Modern C++ Programming

5. Basic Concepts IV

Memory Concepts

Federico Busato

2025-01-19

Table of Contents

1 **Pointers**

- Pointer Operations
- Address-of operator `&`
- `struct` Member Access
- `void` Pointer
- Pointer Conversion
- Pointer Arithmetic
- Wild and Dangling Pointers

2 **Fixed-Size Arrays**

Table of Contents

3 References

4 Heap and Stack

- Stack Memory
- `new`, `delete`
- Non-Allocating Placement Allocation ★
- Non-Throwing Allocation ★
- Memory Leak

Table of Contents

5 **Initialization**

- Variable Initialization
- Uniform Initialization
- Array Initialization
- Structure Initialization
- Structure Binding
- Dynamic Memory Initialization

6 `const` and Constant Expressions

- Constants and Literals
- `const`
- `constexpr`
- `constexpr` Variables
- `constexpr` Functions
- `constexpr` Objects ★
- `consteval`
- `constinit`
- `if constexpr`
- `std::is_constant_evaluated()`

Table of Contents

7 `volatile` **Keyword** ⋆

8 **Explicit Type Conversion**

- `static_cast`
- `const_cast`
- `reinterpret_cast`
- Type Punning
- `std::bit_cast`
- Uniform Initialization Conversion
- `gls::narrow_cast` ⋆

9 **sizeof and alignof Operators**

- sizeof
- alignof
- sizeof/alignof operator applied to struct
- [[no_unique_address]] ★

Pointers

Pointer

A **pointer** `T*` is a value referring to a location in memory

Pointer Dereferencing

Pointer **dereferencing** (`*ptr`) means obtaining the value stored in at the location referred to the pointer

Subscript Operator []

The subscript operator (`ptr[]`) allows accessing to the pointer element at a given position

The **type of a pointer** (e.g. `void*`) is an *unsigned* integer of 32-bit/64-bit depending on the underlying architecture

- It only supports the operators `+, -, ++, --`, comparisons `==, !=, <, <=, >, >=`, subscript `[]`, and dereferencing `*`

- A pointer can be *explicitly* converted to an integer type

```
void*  x;
size_t y = (size_t) x; // ok (explicit conversion)
// size_t y = x;        // compile error (implicit conversion)
```

Dereferencing:

```
int* ptr1 = ...;
*ptr1     = 4;     // dereferencing (assignment)
int a     = *ptr1; // dereferencing (get value)
```

Array subscript:

```
int* ptr2 = ...;
ptr2[2]   = 3;
int var   = ptr2[4];
```

Common error:

```
int *ptr1, ptr2;  // one pointer and one integer!!
int *ptr1, *ptr2; // ok, two pointers
```

Address-of operator &

The **address-of operator** (&) returns the address of a variable

```
int  a = 3;
int* b = &a; // address-of operator,
             // 'b' is equal to the address of 'a'
a++;
cout << *b;  // print 4;
```

To not confuse with the **reference syntax:** `T& var = ...`

`struct` Member Access

- The dot (`.`) operator is applied to local objects and references (see next slides)
- The arrow operator (`->`) is used with a pointer to an object

```
struct A {
    int x;
};

A a;          // local object
a.x;          // dot syntax

A* ptr = &a; // pointer
ptr->x;       // arrow syntax: same of (*ptr).x
```

`void` Pointer - Generic Pointer

Instead of declaring different types of pointer variable it is possible to declare single pointer variable which can act as any pointer types

- `void*` can be compared
- Common pointer operations are not allowed because there is no specific type pointed to

```
cout << (sizeof(void*) == sizeof(int*));  // print true

int   array[] = { 2, 3, 4 };
void* ptr;
cout << (array == ptr);
// *ptr;                 // compile error
// ptr + 2;              // compile error
```

Pointer Conversion

- Any pointer type can be *implicitly* converted to `void*`
- The opposite is not true. Non-`void` pointers must be explicitly converted
- `static_cast` (see next slides) does not allow pointer conversion for safety reasons, except for `void*`

```
int*  ptr1 = ...;
void* ptr2 = ptr1;         // int* -> void*, implicit conversion

void* ptr3 = ...;
int*  ptr4 = (int*) ptr3;  // void* -> int, explicit conversion required
                           // static_cast allowed
int*  ptr5 = ...;
char* ptr6 = (char*) ptr5; // int* -> char*, explicit conversion required,
                           // static_cast not allowed, dangerous
```

Subscript operator meaning:

`ptr[i]` is equal to `*(ptr + i)`

Note: subscript operator also accepts negative values

Pointer arithmetic rule:

`address(ptr + i) = address(ptr) + (sizeof(T) * i)`

where `T` is the type of elements pointed by `ptr`

```
int array[4] = {1, 2, 3, 4};
cout << array[1];       // print 2
cout << *(array + 1); // print 2
cout << array;          // print 0xFFFAFFF2
cout << array + 1;     // print 0xFFFAFFF6!!
int* ptr = array + 2;
cout << ptr[-1];        // print 2
```

`char arr[4] = "abc"`

value	address	
'a'	0x0	←arr[0]
'b'	0x1	←arr[1]
'c'	0x2	←arr[2]
'\0'	0x3	←arr[3]

`int arr[3] = {4,5,6}`

value	address	
4	0x0	←arr[0]
	0x1	
	0x2	
	0x3	
5	0x4	←arr[1]
	0x5	
	0x6	
	0x7	
6	0x8	←arr[2]
	0x9	
	0x10	
	0x11	

Pointer Arithmetic - Undefined Behavior ★

`lib/vsprintf.c` of the Linux kernel

```
int vsnprintf(char *buf, size_t size, ...) {
    char *end;
    /* Reject out-of-range values early
       Large positive sizes are used for unknown buffer sizes */
    if (WARN_ON_ONCE((int) size < 0))
        return 0;
    end = buf + size;
    /* Make sure end is always >= buf */
    if (end < buf) { ... } // Even if pointers are represented with unsigned values,
    ...                    // pointer overflow is undefined behavior.
                           // Both GCC and Clang will simplify the overflow check
                           // buf + size < buf to size < 0 by eliminating
}                          // the common term buf
```

Wild and Dangling Pointers

A **wild pointer** is a pointer not initialized

```
int* ptr;   // wild pointer
```

A **dangling pointer** points to a deallocated memory region

```
int* array = new int[10];
delete[] array; // ok -> "array" now is a dangling pointer
*array;         // Potential segmentation fault
delete[] array; // double free or corruption!!
```

Fixed-Size Arrays

Fixed-Size Arrays

A **fixed-size raw array** is a fundamental C/C++ data structure that stores a sequence of elements of the *same type* in *contiguous memory*. Contrary to other more advanced data structures, it is part of the language.

