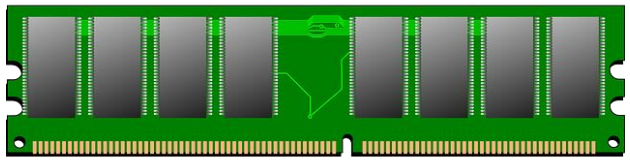
Introduction to Programming

Chapter 8

Memory Management

Slides adapted from: https://www.ipb.uni-bonn.de/teaching/cpp-2020/

Working memory or RAM

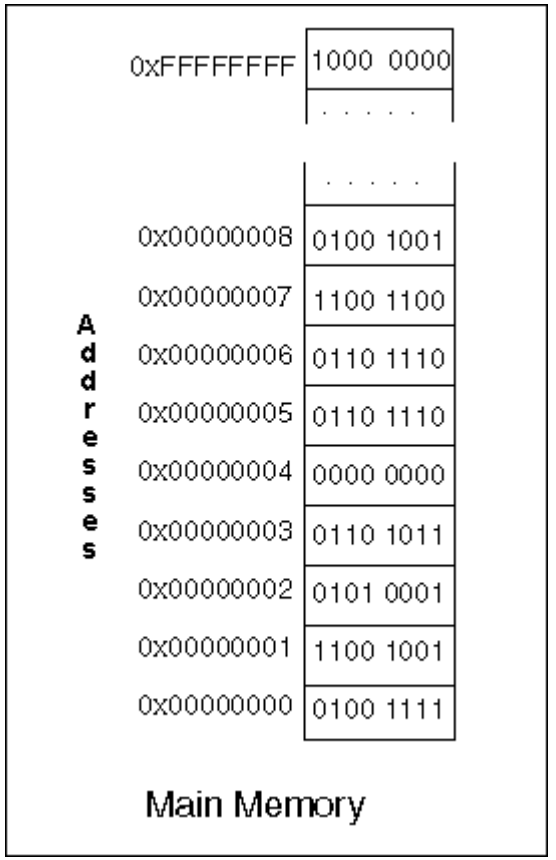


Working memory has linear addressing

Every byte has an address usually presented in hexadecimal form, e.g. `0x7fffb7335fdc`

Any address can be accessed at random

Pointer is a data type to store memory addresses



Pointer

<TYPE> * defines a pointer to type <TYPE>

The pointers have a type

Pointer <TYPE> * can point only to a variable of type <TYPE>

Uninitialized pointers point to a random address

Always initialize pointers to an address or a nullptr

Example

```
int* a = nullptr ;
double *b = nullptr ;
char * c = nullptr ;
```

https://cplusplus.com/doc/tutorial/pointers/

Address operator for pointers

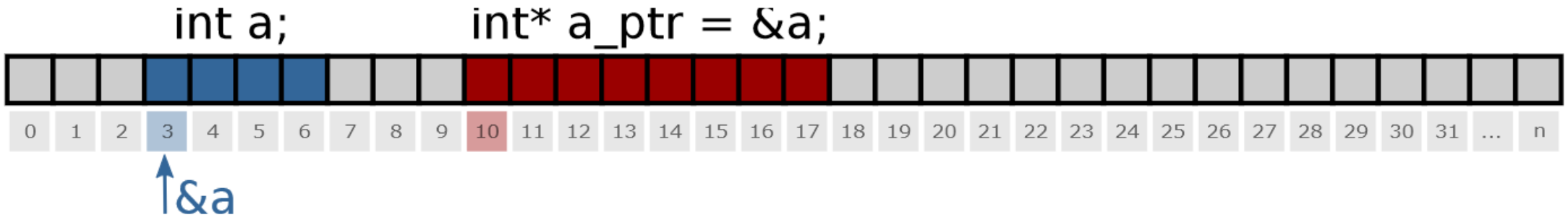
Operator & returns the address of the variable in memory

Return value type is “pointer to value type”

