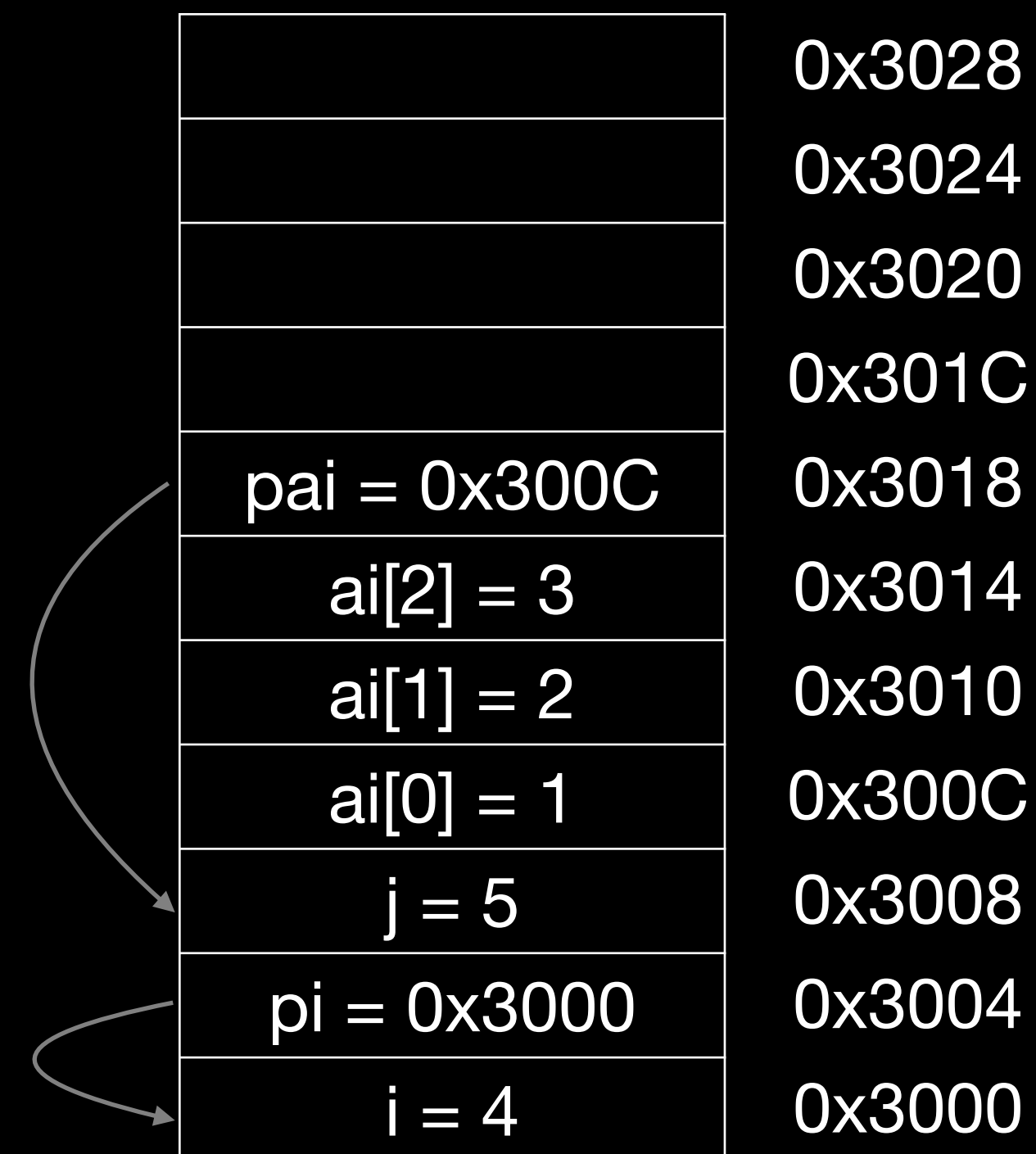|	
	0x3028
	0x3024
	0x3020
	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

