

## C/C++ VERSUS JAVA

---

- Syntax is mostly the same, but different library functions.

```
System.out.println("Hello"); versus cout << "Hello";
```

- No automatic memory management in C++.
  - You have to delete what you create dynamically (i.e., using `new`).
- C++ does have classes (and thus methods) but it also have functions, which are actually the fundamental building blocks.
  - In particular, `main` is a **function**, not a method.
- C++ uses the concept of **header files** containing function, variable, and data type declarations.
  - Pain at the beginning, but you will come to appreciate them.
- In C++, the type `int` is stored in a word of memory whose size is machine dependent.
- C++ has explicit pointers.

## “ONE FUNCTION” PROGRAMS

---

- The simplest possible program pattern:

```
int main(int argc, char** argv) {  
    code that does something useful  
}
```

- The **code** looks similar to what you would see inside a Java method.

- contains **data declaration** and **actual code** (or instructions)

- **data** looks just like in Java, with some minor differences

```
int total;                float balance[size];  
const int size = 100;
```

- **code** looks again similar to Java code

- \* Notable exception: no difference between booleans and integers (0 = false).

- \* Thus the following sucks but is nonetheless correct:

```
int i = 5;  
while (i) { i--; }
```

## “ONE CLASS” PROGRAMS

---

- If your `main` functions grows too long or too complicated, you should split the program into multiple `functions`.

```
public static int square(int x) {  
    return x * x;  
}
```

- You will get something like this:

```
#include <iostream>  
using namespace std;
```

```
int square(int x) {  
    return x * x;  
}
```

```
int main(int argc, char** argv) {  
    cout << "Five squared is " << square(5) << "\n";  
}
```

## BASIC C++ TYPES

---

- The basics:

Type	Declaration	Literals	What it really means
boolean	<code>bool a;</code>	<code>true false</code>	Could also use integers
characters	<code>char a;</code>	<code>'b' '\n' '\\'</code>	Really a one-byte integer
floating	<code>float a;</code>	<code>3.1415</code>	$3.4 \times 10^{-38}$ to $3.4 \times 10^{-38}$ , 7 digits
point	<code>double a;</code>		$1.7 \times 10^{-308}$ to $1.7 \times 10^{-308}$ , 15 digits

- A whole bunch of integers:

Keyword	Min	Max	Bytes
<code>char</code>	-128	127	1
<code>unsigned char</code>	0	255	1
<code>short</code>	-32,768	32,767	2
<code>int</code>	-2,147,483,648	2,147,483,647	4
<code>long int</code>			8

- Actually, `int` is **one machine word**, `short` is half that and `long` is twice that
- `char` is always a byte

## COMPOUND STATEMENTS

---

- (Compound) statement:     { first-statement second-statement ... }

- Conditional:

```
if (condition)
    statement
```

```
if (condition)
    statement
else
    statement
```

- Loops:

```
while (condition)
    statement
```

```
for (init-statement; condition; update-statement)
    statement
```



```
{
    init-statement
    while (condition) {
        statement
        update-statement;
    }
}
```

break continue  remember them?
--

## INPUT

---

- We have an object `cout` that takes care of output; similarly, the object that takes care of input is `cin`.
- The operator that sends data to the output is `<<`; similarly, the operator that takes data from the input is `>>`.

```
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
    double radius = -1, area;
    const double PI = 3.14159;

    while (radius != 0) {
        cout << "Enter radius: ";
        cin >> radius;
        if (radius == 0) cout << "Bye\n";
        else {
            area = PI * radius * radius;
            cout << "Area is " << area << '\n';
        }
    }
}
```

## UNEXPECTED INPUT

---

- Sometime what you see is not what you get:

```
int main(int argc, char** argv) {
    char aString[100];

    cout << "Type something: ";
    cin >> aString;
    cout << "You typed \"" << aString << "\"\n";
}
```

## UNEXPECTED INPUT

---

- Sometime what you see is not what you get:

```
int main(int argc, char** argv) {
    char aString[100];

    cout << "Type something: ";
    cin >> aString;
    cout << "You typed \"" << aString << "\"\n";
}
```

- So we do instead:

```
int main(int argc, char** argv) {
    char aString[100];

    cout << "Type something: ";
    cin.getline(aString,100);
    cout << "You typed \"" << aString << "\"\n";
}
```

## MORE OUTPUT

---

- Output has its own particularities:

```
#include <iostream>
using namespace std;
```

```
int main(int argc, char** argv) {
    cout << "This line should show up before hanging\n";
    while (1); }
```

## MORE OUTPUT

---

- Output has its own particularities:

```
#include <iostream>
using namespace std;
```

```
int main(int argc, char** argv) {
    cout << "This line should show up before hanging\n";
    while (1); }
```