`sizeof(pointer)` is 8 bytes in 64-bit systems

Example

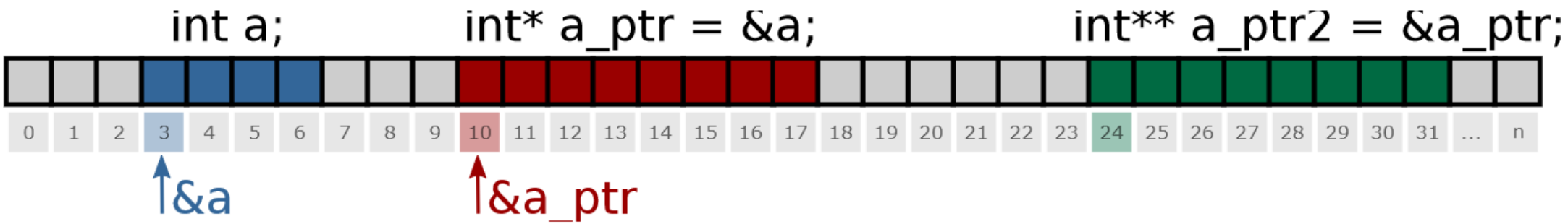
```
int a = 42;
int* a_ptr = &a;
```



Pointer to pointer

Example

```
int a = 42;
int* a_ptr = &a;
int** a_ptr_ptr = &a_ptr;
```



Pointer dereferencing

Operator `*` returns the value of the variable to which the pointer points

Dereferencing of `nullptr`: **Segmentation Fault**

Dereferencing of uninitialized pointer: **Undefined Behavior**

Pointer dereferencing

Example

```
#include <iostream>
using std::cout, std::endl;
int main() {
    int a = 42;
    int* a_ptr = &a;
    int b = *a_ptr;
    cout << "a = " << a << " b = " << b << endl;
    *a_ptr = 13;
    cout << "a = " << a << " b = " << b << endl;
    return 0;
}
```

Output

```
a = 42, b = 42
2 a = 13, b = 42
```

Uninitialized pointer

```
#include <iostream>
using std::cout, std::endl;
int main() {
    int* i_ptr; // BAD! Never leave unitialized !
    cout << "ptr address : " << i_ptr << endl;
    cout << "value under ptr: " << *i_ptr << endl;
    i_ptr = nullptr ;
    cout << "new ptr address : " << i_ptr << endl;
    cout << "ptr size: " << sizeof (i_ptr) << " bytes";
    cout << " (" << sizeof(i_ptr) * 8 << "bit) " << endl;
    return 0;
}
```

```
ptr address : 0 x400830
value under ptr: -1991643855
new ptr address : 0
ptr size: 8 bytes (64 bit)
```

Important

Always initialize with a value or a `nullptr`

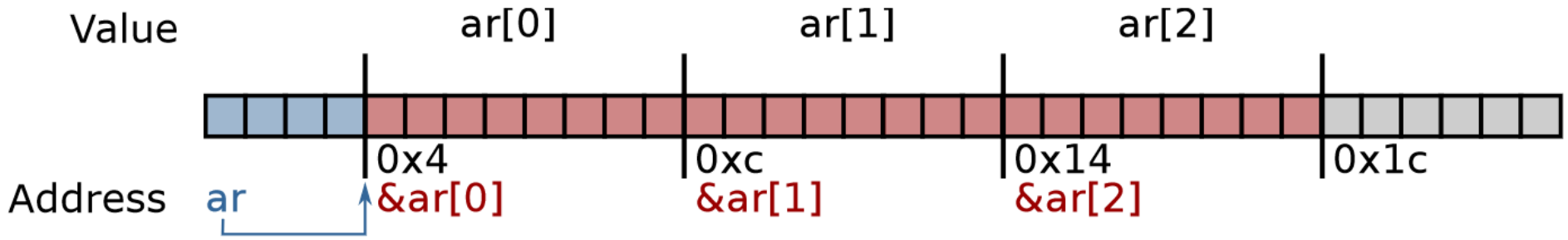
Dereferencing a `nullptr` causes a **Segmentation Fault**

Use `if` to avoid Segmentation Faults

```
if( some_ptr ) {
    // only enters if some_ptr != nullptr
}
if(! some_ptr ) {
    // only enters if some_ptr == nullptr
}
```

Arrays in memory and pointers

Recall that array elements are continuous in memory



Name of an array implicitly converted to a pointer:

```
double ar[3];
double* ar_ptr = ar;
double* ar_ptr = &ar[0];
```

Main difference

- `ar_ptr` can be assigned a different address
- `ar` can never be assigned anything

```
ar = ar_ptr;  // error
```