Pointers

```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

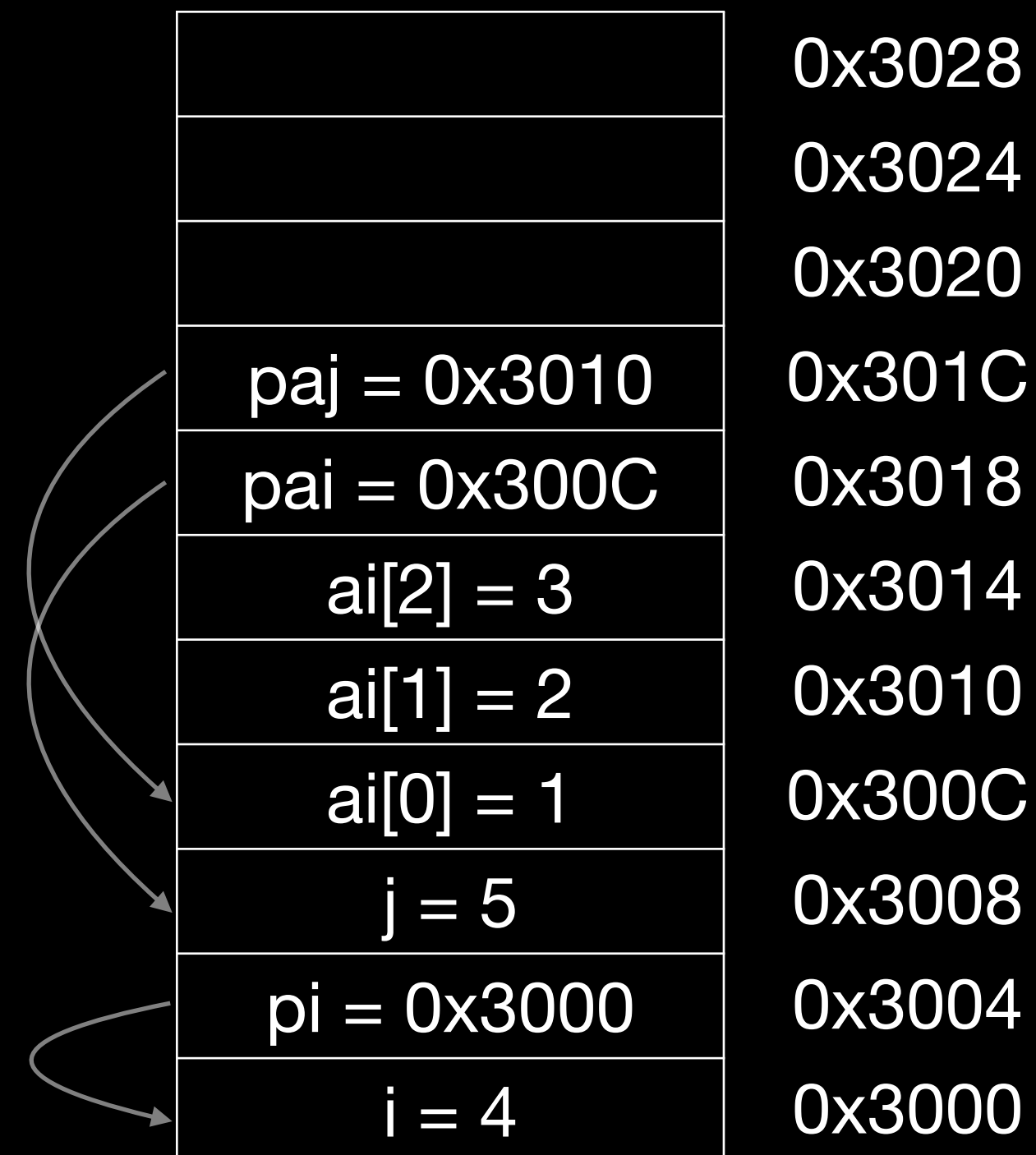|	
	0x3028
	0x3024
	0x3020
paj = 0x3010	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

Pointers

