

Modern C++ Programming

3. BASIC CONCEPTS II

Federico Busato

University of Verona, Dept. of Computer Science
2019, v2.01



Agenda

- **Memory Management: Heap and Stack**

- Heap allocation
- Memory leak
- Stack memory
- Stack 2D allocation
- Default initialization
- Data/Bss memory segment

- **Pointers and References**

- Pointers
- Void pointer
- Address-of operator
- Pointer arithmetic
- Reference

- **sizeof operator**

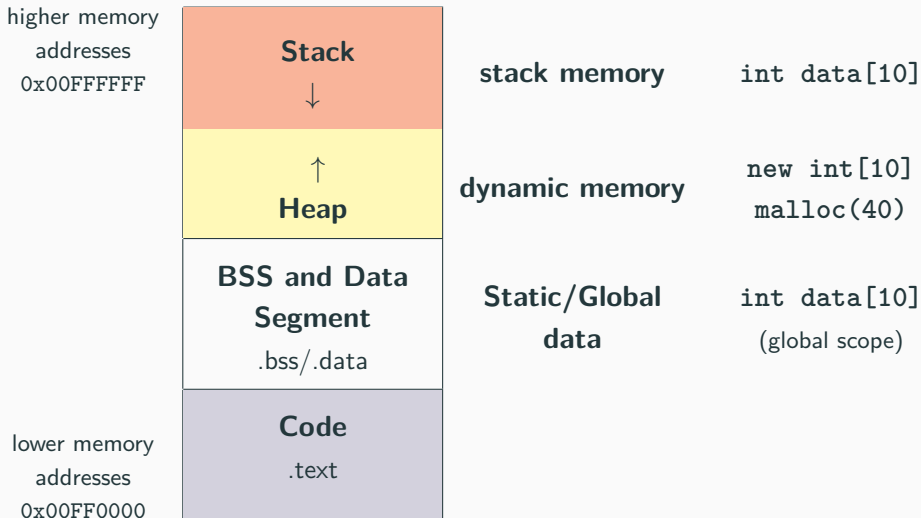
- **const, constexpr**

- **Explicit type conversion**

- Type punning
- Narrowing conversion

Memory Management: Heap and Stack

Process Address Space



new, delete

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

`malloc` and `free` are C functions and they allocate and free *memory blocks*

`new`, `delete` advantages:

- **Return type:** `new` returns exact data type, while `malloc()` returns `void*`
- **Failure:** `new` throws an *exception*, while `malloc()` returns a `NULL` pointer
- **Allocated bytes:** The size of the allocated memory is calculated by compiler for `new`, while the user must take care of manually calculate the size for `malloc()`

Dynamic 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 and zero-initialize N elements

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

- Allocate N structures

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

Dynamic 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;
```

Fundamental rules:

- Each object allocated with `new` must be deallocated with `delete`
- Each object allocated with `new[]` must be deallocated with `delete[]`

`delete` and `delete[]` applied to `NULL/nullptr` pointers do not produce errors

Memory Leak

Memory Leak

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

Problems:

- Illegal memory accesses → segmentation fault
- Undefined values → segmentation fault
- Additional memory consumption

```
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

Wild and Dangling Pointers

Wild pointer:

```
int main() {  
    int* ptr;    // wild pointer: Where will this pointer points?  
    ...        // solution: always initialize a pointer  
}
```

Dangling pointer:

```
int main() {  
    int* array = new int[10];  
    delete[] array; // ok -> "array" now is a dangling pointer  
    delete[] array; // double free or corruption!!  
    // program aborted, the value of "array" is not null  
}
```

Solution:

```
int main() {  
    int* array = new int[10];  
    delete[] array; // ok -> "array" now is a dangling pointer  
    array = nullptr; // no more dangling pointer  
    delete[] array; // ok, no side effect  
}
```

Unless it is allocated in heap memory (i.e. `new`), then it is either in stack memory or CPU registers

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

```
int* wrongFunction() {
    int A[3] = {1, 2, 3};
    return A;
}

int main() {
    int* ptr = wrongFunction();
    cout << ptr[0]; // Illegal memory access!!
}
```

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

It is $\approx 8MB$ on linux by default

```
int* ptr[10]; // array of ten integer pointers
              // read as (int*) ptr[10]

int (*ptr)[10]; // pointer to an array of ten integers
                // equal to:
                //     int a[10];
                //     int* ptr = a;
```

2D Memory Allocation

Easy on stack:

```
int A[3][4];
```

Dynamic Memory 2D allocation/free:

```
int** A = new int*[3];  
for (int i = 0; i < 3; i++)  
    A[i] = new int[4];  
  
for (int i = 0; i < 3; i++)  
    delete[] A[i];  
delete[] A;
```

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

```
auto A = new int[3][4];    // allocate 3 objects of size 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
```

Data and BSS Segment

```
int data[] = {1, 2, 3, 4}; // 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) are larger than stack memory (max \approx 1GB in general) but slower

Initialization

Stack Array Initialization

One dimension:

```
int A[3] = {1, 2, 3}; // explicit size
int B[] = {1, 2, 3}; // implicit size
char C[] = "abcd"; // implicit size
int C[3] = {1, 2}; // C[2] = 0 -> default value

int D[4] = {0}; // all values of D are initialized to 0
int E[3] = {}; // all values of E are initialized to 0 (C++11)
int F[3] {}; // all values of F are initialized to 0 (C++11)
```

Two dimensions:

```
int G[][2] = { {1,2}, {3,4}, {5,6} }; // ok
int H[][2] = { 1, 2, 3, 4 }; // ok
// the type of G and H is an array of type int[]
// int F[][] = ...; // compile error!!
// int G[2][] = ...; // compile error!!
```


Default Initialization

Rules:

- An object with **dynamic** storage duration (heap) has indeterminate value
- An object whose initializer is an **empty set of parentheses** `{ }` is zero or default initialized

Initialization

```
int a1;           // indeterminate
int* a2 = new int; // indeterminate
int* a3 = new int(); // indeterminate
int* a4 = new int(4); // allocate a single value equal to 4!!

int* b1 = new int[4](); // allocate 4 elements zero-initIALIZED
int* b2 = new int[4]{}; // indeterminate
int* b3 = new int[4]{1, 2}; // set first, second, indeterminate
                          // other values

int c1(4); // c1 = 4;
int c2 = int(); // zero-initIALIZED
int c4 { 0 }; // zero-initIALIZED
int c5 = { 0 }; // zero-initIALIZED
int c6 {}; // zero-initIALIZED

// int d3(); // d3 is a function
```

Pointers and References

Pointers and Pointer Dereferencing

Pointer

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

Pointer Dereferencing

Pointer **dereferencing** means obtaining the value stored in at the location referred to the pointer

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

Common error:

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

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

- A `void*` can be assigned to another `void*`
- `void*` can be compared for equality and inequality
- A `void*` can be explicitly converted to another type
- Other operations would be unsafe because the compiler cannot know what kind of object is really pointed to. Consequently, other operations result in compile-time errors

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

```
int array[] = { 2, 3, 4 };
```

```
void* ptr = array;
```

```
cout << *array; // print 2
```

```
// cout << *ptr; // compile error!!
```

```
cout << *((int*) ptr); // print 2
```

```
// void* ptr2 = ptr + 2; // compile error!!
```

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

```
int array[4];
// &array is a pointer to an array of size 4
int size1 = (&array)[1] - array;
int size2 = *(&array + 1) - array;
cout << size1; // print 4
cout << size2; // print 4
```

1 + 1 ≠ 2 : Pointer Arithmetic

Pointer syntax:

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

Pointer arithmetic rule:

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

where T is the type of elements pointed by ptr

Example:

```
int array[4] = {1, 2, 3, 4};
cout << array[1];      // print 2
cout << *(array + 1); // print 2
cout << array;         // print 0xFFFFFFF2
cout << array + 1;     // print 0xFFFFFFF6!!
```

```
char arr[3] = "abc"
```

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

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

value	address	
4	0x0	←arr[0]
	0x1	
	0x2	
	0x3	
5	4	←arr[1]
	0x5	
	0x6	
	0x7	
	0x8	←arr[2]

Reference

A variable **reference** 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)
- References are internally implemented as *pointer*, but the compiler treats them in a very different way

References are safer than pointers:

- References cannot have NULL value. You must always be able to assume that a reference is 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 be pointed to another object at any time)
- References must be initialized when they are created (Pointers can be initialized at any time)

Reference (Examples)

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

```
//int& d; // reference. compile error!! no initialization  
int c = 2;  
int& e = c; // reference. ok valid initialization  
e++; // increment  
cout << c; // print 3
```

```
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'  
  
int& c = a; // reference  
c++; // change the value of 'a'
```

Reference (Function Arguments)

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

References can be used to indicate fixed size arrays:

```
f(int (&array)[3]) {} // accepts only arrays of size 3
                    // f(int array[]) accepts any size

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
```

Reference (Arrays)

```
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
```

Reference:

[1] www3.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html

Reference and struct

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

```
#include <iostream>
struct A {
    int x = 3;
};

int main() {
    A obj;

    A* p = &obj;    // pointer
    p->x;           // arrow syntax

    A& ref = obj;  // reference
    cout << obj.x; // dot syntax
    cout << ref.x; // dot syntax
}
```

sizeof Operator

sizeof operator

sizeof

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

- `sizeof` returns a value of type `size_t`
- `sizeof(incomplete type)` produces compile error
- `sizeof(bitfield)` produces compile error
- `sizeof(anything)` never returns 0, except for array of size 0
- `sizeof(char)` always returns 1
- When applied to structures it also takes into account padding
- When applied to a reference, the result is the size of the referenced type

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

```
int f(int[] x) {
    cout << sizeof(x);
}

int A[10];
int* B = new int[10];
cout << sizeof(A); // print sizeof(int) * 10 = 40
cout << sizeof(B); // print sizeof(int*) = 8 (64-bit)
f(A); // print 4
```