An array can be declared with *explicit* or *implicit* size:

```
int array[3] = {1, 2, 3}; // size=3
int array[]  = {1, 2};    // size=2
```

Special cases:

```
int size = 5;
int array[size]; // warning: run-time size is a compiler extension and
                 // potentially dangerous
// int array[0]; // compile error zero-size array is not allowed in C++
                 // (gcc allows it)
```

Number of Elements and Hierarchical Representation

The **number of elements** in an array is equal to the total byte size divided by the bytes of a single element. C++17 also provides the method `std::size()`

```
int array[5];

int num_item1 = sizeof(array) / sizeof(array[0]); // 5
int num_item2 = std::size(array); // 5, defined in many standard library headers,
                                  // e.g. <array>
```

Multidimensional arrays are represented in a *"hierarchical"* way

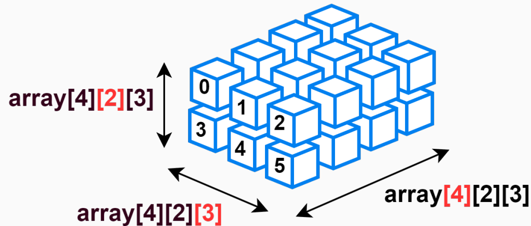
```
int array[4][2][3];
array[0][0][0]; // return a reference to a single element of type int
array[0][0];    // return a reference to an array of type int[3]
array[0];       // return a reference to an array of type int[2][3]
```

Memory Layout

C/C++ represents arrays with **row-major layout**, starting from the **index 0**.

The row-major layout is also called *right layout* because the index mapping start from the rightmost dimension (*right-to-left*).

This is in contrast to the historical layout adopted by `Fortran`, where the array representation adopts a *column-major layout* with starting index 1



Start and End Addresses

Get the **start address** of the array `int array[4][2][3]`

```
int* ptr1 = array[0][0];
int* ptr2 = &array[0][0][0]; // the meaning of & will be explained later
int* ptr3 = (int*) array;
int* ptr4 = (int*) array[0];
int* ptr5 = (int*) std::begin(array); // C++14 <iterator>
```

Get the **end address** of the array `int array[4][2][3]`

```
int* ptr1 = (int*) array + std::size(array);
int* ptr2 = (int*) std::end(array); // C++14  <iterator>
int* ptr3 = &array[3][1][3];         // bug-prone
```

References

Reference

A variable **reference** `T&` is an **alias**, namely another name for an already existing variable. Both variable and variable reference can be applied to refer the value of the variable

- A pointer has its own memory address and size on the stack, reference shares the **same memory address** (with the original variable)
- The compiler can internally implement references as *pointers*, but treats them in a very different way

References are safer than pointers:

- References **cannot have NULL** value. A reference is always connected to a legitimate storage
- References **cannot be changed**. Once a reference is initialized to an object, it cannot be changed to refer to another object
(Pointers can refer to another object at any time)
- References must be **initialized** when they are created
(Pointers can be initialized at any time)

Examples

Reference syntax: `T& var = ...`

```cpp
//int& a;      // compile error no initialization
//int& b = 3;  // compile error "3" is not a variable
int  c = 2;
int& d = c;    // reference. ok valid initialization
int& e = d;    // ok. the reference of a reference is a reference
++d;           // increment
++e;           // increment
cout << c;     // print 4
```

```cpp
int  a = 3;
int* b = &a; // pointer
int* c = &a; // pointer
++b;         // change the value of the pointer 'b'
++*c;        // change the value of 'a' (a = 4)
int& d = a;  // reference
++d;         // change the value of 'a'  (a = 5)
```

Reference vs. pointer arguments:

```
void f(int* value) {} // value may be a nullptr

void g(int& value) {} // value is never a nullptr

int a = 3;
f(&a);    // ok
f(0);     // dangerous but it works!! (but not with other numbers)
//f(a);   // compile error "a" is not a pointer

g(a);     // ok
//g(3);   // compile error "3" is not a reference of something
//g(&a);  // compile error "&a" is not a reference
```

References can be use to indicate fixed size arrays:

```
void f(int (&array)[3]) { // accepts only arrays of size 3
    cout << sizeof(array);
}
void g(int array[]) {
    cout << sizeof(array); // any surprise?
}

int  A[3], B[4];
int* C = A;
//------------------------------------------------------
f(A);    // ok
// f(B); // compile error B has size 4
// f(C); // compile error C is a pointer
g(A);   // ok
g(B);   // ok
g(C);   // ok
```

Reference - Arrays ★

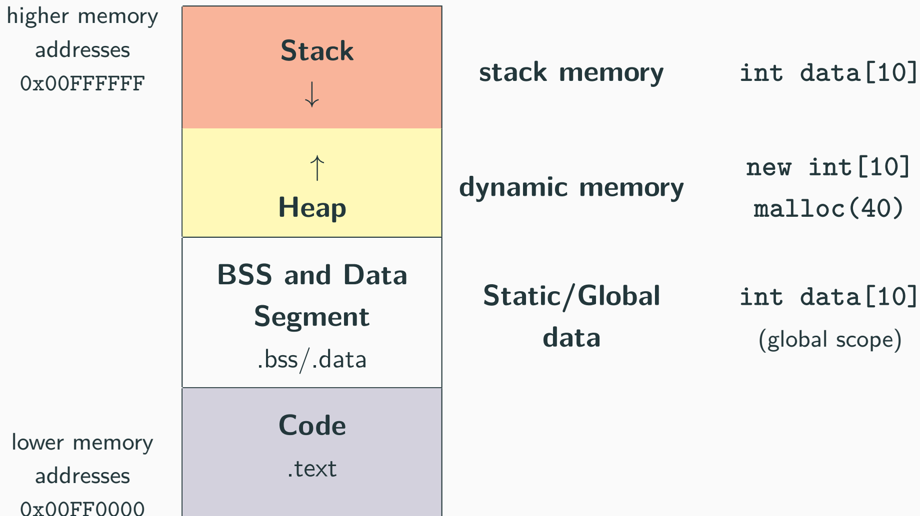
```cpp
int A[4];
int (&B)[4] = A;      // ok, reference to array
int C[10][3];
int (&D)[10][3] = C; // ok, reference to 2D array

auto c = new int[3][4];  // type is int (*)[4]
// read as "pointer to arrays of 4 int"
// int (&d)[3][4] = c;    // compile error
// int (*e)[3]     = c;    // compile error
int (*f)[4] = c;          // ok
```