Careful! Out of bound!

```
#include <iostream>
int main() {
    int ar[] = {1, 2, 3};
    // WARNING ! Iterating too far!
    for (int i = 0; i < 6; i++){
        std::cout << i << ": value: " << ar[i]
                  << "\t addr:" << &ar[i] << std :: endl;
    }
    return 0;
}
```

```
0: value: 1 addr :0 x7ffd17deb4e0
1: value: 2 addr :0 x7ffd17deb4e4
2: value: 3 addr :0 x7ffd17deb4e8
3: value: 0 addr :0 x7ffd17deb4ec
4: value: 4196992 addr :0 x7ffd17deb4f0
5: value: 32764 addr :0 x7ffd17deb4f4
```

Using `const` with pointers

Pointers can point to a `const` variable:

```
// Cannot change value, can reassign pointer.
const int * const_var_ptr = &var;
const_var_ptr = &var_other;
```

Pointers can be `const`:

```
// Cannot reassign pointer, can change value.
int * const var_const_ptr = &var;
*var_const_ptr = 10;
```

Pointers can do both at the same time:

```
// Cannot change in any way, read-only.
const int * const const_var_const_ptr = &var;
```

Read from right to left to see which `const` refers to what

Memory management structures

Working memory is divided into two parts:

Automatic storage aka Stack

Dynamic storage aka Heap

Automatic storage (aka Stack)

Static memory

Available for short term storage (scope)

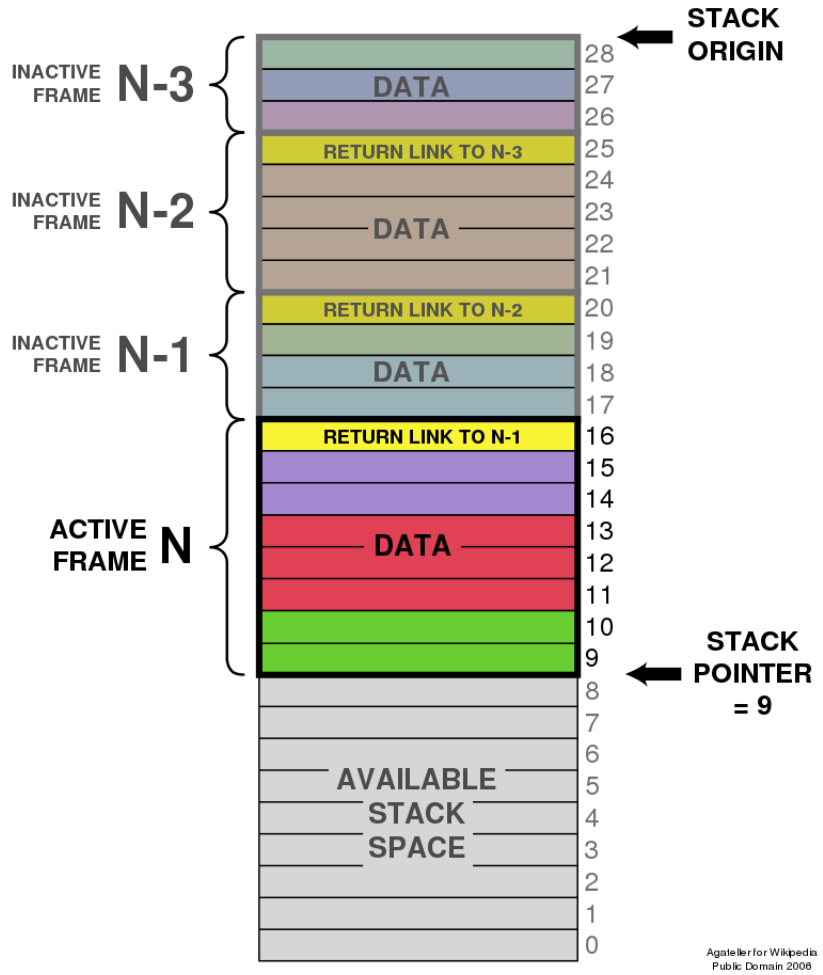
Small / limited (8 MB Linux typically)

Memory allocation is fast

LIFO (**L**ast **i**n **F**irst **o**ut) structure

- Items added to top of the stack with push
- Items removed from the top with pop

Automatic storage (aka Stack)



A Stack frame for every function call

Local variables push on frame when declared and pop when goes out of scope

```
int f() {
    int x = 313;
    {
        int ar[2];
        cin >> ar[0];
        ar[1] = 13;
        cout << ar[0];
    }
    int br[2];
    cout << br << "\n";
}
```

Stack Overflow

When large memory is requested

```
#include <iostream>

int main() {
    int stack[10'000'000];
    std::cout << "hi" << stack[0];
    // we'll use stack[0] here
    // so the compiler won’t
    // optimize the array away

    return 0;
}
```