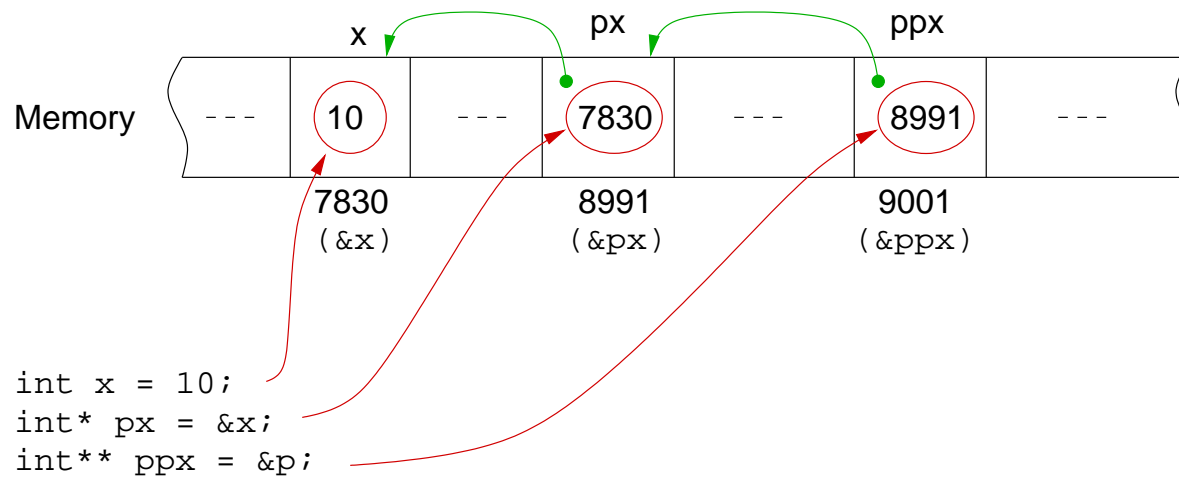
- So we do instead:

```
#include <iostream>
using namespace std;
```

```
int main(int argc, char** argv) {
    cout << "This line should show up before hanging\n";
    cout.flush();
    while (1); }
```

## ARRAYS AND POINTERS

- What is an array?
  - It is a **pointer** to its content!
  - Yes, but what is a pointer?



- Special pointer: 0 (NULL), which points to nothing.

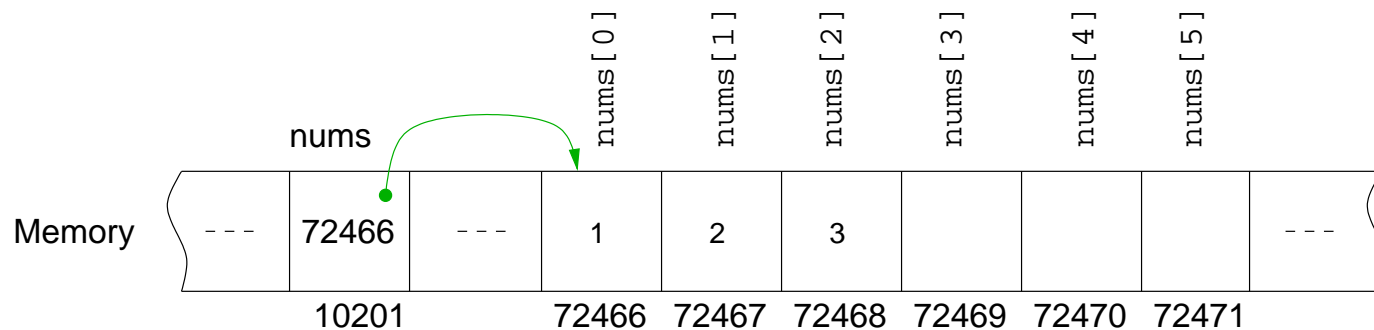
- So, what is an array?

## ARRAYS AND POINTERS (CONT'D)

---

- As I was saying, an array is a pointer to its content:

```
int nums[6] = {1,2,3}
```



- What do they all mean?

```
int nums[] = {1,2,3,4};      *p2 = 10;
int* p1 = nums;             *(nums+3) = 10;
int* p2 = nums + 3;         nums[3] = 10;
```

- How do you copy an array?

## C STRINGS

---

- There is no special type for strings (literal strings can be written surrounded by double quotes though).
  - Instead, strings are simply arrays of characters, or `char*`:

```
char message[20] = "Hello.";  
char *message = "Hello";
```
  - The last character in a string is always the null byte (`'\0'`). So if you declare a string of size 20 it will hold a maximum of 19 characters.
    - \* C **does not check for array overflow**; be careful not to go over the array size.
  - You can access individual characters just as you access elements in a normal array:

```
message[1] = 'x';
```
- Strings **cannot** be compared using the usual comparison operators (e.g., `==`).
  - Use `strcmp` instead.