see also www3.ntu.edu.sg/home/

Heap and Stack

Process Address Space



Data and BSS Segment

```
int data[]              = {1, 2}; // DATA segment memory
int big_data[1000000] = {};      // BSS segment memory
// (zero-initialized)

int main() {
    int A[] = {1, 2, 3}; // stack memory
}
```

Data/BSS (Block Started by Symbol) segments are larger than stack memory (max $\approx$ 1GB in general) but slower

Stack and Heap Memory Overview

	Stack	Heap
Memory Organization	Contiguous (LIFO)	Contiguous within an allocation, Fragmented between allocations (relies on virtual memory)
Max size	Small (8MB on Linux, 1MB on Windows)	Whole system memory
If exceed	Program crash at function entry (hard to debug)	Exception or `nullptr`
Allocation	Compile-time	Run-time
Locality	High	Low
Thread View	Each thread has its own stack	Shared among threads

Stack Memory

A local variable is either in the stack memory or CPU registers

```
int x = 3; // not on the stack (data segment)

struct A {
    int k; // depends on where the instance of A is
};
int main() {
    int   y   = 3;          // on stack
    char  z[] = "abc";      // on stack
    A     a;                // on stack (also k)
    void* ptr = malloc(4); // variable "ptr" is on the stack
}
```

The organization of the stack memory enables much higher performance. On the other hand, this memory space is limited!!

Stack Memory Data

Types of data stored in the stack:

Local variables	Variable in a local scope
Function arguments	Data passed from caller to a function
Return addresses	Data passed from a function to a caller
Compiler temporaries	Compiler specific instructions
Interrupt contexts	

Stack Memory

Every object which resides in the stack is not valid outside his scope!!

```
int* f() {
    int array[3] = {1, 2, 3};
    return array;
}
int* ptr = f();
cout << ptr[0]; // Illegal memory access!! ☠
```

```
void g(bool x) {
    const char* str = "abc";
    if (x) {
        char xyz[] = "xyz";
        str        = xyz;
    }
    cout << str; // if "x" is true, then Illegal memory access!! ☠
}
```

Heap Memory - `new`, `delete` Keywords

`new`, `delete`

`new`/`new[]` and `delete`/`delete[]` are C++ *keywords* that perform dynamic memory allocation/deallocation, and object construction/destruction at runtime

`malloc` and `free` are C functions and they only allocate and free *memory blocks* (expressed in bytes)

`new`, `delete` Advantages

- **Language keywords**, not functions → *safer*
- **Return type**: `new` returns exact data type, while `malloc()` returns `void*`
- **Failure**: `new` throws an *exception*, while `malloc()` returns a `NULL` pointer → *it cannot be ignored*, zero-size allocations do not need special code
- **Allocation size**: The number of bytes is calculated by the compiler with the `new` keyword, while the user must take care of manually calculate the size for `malloc()`
- **Initialization**: `new` can be used to initialize besides allocate
- **Polymorphism**: objects with `virtual` functions must be allocated with `new` to initialize the virtual table pointer

Dynamic Memory Allocation

- **Allocate a single element**

```
int* value = (int*) malloc(sizeof(int)); // C
int* value = new int;                    // C++
```

- **Allocate N elements**

```
int* array = (int*) malloc(N * sizeof(int)); // C
int* array = new int[N];                     // C++
```

- **Allocate N structures**

```
MyStruct* array = (MyStruct*) malloc(N * sizeof(MyStruct)); // C
MyStruct* array = new MyStruct[N];                          // C++
```

- **Allocate and zero-initialize N elements**

```
int* array = (int*) calloc(N, sizeof(int)); // C
int* array = new int[N]();                  // C++
```

Dynamic Memory Deallocation

- **Deallocate a single element**

```
int* value = (int*) malloc(sizeof(int)); // C
free(value);

int* value = new int;                    // C++
delete value;
```

- **Deallocate N elements**

```
int* value = (int*) malloc(N * sizeof(int)); // C
free(value);

int* value = new int[N];                     // C++
delete[] value;
```

Allocation/Deallocation Properties

Fundamental properties:

- Each object allocated with `malloc()` must be deallocated with `free()`
- Each object allocated with `new` must be deallocated with `delete`
- Each object allocated with `new[]` must be deallocated with `delete[]`
- `malloc()`, `new`, `new[]` never produce `NULL` pointer in the *success* case, except for zero-size allocations (implementation-defined)
- `free()`, `delete`, and `delete[]` applied to `NULL` / `nullptr` pointers do not produce errors

Mixing `new`, `new[]`, `malloc` with something different from their counterparts leads to *undefined behavior*

Easy on the stack - dimensions known at compile-time:

```
int A[3][4]; // C/C++ uses row-major order: move on row elements, then columns
```

Dynamic Memory 2D allocation/deallocation - dimensions known at run-time:

```
int** A = new int*[3];        // array of pointers allocation
for (int i = 0; i < 3; i++)
    A[i] = new int[4];        // inner array allocations

for (int i = 0; i < 3; i++)
    delete[] A[i];            // inner array deallocations
delete[] A;                   // array of pointers deallocation
```

Dynamic memory 2D allocation/deallocation C++11:

```
auto A = new int[3][4];     // allocate 3 objects of type int[4]
int  n = 3;                 // dynamic value
auto B = new int[n][4];     // ok
// auto C = new int[n][n]; // compile error
delete[] A;                 // same for B, C
```