```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

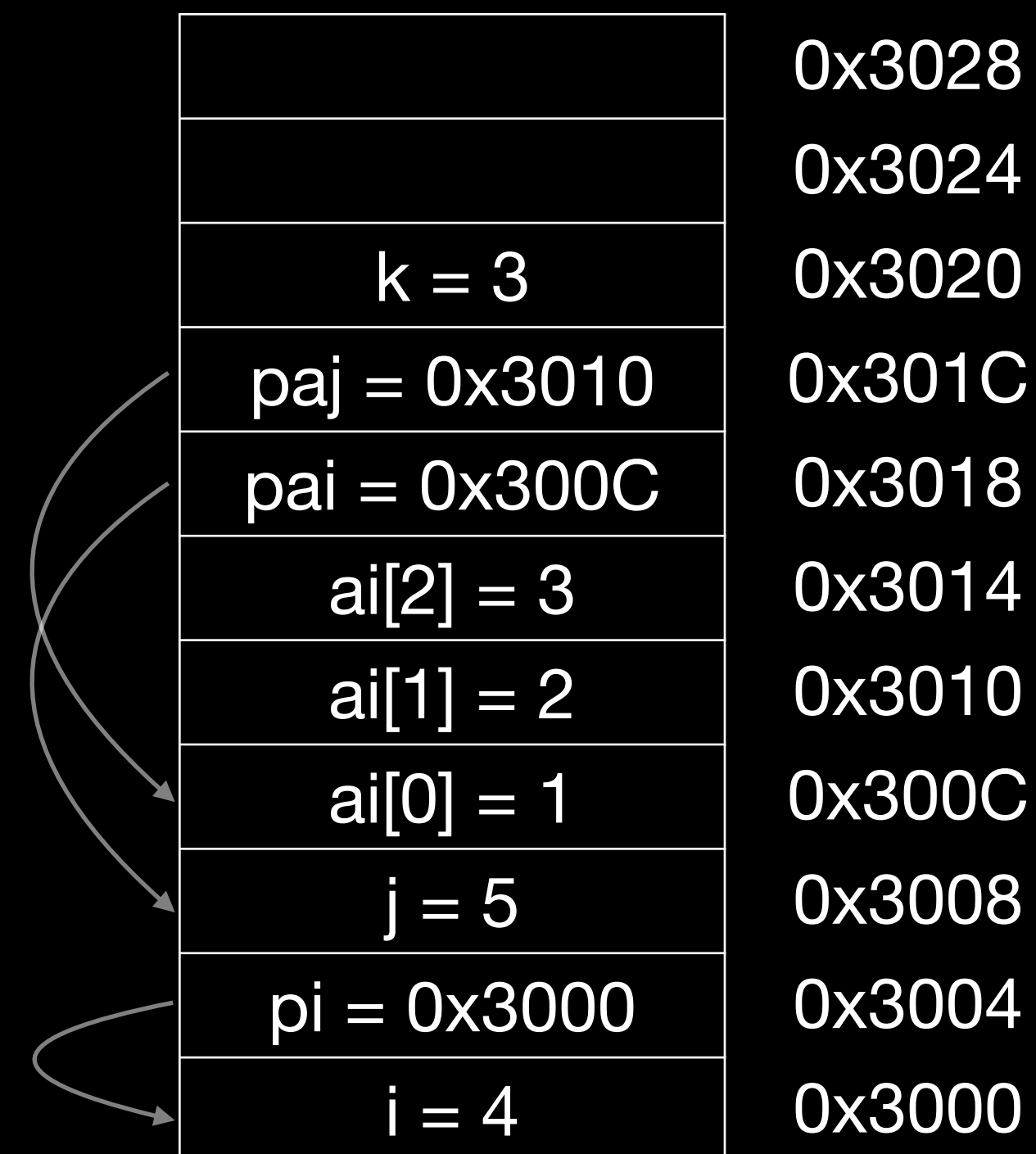|	
	0x3028
	0x3024
k = 3	0x3020
paj = 0x3010	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

Pointers

```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;
```
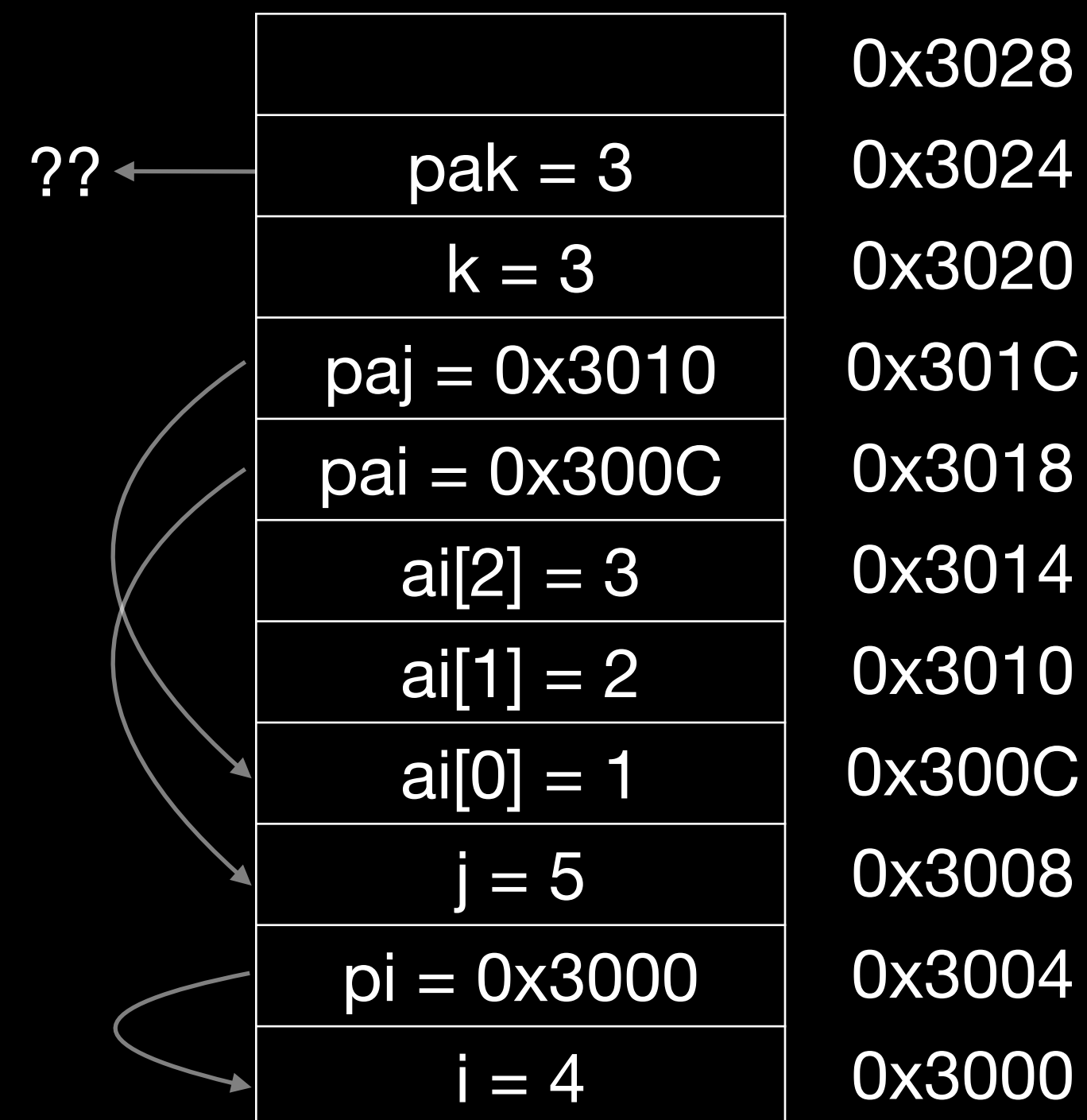
Memory layout