```
struct B {
    int x;
    char y;
};

struct C : B { // C extends B
    short z;
};

sizeof(B);      // 8 : 4 + 1 (+ 3) (padding)
sizeof(C);      // 12 : sizeof(B) + 2 (+ 2) (padding)

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

```
char a;
char& b = a;
sizeof(&a);    // 8 in a 64-bit OS (pointer)
sizeof(b);    // 1 sizeof(char)

struct A {};
A array1[10];
sizeof(array1); // 1 : array of empty structures

int array2[0];
sizeof(array2); // 0

int array3[4]
sizeof(array3) // 16: 4 elements of 4 bytes
sizeof(array3) / sizeof(int); // 4 elements
```

const and constexpr

const keyword

The `const` keyword indicates objects never changing value after their initialization (they must be initialized when declared)

Compile-time value if the right expression is evaluated at compile-time

```
int size = 3;
int A[size] = {1, 2, 3}; // Technically possible (size is dynamic)
                        // But NOT approved by the C++ standard

const int SIZE = 3;
// SIZE = 4;           // compile error!! (SIZE is const)
int B[SIZE] = {1, 2, 3}; // ok

const int size2 = size;
int B[size2] = {1, 2, 3}; // BAD programming!! size is not const
// (some compilers allow variable size stack array -> dangerous!!) 30/40
```

Constness rules:

- `int* → const int*`
- `const int* ↯ int*`

```
int f1(const int* array) { // the values of the array cannot be
    ...                    // modified
}
```

```
int f2(int* array) {}
```

```
int*    ptr    = new int[3];
const int* c_ptr = new int[3];
f1(ptr);      // ok
f2(ptr);      // ok
f1(c_ptr);    // ok
// f2(c_ptr); // compile error!!
```

```
void g(const int) { // pass-by-value combined with 'const'
    ...            // note: it is not useful because the value
}                  // is copied
```

- `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*` is equal to `int const*`

Tip: pointer types should be read from right to left

constexpr

C++11/C++14/C++17 guarantees compile-time evaluation of an expression as long as all its arguments are constant

- `const` guarantees the value of a variable to be fixed overall the execution of the program
- `constexpr` tells the compiler that the expression results is at compile-time. *constexpr value implies const*
- C++11: `constexpr` must contain exactly one `return` statement and it must not contain loops or switch
- C++14: `constexpr` has no restrictions

```
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 c1 = v1;   // ok
// constexpr c2 = v3; // compile error!!
```

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

square(4); // compile-time evaluation

int a = 4; // "a" is dynamic
square(a); // run-time evaluation
```


if constexpr

C++17 introduces `if constexpr` feature which allows *conditionally* compiling code based on a *compile-time* value

It is an `if` statement where the branch is chosen at compile-time (similarly to the `#if` preprocessor)

```
void f() {  
    if constexpr (true)  
        std::cout << "compile!";  
    else  
        std::cout << "error!"; // never compiled  
}
```

constexpr example

```
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):

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

Explicit Type Conversion

Old style cast `(type) value`

New style cast:

- `static_cast` does compile-time, not run-time checking of the types involved. In many situations, this can make it the safest type of cast, as it provides the least room for accidental/unsafe conversions between various types
- `reinterpret_cast`
`reinterpret_cast<T*>(v)` equal to `(T*) v`
`reinterpret_cast<T&>(v)` equal to `*((T*) &v)`
- `const_cast` may be used to cast away (remove) constness or volatility

Static cast vs. old style cast:

```
char a[] = {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
```

Const cast:

```
const int a = 5;
const_cast<int>(a) = 3; // ok
```

Reinterpret cast: (bit-level conversion)

```
float b = 3.0f;
// bit representation of b: 01000000010000000000000000000000
int c = reinterpret_cast<int&>(b);
// bit representation of c: 01000000010000000000000000000000
int a[3][4]; // array reshaping example
int (&b)[2][6] = reinterpret_cast<int (&)[2][6]>(a);
int (*c)[6] = reinterpret_cast<int (*)[6]>(a);
```

Type punning

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

```
bool is_negativeA(float x) {  
    return x < 0.0;  
}  
bool is_negativeB(float x) {  
    unsigned int* ui = (unsigned int *) &x; // gcc warning:  
    return (*ui) & 0x80000000;           // -Wstrict-aliasing  
}
```

Narrowing Conversion

C++11 provides protection against **narrowing**, i.e. assigning a numeric value to a numeric type not capable of holding that value

```
int main() {
    int a1 = 36.6;      // ok
    // int a2 = { 36.6 }; // compile error!!
    int a3 { 36.6 };   // ok!! (constructor)

    float b1 = 36.6;   // ok
    // float b2 = { 36.6 }; // compile error!!
    int a3 { 36.6 };   // ok!! (constructor)

    char c1 = 512;     // ok
    // char c2 = { 512 }; // compile error!!
    char c3 = { 512 }; // ok!! (constructor)
}
```

- Prefer `{ }` syntax for variable initialization