Because of infinite recursive loop

```
#include <iostream>
int g_counter{ 0 };

void eatStack() {
    std::cout << ++g_counter << ' ';

    // to avoid compiler warnings
    if (g_counter > 0)
        eatStack();

    // to prevent tail-call optimization
    std::cout << "hi";
}

int main() {
    eatStack();
    return 0;
}
```

Dynamic storage (aka Heap)

Dynamic memory

Available for long term storage (program runtime)

Raw modifications possible with new and delete (usually done indirectly)

Allocation is slower than stack allocations

Operators new and new[ ]

User controls memory allocation (unsafe)

Use new to allocate data:

```
// pointer variable stored on stack
int* int_ptr = nullptr;
// 'new' returns a pointer to memory in heap
int_ptr = new int;

// also works for arrays
float* float_ptr = nullptr;
// 'new' returns a pointer to an array on heap
float_ptr = new float[number];
```

new returns an address of the variable on the heap

Prefer using alternates!

Operators `delete` and `delete[]`

Memory is not freed automatically!

User must remember to free the memory

Use `delete` or `delete[]` to free memory:

```
int* int_ptr = nullptr ;
int_ptr = new int;
// delete frees memory to which the pointer points
delete int_ptr ;

// also works for arrays
float* float_ptr = nullptr ;
float_ptr = new float[number ];
// make sure to use 'delete []' for arrays
delete [] float_ptr ;
```

Prefer using alternates!

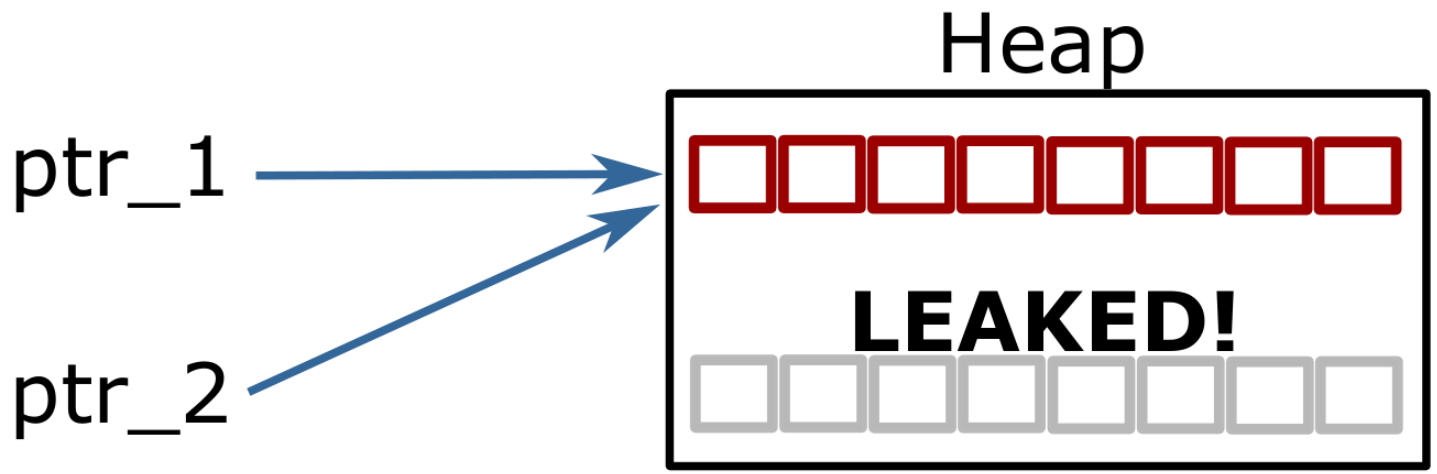
Example: heap memory

```
#include <iostream>
using std::cout, std::endl;
int main() {
    int size = 2; int* ptr = nullptr ;
    {
        ptr = new int[size ];
        ptr[0] = 42; ptr[1] = 13;
    } // End of scope does not free heap memory !
    // Correct access , variables still in memory .
    for (int i = 0; i < size; ++i) {
        cout << ptr[i] << endl;
    }
    delete [] ptr; // Free memory .
    for (int i = 0; i < size; ++i) {
        // Accessing freed memory . UNDEFINED !
        cout << ptr[i] << endl;
    }
    return 0;
}
```