A **non-allocating placement** `(ptr) type` allows to explicitly specify the memory location (previously allocated) of individual objects

```
// STACK MEMORY
char   buffer[8];
int*   x = new (buffer) int;
short* y = new (x + 1)  short[2];
// no need to deallocate x, y
```

```
// HEAP MEMORY
unsigned* buffer2 = new unsigned[2];
double*   z       = new (buffer2) double;
delete[] buffer2; // ok
// delete[] z;    // ok, but bad practice
```

Non-Allocating Placement and Objects ★ ⇝

Placement allocation of *non-trivial objects* requires to explicitly call the object destructor as the runtime is not able to detect when the object is out-of-scope

```
struct A {
    ~A() { cout << "destructor"; }
};

char buffer[10];
auto x = new (buffer) A();
// delete x; // runtime error 'x' is not a valid heap memory pointer
x->~A();     // print "destructor"
```

C++23 introduces a type safe placement allocation function
`std::start_lifetime_as()`

Non-Throwing Allocation ★

The `new` operator allows a non-throwing allocation by passing the `std::nothrow` object. It returns a `NULL` pointer instead of throwing `std::bad_alloc` exception if the memory allocation fails

```
int* array = new (std::nothrow) int[very_large_size];
```

note: `new` can return `NULL` pointer even if the allocated size is 0

`std::nothrow` doesn't mean that the allocated object(s) cannot throw an exception itself

```
struct A {
    A() { throw std::runtime_error{}; }
};
A* array = new (std::nothrow) A; // throw std::runtime_error
```

Memory Leak

A **memory leak** is a dynamically allocated entity in the heap memory that is no longer used by the program, but still maintained overall its execution

Problems:

- Illegal memory accesses → segmentation fault/wrong results
- Undefined values a their propagation→ segmentation fault/wrong results
- Additional memory consumption (potential segmentation fault)

```
int main() {
    int* array = new int[10];
    array      = nullptr; // memory leak!!
} // the memory can no longer be deallocated!!
```

Note: the memory leaks are especially difficult to detect in complex code and when objects are widely used

Dynamic Memory Allocation and OS

A program does not directly allocate memory itself but, it asks for a chuck of memory from the OS. The OS provides the memory at the granularity of *memory pages* (virtual memory), e.g. 4KB on Linux

Implication: out-of-bound accesses do not always lead to segmentation fault (lucky case). The worst case is an execution with undefined behavior

```
int* x         = new int;
int  num_iters = 4096 / sizeof(int); // 4 KB

for (int i = 0; i < num_iters; i++)
    x[i] = 1; // potential segmentation fault
```

Initialization

Variable Initialization

C++03:

```
int a1;           // default initialization (undefined value)

int a2(2);        // direct (or value) initialization
int a3(0);        // direct (or value) initialization (zero-initialization)
// int a4();      // a4 is a function

int a5 = 2;       // copy initialization
int a6 = 2u;      // copy initialization (+ implicit conversion)
int a7 = int(2);  // copy initialization
int a8 = int();   // copy initialization (zero-initialization)

int a9 = {2};     // copy list initialization, brace-initialization/braced-init-list syntax
```

Uniform Initialization

C++11 **Uniform Initialization** syntax allows to initialize different entities (variables, objects, structures, etc.) in a consistent way with brace-initialization or braced-init-list syntax:

```
int b1{2};          // direct list (or value) initialization
int b2{};           // direct list (or value) initialization (default constructor/
                    //                                       zero-initialization)
int b3 = int{};     // copy initialization (default constr./zero-initialization)
int b4 = int{4};    // copy initialization


int b5 = {};        // copy list initialization (default constr./zero-initialization)
```

Brace Initialization Advantages

The **uniform initialization** can be also used to *safely* convert arithmetic types, preventing implicit *narrowing*, i.e potential value loss. The syntax is also more concise than modern casts

```
int          b4 = -1; // ok
int          b5{-1};  // ok
unsigned     b6 = -1; // ok
//unsigned b7{-1};   // compile error

float    f1{10e30};   // ok
float    f2 = 10e40; // ok, "inf" value
//float f3{10e40};   // compile error
```

Array Initialization

Arrays are *aggregate* types and can be initialized with brace-initialization syntax, also called braced-init-list or aggregate-initialization

One dimension:

```
int  a[3] = {1, 2, 3}; // explicit size
int  b[]  = {1, 2, 3}; // implicit size
char c[]  = "abcd";    // implicit size
int  d[3] = {1, 2};    // d[2] = 0 -> zero/default value

int  e[4] = {0};       // all values are initialized to 0
int  f[3] = {};        // all values are initialized to 0 (C++11)
int  g[3] {};          // all values are initialized to 0 (C++11)
```

Two dimensions:

```
int a[][2] = { {1,2}, {3,4}, {5,6} }; // ok
int b[][2] = { 1, 2, 3, 4 };          // ok
// the type of "a" and "b" is an array of type int[]

// int c[][]   = ...;                 // compile error
// int d[2][] = ...;                  // compile error
```

Structures are also *aggregate* types and can be initialized with brace-initialization syntax, also called braced-init-list or aggregate-initialization

```
struct S {
    unsigned x;
    unsigned y;
};
S s1;           // default initialization, x,y undefined values
S s2 = {};      // copy list initialization, x,y default constr./zero-init
S s3 = {1, 2};  // copy list initialization, x=1, y=2
S s4 = {1};     // copy list initialization, x=1, y default constr./zero-init
//S s5(3, 5);   // compiler error, constructor not found

S f() {
    S s6 = {1, 2}; // verbose
    return s6;
}
```

```
struct S {
    unsigned x;
    unsigned y;
    void*    ptr;
};

S s1{};        // direct list (or value) initialization
               //     x,y,ptr default constr./zero-initialization

S s2{1, 2};    // direct list (or value) initialization
               //     x=1, y=2, ptr default constr./zero-initialization

// S s3{1, -2}; // compile error, narrowing conversion

S f() { return {3, 2}; } // non-verbose
```

C++11 **Non-Static Data Member Initialization (NSDMI)**, also called *brace or equal initialization* allows to initialize structure members in the declaration:

```
struct S1 {
    unsigned x = 3; // equal initialization
    unsigned y = 2; // equal initialization
//  auto     z = 3; // auto is not allowed for non-static member variables
};
struct S2 {
    unsigned x {3}; // brace initialization
};
//--------------------------------------------------------------------------------
S1 s1;         // call the default constructor (x=3, y=2)
S1 s2{};       // call the default constructor (x=3, y=2)
S1 s3{1, 4};   // set x=1, y=4
S2 s4;         // call the default constructor (x=3)
S2 s5{5};      // set x=5
```

Non-Static Data Member Initialization can be also used to initialize variables referring to other data members.