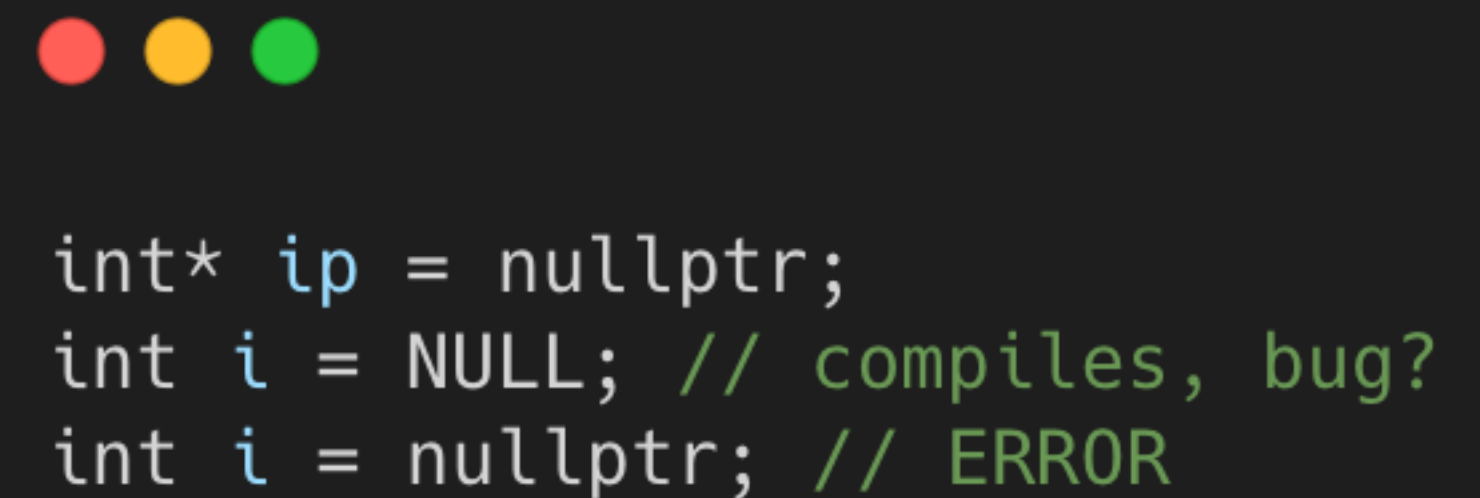|	
	0x3028
pak = 3	0x3024
k = 3	0x3020
paj = 0x3010	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

??

nullptr

- if a pointer doesn't point to anything, set it to nullptr

  - useful to e.g. mark the end of a linked data structure

  - or absence of an optional function argument (pointer)

- same as setting it to 0 or NULL (before C++ 11)

- triggers compilation error when assigned to integer

Gammasoft   Modern C++ Course

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
```

Gammasoft

Modern C++ Course

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

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

Gammasoft

Modern C++ Course

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

Gammasoft

Modern C++ Course

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

Gammasoft

Modern C++ Course

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

Gammasoft

Modern C++ Course

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

Gammasoft   59   Modern C++ Course

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

Gammasoft

60

Modern C++ Course

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

Gammasoft 63 **Modern C++ Course**

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

Gammasoft

Modern C++ Course

End