Memory Leak

Can happen when working with Heap memory if we are not careful

Memory leak: memory allocated on Heap access to which has been lost



Memory leak

```
int main() {
    int *ptr_1 = nullptr;
    int *ptr_2 = nullptr;

    // Allocate memory for two ints on the heap.
    ptr_1 = new int;
    ptr_2 = new int;

    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;

    // Overwrite ptr_2 and make it point where ptr_1
    ptr_2 = ptr_1;

    // ptr_2 overwritten , no chance to access the memory.
    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;

    delete ptr_1;
    delete ptr_2;
}
```

Error: double free

```
ptr_1: 0x10a3010 , ptr_2: 0x10a3070
ptr_1: 0x10a3010 , ptr_2: 0x10a3010
*** Error: double free ***
```

The memory under address 0x10a3070 is never freed

Instead we try to free memory under 0x10a3010 twice

Freeing memory twice is an error

Tools to the rescue

Valgrind: on terminal `valgrind ./my_program`

AddressSanitizer: using compiler flag `-fsanitize=address`

Stackoverflow!

code -fsanitize=address

```
ERROR: AddressSanitizer: attempting double-free on 0x602000000010 in thread T0:
# ... more stuff
0x602000000010 is located 0 bytes inside of 4-byte
# ... even more stuff
SUMMARY: AddressSanitizer: double -free in operator delete(void*, unsigned long)
# ... even more more stuff
ABORTING
```

valgrind output

```
HEAP SUMMARY:
       in use at exit: 4 bytes in 1 blocks
     total heap usage: 4 allocs , 4 frees, 76,808 bytes allocated

LEAK SUMMARY:
      definitely lost: 4 bytes in 1 blocks
      indirectly lost: 0 bytes in 0 blocks
        possibly lost: 0 bytes in 0 blocks
      still reachable: 0 bytes in 0 blocks
           suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of leaked memory

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Memory leak example

```
int main() {
    char *data = nullptr;
    int size = pow(1024, 3);    // 1GB

    for (int i = 0; ; ++i) {
        // Allocate memory for the data.
        data = new char[size];
        std::fill(data, data + size, 'x'); // fill data with 'x'
        cout << "Iteration: " << i << " done. " << (i + 1)
             << " GB has been allocated!" << endl;
    }

    // This will only free the last allocation!
    delete[] data;

    char c; cin >> c; // wait for user input
    return 0;
}
```

Be careful running this example, everything might become slow

Memory leak example

If we run out of memory an std::bad_alloc error is thrown

Be careful running this example, everything might become slow

```
...
Iteration: 19 done. 20 GiB has been allocated!
Iteration: 20 done. 21 GiB has been allocated!
Iteration: 21 done. 22 GiB has been allocated!
Iteration: 22 done. 23 GiB has been allocated!
terminate called after throwing an instance of 'std::bad_alloc’
    what(): std::bad_alloc
    [1] 30561 abort (core dumped) ./memory_leak_2
```

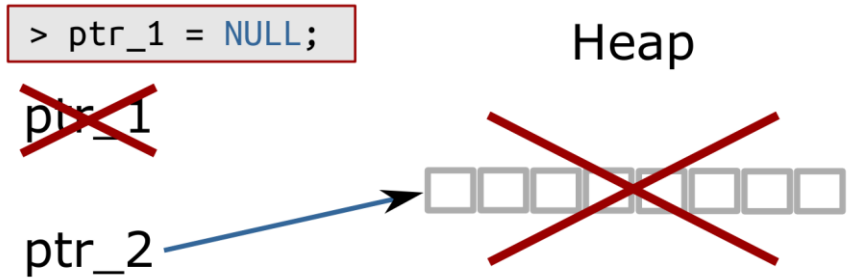
Dangling pointer

Dangling pointer: pointer to freed memory

Think of it as the opposite of a memory leak

Dereferencing a dangling pointer causes undefined behavior

```
int* ptr_1 = new int;
int* ptr_2 = ptr_1;
delete ptr_1;
ptr_1 = nullptr;
// Cannot use ptr_2 anymore! Behavior undefined!
```



Dangling pointer example

```
int main() {
    int size = 5;
    int *ptr_1 = new int[size];
    int *ptr_2 = ptr_1;       // Point to same data!
    ptr_1[0] = 100;           // Set some data.