It is important to note that the initialization process follows the order of declarations

```
struct S1 {
    int x = 3;
    int y = x * 2; // y=6
};

struct S2 {
    int y = x * 2; // y=undefined
    int x = 3;
};

S1 s1;
S2 s2;
```

C++20 introduces the **designated initializer list**

```
struct A {
    int x, y, z;
};
A a1{1, 2, 3};                  // is the same of
A a2{.x = 1, .y = 2, .z = 3}; // designated initializer list
```

Designated initializer list can be very useful to improve code readability

```
void f1(bool a, bool b, bool c, bool d, bool e) {}
// long list of the same data type -> error-prone

struct B {
    bool a, b, c, d, e;
};                                // f2(B b)
f2({.a = true, .c = true});  // b, d, e = false
```

Structure Binding

C++17 *Structure Binding* declaration binds the specified names to elements of initializer. The variables associated with the structure binding are references

```
struct A {
    int x = 1;
    int y = 2;
} my_struct;
A f() { return A{4, 5}; }
                             // Case (1): struct
auto [x1, y1] = my_struct; // x1=1, y1=2
auto [x2, y2] = f();       // x2=4, y2=5
int  array[2] = {1,2}; // Case (2): raw arrays
auto [x3, y3] = array;     // x3=1, y3=2
x3            = 3;         // now also array[0] = 3
auto [x4, y4] = std::tuple<float, int>{3.0f, 2}; // Case (3): tuples
```

Dynamic Memory Initialization

Dynamic memory initialization applies the same rules of the object that is allocated

C++03:

```
int* a1 = new int;          // undefined
int* a2 = new int();        // zero-initialization, call "= int()"
int* a3 = new int(4);       // allocate a single value equal to 4
int* a4 = new int[4];       // allocate 4 elements with undefined values
int* a5 = new int[4]();     // allocate 4 elements zero-initialized, call "= int()"
// int* a6 = new int[4](3); // not valid
```

C++11:

```
int* b1 = new int[4]{};     // allocate 4 elements zero-initialized, call "= int{}"
int* b2 = new int[4]{1, 2}; // set first, second, zero-initialized
```

Initialization - Undefined Behavior Example ★

`lib/libc/stdlib/rand.c` of the FreeBSD libc

```
struct timeval tv;
unsigned long junk;                        // not initialized, undefined value

/* XXX left uninitialized on purpose */
gettimeofday(&tv, NULL);
srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
  // A compiler can assign any value not only to the variable,
  // but also to expressions derived from the variable

  // GCC assigns junk to a register. Clang further eliminates computation
  // derived from junk completely, and generates code that does not use
  // either gettimeofday or getpid
```

`const` and Constant Expressions

Constants and Literals

A **`constant expression`** is an expression that can be *evaluated at compile-time*

A **`literal`** is a *fixed value* that can be assigned to a *constant*

formally, *"Literals are the tokens of a C++ program that represent constant values embedded in the source code"*

Literal types:

- **Concrete values** of the scalar types `bool`, `char`, `int`, `float`, `double`, e.g. `true`, `'a'`, `3`, `2.0f`
- **String literal** of type `const char[]`, e.g `"literal"`
- `nullptr`
- User-defined literals, e.g. `2s`

`const` keyword

The `const` keyword declares an object that **never changes** value after the initialization. A **`const`** variable must be initialized when declared

A `const` variable is evaluated at compile-time value if the right expression is also evaluated at compile-time

```
int size = 3;              // 'size' is dynamic
int A[size] = {1, 2, 3}; // technically possible but, variable size stack array
                           // are considered BAD programming
const int SIZE = 3;
// SIZE = 4;               // compile error, SIZE is const
int B[SIZE] = {1, 2, 3}; // ok

const int size2 = size;  // 'size2' is dynamic
```

- `int*` → `const int*`
- `const int*` ↛ `int*`

```
void read(const int* array) {} // the values of 'array' cannot be modified

void write(int* array) {}

int*       ptr       = new int;
const int* const_ptr = new int;
read(ptr);           // ok
write(ptr);          // ok
read(const_ptr);     // ok
// write(const_ptr); // compile error
```

- `int*` pointer to `int`
  - The value of the pointer can be modified
  - The elements referred by the pointer can be modified
- `const int*` pointer to `const int`. Read as `(const int)*`
  - The value of the pointer can be modified
  - The elements referred by the pointer cannot be modified
- `int *const` const pointer to `int`
  - The value of the pointer cannot be modified
  - The elements referred by the pointer can be modified
- `const int *const` const pointer to `const int`
  - The value of the pointer cannot be modified
  - The elements referred by the pointer cannot be modified

Note: `const int*` (*West notation*) is equal to `int const*` (*East notation*)

Tip: pointer types should be read from right to left

`const` Keyword and Pointers ★

Common error: adding `const` to a pointer is <u>not</u> the same as adding `const` to a type alias of a pointer

```
using ptr_t       = int*;
using const_ptr_t = const int*;

void f1(const int* ptr) {   // read as '(const int)*'
//  ptr[0] = 0;             // not allowed: pointer to const objects
    ptr    = nullptr;       // allowed
}

void f2(const_ptr_t ptr) {} // same as before

void f3(const ptr_t ptr) {  // warning!! equal to 'int* const'
    ptr[0] = 0;             // allowed!!
//  ptr    = nullptr;       // not allowed: const pointer to modifiable objects
}
```

`constexpr` (C++11)

`constexpr` specifier declares an expression that **can** be evaluated at compile-time

- `constexpr` can improve performance and memory usage
- `constexpr` can potentially impact the compilation time

constexpr Variables

constexpr Variable

`constexpr` variables are **always** evaluated at compile-time

- `const` guarantees the value of a variable cannot change after the initialization
- `constexpr` implies `const`

```
const int v1 = 3;           // compile-time evaluation
const int v2 = v1 * 2;      // compile-time evaluation

int       a  = 3;           // "a" is dynamic
const int v3 = a;           // run-time evaluation!!

constexpr int c1 = v1;      // ok
// constexpr int c2 = v3; // compile error, "v3" is a run-time variable
```

`constexpr` Function

A `constexpr` **function** can be evaluated at compile-time as long as

- all its arguments are evaluated at compile-time
- the context of the return value requires a compile-time constant

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

constexpr int x = square(4); // compile-time evaluation, '4' is a literal
int           y = square(4); // run-time evaluation, 'y' is a run-time value
int           a = 4;
square(a);                   // run-time evaluation, 'a' is a run-time value
```

- C++11: must contain exactly one `return` statement, and no loops or `switch`
- C++14: no restrictions

A `constexpr` function is always *evaluated at run-time* if:

- contains run-time arguments with a lifetime that begins with the expression, even if the function doesn't depend on them (see `expr.const#4.7`)

```
constexpr int f(int  v) { return 3; }
constexpr int g(int& v) { return 3; }
int v = ...
f(v); // run-time evaluation
g(v); // compile-time evaluation lifetime of 'v' began outside the expression
```

- contains references to run-time global variables (*pure function*) or run-time-only functions
  - it is not always a compile error depending on code complexity and compiler
  - `-Winvalid-constexpr` can help to highlight the problem
  - C++23 doesn't allow this behavior

`constexpr` Functions - Relaxed Limitations ⋆

- cannot contain `assert()` until C++14
- cannot be a `virtual` member function or a destructor `~T` until C++20
- cannot contain or `try-catch` blocks or `asm` statements until C++20
- cannot contain `static` variables or `goto` until C++23
- cannot be a *reference* to *automatic storage* variable until C++26, P2686
- cannot be a *structure binding* until C++26, P2686
- cannot contain *exceptions* until C++26, P3068

- cannot contain *Run-Time Type Information* (RTTI)
- undefined behavior code is not allowed, e.g. `reinterpret_cast`, unsafe usage of `union`, signed integer overflow, etc.
- Assembly statement `asm`

`constexpr` Objects ★

`constexpr` objects initialize the *internal state* (set of member variables) at compile-time

```
struct A {
    int v;
    constexpr A(int v1) : v{v1 * 2} {}

    constexpr int f(int x) { return v * x; } // the compile-time evaluation of 'f'
};                                           // depends on both the internal state
                                             // 'v' and the parameter 'x'
                                             // (see next slide)

constexpr A a{3};

constexpr int  V      = 3;
constexpr auto lambda = [](int x) { return V * x; }; // C++17
```

`constexpr` Member Functions ★

`constexpr` *non-static member functions* of run-time objects cannot be used at compile-time if they contain data members or non-compile-time functions

Note: `static constexpr` *member functions* don't present this issue because they don't depend on a specific instance

```
struct A {
    int v = 3;
    constexpr int        f() const { return v; }
    static constexpr int g()       { return 3; }
};
A a1;
// constexpr int x = a1.f(); // compile error, f() is evaluated at run-time
constexpr    int y = a1.g(); // ok, same as 'A::g()'

constexpr    A   a2;
constexpr    int x = a2.f(); // ok
```

`consteval` Keyword

`consteval` (C++20)

`consteval`, or *immediate function*, guarantees compile-time evaluation.

A run-time value always produces a compile error

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

square(4);     // compile-time evaluation

int v = 4;     // "v" is at run-time
// square(v); // compile error
```

constinit Keyword

constinit (C++20)

constinit guarantees compile-time initialization of a variable. A run-time initialization value always produces a compile error

- The value of a variable can change during the execution
- `const constinit` does not imply `constexpr`, while the opposite is true

```
constexpr int square(int value) {
    return value * value;
}
constinit int v1 = square(4);    // compile-time evaluation
v1 = 3;                          // ok, v1 can change

int a = 4;                       // "v" is dynamic
// constinit int v2 = square(a); // compile error
```

`if constexpr`

`if constexpr` C++17 allows to *conditionally* compile code based on a *compile-time* predicate

The `if constexpr` statement forces the compiler to evaluate the branch at compile-time (similarly to the `#if` preprocessor)

```
auto f() {
    if constexpr (sizeof(void*) == 8)
        return "hello";                 // const char*
    else
        return 3;                       // int, never compiled
}
```

Note: Ternary (conditional) operator does not provide `constexpr` variant

if constexpr Example

```cpp
constexpr int fib(int n) {
    return (n == 0 || n == 1) ? 1 : fib(n - 1) + fib(n - 2);
}

int main() {
    if constexpr (sizeof(void*) == 8)
        return fib(5);
    else
        return fib(3);
}
```

Generated assembly code (x64 OS):

```asm
main:
    mov eax, 8
    ret
```

Advanced example: C++17 Compile-time Quick-Sort

`if constexpr` Pitfalls

`if constexpr` only works with *explicit* `if/else` statements

```
auto f1() {
    if constexpr (my_constexpr_fun() == 1)
        return 1;
//  return 2.0; compile error   // this is not part of if constexpr
}
```

`else if` branch requires `constexpr`

```
auto f2() {
    if constexpr (my_constexpr_fun() == 1)
        return 1;
    else if (my_constexpr_fun() == 2) // -> else if constexpr
//      return 2.0; compile error     // this is not part of constexpr
    else
        return 3L;
}
```

std::is_constant_evaluated()

C++20 provides `std::is_constant_evaluated()` utility to evaluate if the current function is evaluated at compile time

```
#include <type_traits> // std::is_constant_evaluated

constexpr int f(int n) {
    if (std::is_constant_evaluated())
        return 0;
    return 4;
}

f(3); // return 0

int v = 3;
f(v); // return = 4
```

`std::is_constant_evaluated()` has two problems that `if consteval` C++23 solves:

(1) Calling a `consteval` function cannot be used within a `constexpr` function if it is called with a run-time parameter

```
consteval int g(int n) { return n * 3; } // <- consteval

constexpr int f(int n) {
    if (std::is_constant_evaluated())  // it works with if consteval
        return g(n);
    return 4;
}

// f(3); compiler error
```