    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
    cout << "ptr_2[0]: " << ptr_2[0] << endl;

    delete[] ptr_1; // Free memory.
    ptr_1 = nullptr;

    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;

    // Data under ptr_2 does not exist anymore!
    cout << "ptr_2[0]: " << ptr_2[0] << endl;

}
```

Even worse when used in functions

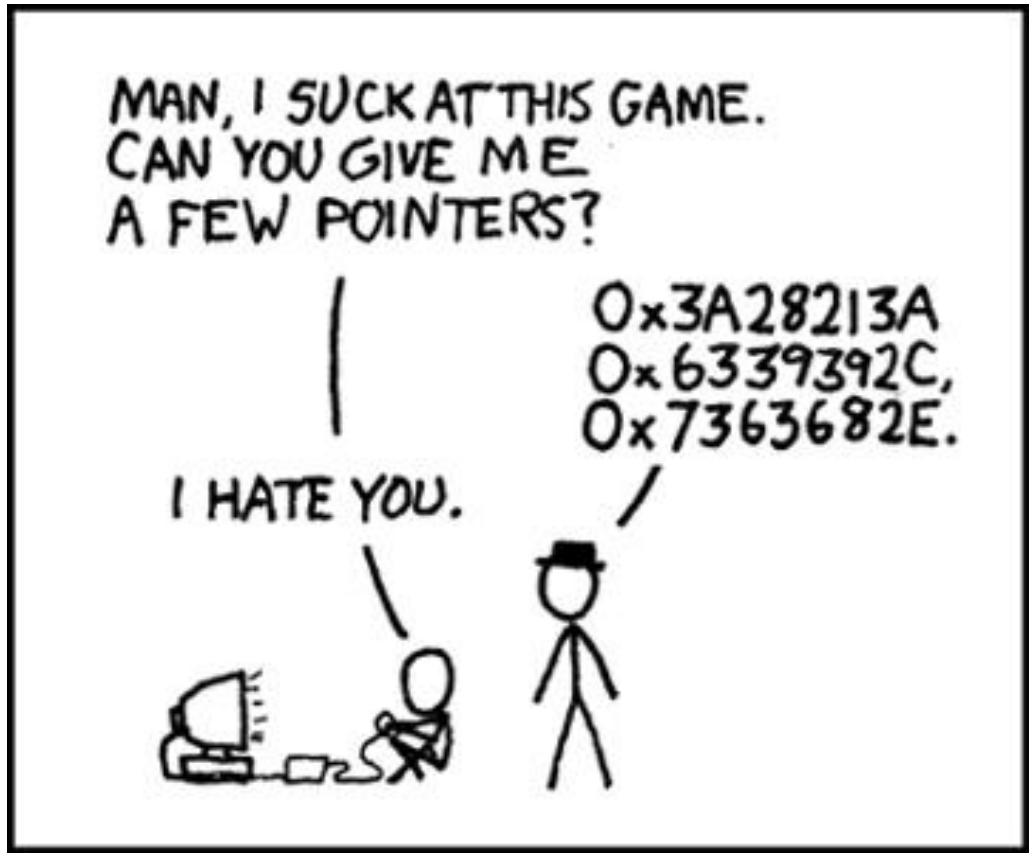
```
#include <stdio.h>
// data processing
int* GenerateData(int size);
void UseDataForGood(const int* const data, int size);
void UseDataForBad(const int* const data, int size);

int main() {
    int size = 10;
    int* data = GenerateData(size);

    UseDataForGood(data, size);
    UseDataForBad(data, size);

    // Is data pointer valid here? Should we free it?
    // Should we use 'delete[]' or 'delete'?

    delete[] data;
    // ???????????
}
```


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

Raw pointers are hard to love

- Its declaration doesn’t indicate whether it points to a single object or to an array.
- Its declaration reveals nothing about whether you should destroy what it points to when you’re done using it, i.e., if the pointer owns the thing it points to.
- There’s typically no way to tell if the pointer dangles, i.e., points to memory that no longer holds the object the pointer is supposed to point to.

Application: Pass by Reference

```
#include<iostream>
#include<vector>
using std::cout, std::endl, std::vector;

// "pass by reference" using pointers
void print(vector<int>* v) {
    for(int x: *v)
        cout << x << " ";
    cout << "\n";
}

int main() {
    vector<int> z = {1,2,3,4,5,6,7,8};
    print(&z);
}
```

Application: Pass by Reference (another example)

```
#include<iostream>
using std::cout, std::endl;

// "pass by reference" using pointers
void print(int* v, int size) {
    for(int i=0; i<size; i++)
        cout << v[i] << " ";
    cout << "\n";
}

int main() {
    int z[8] = {1,2,3,4,5,6,7,8};
    print(z, size(z));
}
```