(2) `if constexpr (std::is_constant_evaluated())` is a bug because it is always evaluated to `true`

```
constexpr int f(int x) {
    if constexpr (std::is_constant_evaluated())  // if consteval avoids this error
        return 3;
    return 4;
}

constexpr int g(int x) {
    if consteval {
        return 3;
    }
    return 4;
}
```

`volatile` Keyword ★

`volatile` Keyword

`volatile`

`volatile` is a hint to the compiler to avoid aggressive memory optimizations involving a pointer or an object

Use cases:

- *Low-level programming*: driver development, interaction with assembly, etc. (force writing to a specific memory location)
- *Multi-thread program*: variables shared between threads/processes to communicate (don't optimize, delay variable update)
- *Benchmarking*: some operations need to not be optimized away

Note: `volatile` reads/writes can still be reordered with respect to non-volatile ones

`volatile` Keyword - Example

The following code compiled with `-O3` (full optimization) and without `volatile` could work fine

```
volatile int* ptr = new int[1];                // actual alloction size is much
int           pos = 128 * 1024 / sizeof(int); // larger, typically 128 KB
ptr[pos]          = 4;                         // ☠ segfault
```

`volatile` Deprecation

C++20 deprecates `volatile` outside single load and store operations

```
volatile int v  = 3;
auto         v1 = v + 4;  // ok, one load
v               = 4;      // ok, one store
v              += 4;      // deprecated, load + store

volatile int f() {}       // deprecated, volatile return value

void g1(volatile int) {}  // deprecated, volatile argument

void g2(volatile int*) {} // ok

struct A {
     volatile int x = 4;  // deprecated, volatile data member
};
```

Explicit Type Conversion

`static_cast` converts between types and performs compile-time (not run-time) type check

It is equivalent to the **old style cast** `(T) var` or `T(var)` for *value semantic*

```
int   a  = 6;
short b1 = (short) a;          // the compiler can issue a warning without
short b2 = short(a);           // explicit cast
short b3 = static_cast<short>(a);
long  c  = a;                  // not needed
```

`static_cast` prevents accidental/unsafe conversions between pointer types, especially across classes in a hierarchy

```
char* a  = new char[4]{1, 2, 3, 4};
int*  b  = (int*) a;              // ok
cout << b[0];                     // print 67305985, not 1!!
//int* c = static_cast<int*>(a); // compile error unsafe conversion
```

`static_cast` also prevents accidental/unsafe `const` conversions

```
const char* a = new char;
char*       b = (char*) a;              // ok
//char*     c = static_cast<char*>(a); // compile error unsafe conversion
```

`static_cast` prevents accidental/unsafe conversions between unrelated classes

```
struct A {};
struct B : A {};
struct C {};

A      a;
B      b;
auto   x1 = (A&) b;                // ok
auto   x2 = (C&) a;                // ok
auto   x3 = (C*) &a;               // ok
auto   x4 = static_cast<A&>(b);    // ok
//auto x5 = static_cast<C&>(a);    // compile error unsafe conversion
//auto x6 = static_cast<C*>(&a);   // compile error unsafe conversion
```

Note: `(T&) v` is equal to `*((T*) &v)`

`const_cast` can add or cast away (remove) constness or volatility

```
const int* ptr = new int[4];
auto       x1  = (int*) ptr ;             // ok
auto       x2  = (char*) ptr;             // ok
auto       x3  = const_cast<int*>(ptr);   // ok
//auto     x4  = const_cast<char*>(ptr);  // compile error unsafe conversion

const int        a   = 5;
const_cast<int>(a) = 3; // ok, but undefined behavior

int                        b   = 5;
const_cast<volatile int>(b) = 3; // ok
```

`reinterpret_cast` allows a subset of unsafe conversion:

- between pointers/references of different type with same constness
- between pointers and integer types

```
float b = 3.0f;                           // bits: 01000000010000000000000000000000
int   c = reinterpret_cast<int&>(b); // bits: 01000000010000000000000000000000

const int* ptr     = new int;
//reinterpret_cast<int*>(ptr);         // compile error
uintptr_t  my_int = reinterpret_cast<uintptr_t>(ptr); // ok

// ARRAY RESHAPING
int a[3][4];
int (&b)[2][6] = reinterpret_cast<int (&)[2][6]>(a);
int (*c)[6]    = reinterpret_cast<int (*)[6]>(a);
```

Pointer Aliasing

One pointer **aliases** another when they both point to the same memory location

Type Punning

Type punning refers to circumvent the type system of a programming language to achieve an effect that would be difficult or impossible to achieve within the bounds of the formal language

The compiler assumes that the ***strict aliasing rule*** *is never violated*: Accessing a value using a type which is different from the original one is not allowed and it is classified as *undefined behavior*

```cpp
// slow without optimizations. The branch breaks the CPU instruction pipeline
float abs(float x) {
    return (x < 0.0f) ? -x : x;
}
// optimized with bitwise operation
float abs(float x) {
    unsigned uvalue = reinterpret_cast<unsigned&>(x);
    unsigned tmp    = uvalue & 0x7FFFFFFF; // clear the last bit
    return reinterpret_cast<float&>(tmp);
}
// this is undefined behavior!!
```

GCC warning (not clang): `-Wstrict-aliasing`

- blog.qt.io/blog/2011/06/10/type-punning-and-strict-aliasing
- What is the Strict Aliasing Rule and Why do we care?
- Type Punning In C++17

std::bit_cast

The right way to avoid undefined behavior is by using `memcpy`

```
#include <cstring> // std::memcpy
float    v1 = 32.3f;
unsigned v2;
std::memcpy(&v2, &v1, sizeof(float));
```

Problems: `memcpy` is unsafe if the variables have not the same size or are not *trivially copyable*. Also, it doesn't work at compile-time (`constexpr`)

C++20 `std::bit_cast` provides a safe alternative to `reinterpret_cast` and `memcpy` that also works at compile-time

```
#include <bit> // std::bit_cast
constexpr float    v1 = 32.3f;
constexpr unsigned v2 = std::bit_cast<unsigned>(v1);
```

Uniform Initialization Conversion

A **narrowing conversion** occurs when the destination type may not be able to represent all the values of the source type

Brace initialization `{}` C++11 disallows narrowing conversions

```
// RUN-TIME VALUES
int          a = 3;
long long    x1{a};  // ok
//unsigned   x2{a};  // compile error, 'a' could be negative
//float      x3{a};  // compile error, 'a' could not be representable with float

double       b = 3;
//long long x4{b};  // compile error, 'b' could be a number with decimals
//float      x5{b};  // compile error, 'b' could not be representable with float
```

gcc issues a warning instead of a compile error for run-time narrowing conversions

Uniform Initialization Conversion

```
// COMPILE-TIME VALUES
constexpr int c = 3;
unsigned      x6{c};   // ok

constexpr int d = -1;
unsigned      x7{d};   // compile error, 'd' is negative

constexpr float e = 4;
//int             x8{e}; // compile error, 'float' cannot be narrowed to 'int'

constexpr double f = std::numbers::pi_v<double>; // π, C++20 <numbers>
float            x9{f};                          // ok
constexpr double g = 1e+40;
//float           x10{g}; // compile error, too large for 'float'
```

gls::narrow_cast ★

The `Guidelines Support Library (GSL)` contains functions and types that are suggested for use by the `C++ Core Guidelines` maintained by the Standard C++ Foundation

GLS offers `narrow_cast` operation for specifying that narrowing is acceptable and a `narrow` ("narrow if") that throws an exception if a narrowing would throw away legal values

```
#include <gsl/gsl>

double a  = 1.1;
int    x1 = gsl::narrow_cast<int>(d); // ok, explicit narrowing: 'a' becomes 1
int    x2 = gsl::narrow<int>(d);      // ok, throws 'narrowing_error'
```

`sizeof` and `alignof` Operators

`sizeof`

The `sizeof` is a compile-time operator (keyword) that determines the size, in bytes, of a *variable* or *data type*

Basic properties:

- `sizeof` returns a value of type `size_t`
- `sizeof(anything)` never returns 0 (*)
- `sizeof(char/signed char/unsigned char)` always returns 1
- `sizeof(incomplete type)` produces compile error, e.g. `void`

---

* `gcc` allows array of size 0 (not allowed by the C++ standard)

`sizeof` of *fundamental types* is simply the number of bytes defined by the c++ data model

`sizeof` applied to *compound types* (non fundamental types):

- `Pointer` number of bytes defined by the c++ data model
- `Reference` size of the referenced type
- `Array` size of a single element type multiplied by the number of elements in the array
- `struct/class` sum of the data member sizes + internal padding (*alignment*)

sizeof - Pointer

```
sizeof(int);    // 4 bytes
sizeof(int*)    // 8 bytes on a 64-bit OS
sizeof(void*)   // 8 bytes on a 64-bit OS
sizeof(size_t)  // 8 bytes on a 64-bit OS
```

```
int f(int array[]) {        // dangerous!!
    cout << sizeof(array);
}

int  array1[10];
int* array2 = new int[10];
cout << sizeof(array1); // sizeof(int) * 10 = 40 bytes
cout << sizeof(array2); // sizeof(int*) = 8 bytes
f(array1);              // 8 bytes (64-bit OS)
```

sizeof - Reference and Array

```
char  a;
char& b = a;
sizeof(&a);      // 8 bytes in a 64-bit OS (pointer)
sizeof(b);       // 1 byte, equal to sizeof(char)
                 // NOTE: a reference is not a pointer
struct S1 {
    void* p;
};
sizeof(S1);      // 8 bytes

struct S2 {
    char& c;
};
sizeof(S2);      // 8 bytes, same as sizeof(void*)
sizeof(S2{}.c); // 1 byte
```

sizeof - Special Cases

```cpp
struct A {};
sizeof(A);       // 1 : sizeof never return 0

A array1[10];
sizeof(array1); // 10 : array of empty structures

int array2[0];  // C++ doesn't allow array of size 0, as opposed to C
                // only gcc, compiler error for other compilers
sizeof(array2); // 0 : special case
```

`alignof` (C++11)

The `alignof` is a compile-time operator (keyword) that determines the alignment requirements, in bytes, of a *variable* or *data type*

Basic properties:

- `/*memory address*/std::addressof(var) % alignof(var) == 0`
- `alignof` returns a value of type `size_t`
- `alignof` always returns a power of two
- `alignof(anything)` never returns 0
- `alignof(char)` always returns 1
- `alignof(incomplete type)` produces compile error, e.g. `void`

`alignof` operator 2/2

`alignof` of *fundamental types* is simply the number of bytes defined by the c++ data model

`alignof` applied to *compound types* (non fundamental types):

`Pointer`	number of bytes defined by the c++ data model
`Reference`	alignment of the referenced type
`Array`	alignment of a single element type
`struct/class`	the largest alignment among data members

sizeof/alignof - struct

```
struct A {
    int  x; // alignment: 4, size: 4, offset: 0
    char y; // alignment: 1, size: 1, offset: 4 (= sizeof(x))
};
A a;        // alignment: 4, size: 8 -> next_multiple(sizeof(x) + sizeof(y), alignof(x))

struct B {
    char x; // alignment: 1, size: 1, offset: 0
    int  y; // alignment: 4, size: 4, offset: 4 -> next_multiple(sizeof(x), alignof(y))
};
B b;        // alignment: 4, size: 8 (same of A)

struct C {
    short x; // alignment: 2, size: 2, offset: 0
    int   y; // alignment: 4, size: 4, offset: 4 -> next_multiple(sizeof(x), alignof(y))
    char  z; // alignment: 1, size: 1, offset: 8
};
C c;         // alignment: 4, size: 12 -> next_multiple(sizeof({x,y}) + sizeof(z), alignof(y))
```

`[[no_unique_address]]` ★

C++20 `[[no_unique_address]]` allows a structure member to be overlapped with other data members of a different type

```
struct Empty {}; // empty class, sizeof(Empty) == 1

struct A {        // sizeof(A) == 8 (4 + 1 + 3 for padding)
    int   i;
    Empty e;
};
struct B {        // sizeof(B) == 4, 'e' overlaps with 'i'
    int i;
    [[no_unique_address]] Empty e;
};
```

Notes: `[[no_unique_address]]` is ignored by MSVC even in C++20 mode; instead, `[[msvc::no_unique_address]]` is provided

`sizeof` and Size of a Byte

Interesting: C++ does not explicitly define the size of a byte (see `Exotic architectures the standards committees care about`)