## OPERATIONS ON STRINGS

---

- You can implement your own operations on strings (just do not forget about the null byte at the end).
- Some operations are already defined for you though, including:
  - Copy a string: `strcpy` and `strncpy`
  - Length of a string: `strlen`
  - Best thing since sliced bread: `snprintf`
- Just do not forget to include the appropriate header:  
`#include <string.h>`

## ARRAYS OF STRINGS

---

- Arrays of strings are bi-dimensional arrays of characters:

```
char days[7][3] = { "Mon", "Tue", "Wed" };  
char *days[] = { "Mon", "Tue", "Wed" };
```

- Used between other things for **passing arguments and other stuff to programs**.

- As in `int main (int argc, char** argv, char** envp)`
- The size of `argv` is given by `argc`
- The array `envp` however is **NULL-terminated!**
- Equally good (and equally used) strategies for specifying the size of an array
- Incidentally speaking, `envp` is the **environment** of the program; each string has the form “VARIABLE=value” (or you can use `getenv`)

## ARRAYS, POINTERS, AND FUNCTIONS

---

```
#include <iostream>
using namespace std;

void translate(char a) {
    if (a == 'A') a = '5'; else a = '0';
}

void translate(char* array, int size) {
    for (int i = 0; i < size; i++) {
        if (array[i] == 'A') array[i] = '5';
        else array[i] = '0';
    }
}

int main () {
    char mark = 'A'; char marks[5] = {'A', 'F', 'A', 'F', 'F'};
    translate(mark);
    translate(marks, 5);
    cout << mark << "\n";
    for (int i = 0; i < 5; i++)
        cout << marks[i] << " ";
    cout << "\n";
}
```

## ARRAYS, POINTERS, AND FUNCTIONS

---

```
#include <iostream>
using namespace std;

void translate(char a) { // by the way, an OVERLOADED FUNCTION (C++)
    if (a == 'A') a = '5'; else a = '0';
}

void translate(char* array, int size) {
    for (int i = 0; i < size; i++) {
        if (array[i] == 'A') array[i] = '5';
        else array[i] = '0';
    }
}

int main () {
    char mark = 'A'; char marks[5] = {'A', 'F', 'A', 'F', 'F'};
    translate(mark);
    translate(marks, 5);
    cout << mark << "\n";
    for (int i = 0; i < 5; i++)
        cout << marks[i] << " ";
    cout << "\n";
}
```

A
5 0 5 0 0

## POINTERS AND FUNCTIONS

---

- An argument can be passed in C++ to a function using:
  - **Call by value:** the **value** of the argument is passed; argument cannot be changed by the function.

```
int aFunction(int i);
```

- **Call by reference:** the **pointer** to the argument is passed to the function; argument can be changed at will by the function.

```
int aFunction(int* i);
```

Used for **output arguments**.

- **Call by constant reference:** the **pointer** to the argument is passed to the function; **but** the function is not allowed to change the argument.

```
int aFunction(const int* i);
```

most used:

```
int aFunction(const char* i);
```

Used for **bulky arguments**.

## CALL BY REFERENCE

---

foo.cc

```
void increment (int* i) {
    *i = *i + 1;
}

void increment1 (const int* i) {
    *i = *i + 1;
}

int main () {
    int n = 0;
    increment(&n);
    increment1(&n);
}
```

g++ -Wall foo.cc

```
foo.cc: In function 'void increment1(const int *)':
foo.cc:9: assignment of read-only location
```

## STRUCTURES

---

- An array holds a number of elements of a given type.
  - Individual elements are referred to by integer indices.
- By contrast, a **structure** holds elements of not necessarily the same type.
  - Individual elements are referred to by **symbolic names**.
  - Of course, we cannot thus loop over the members of a structure.
- For instance, a structure representing a student might contain
  - the given name and surname (strings),
  - the student number (integer),
  - the mailbox number (integer), and
  - the grade point average (floating point).

## STUDENT STRUCTURE

---

```
struct student {
    char* name;
    char* surname;
    unsigned int number;
    unsigned short mailbox;
    float gpa;
};

int main () {
    student studs[5];
    studs[0].name = "Jane";
    studs[0].surname = "Doe";
    studs[0].number = 1234567;
    studs[1].name = "John";
    studs[1].surname = "Smith";
    studs[1].number = 7654321;
    cout << studs[1].name << " "
         << studs[1].surname
         << " ("
         << studs[1].number
         << ")\n";
}

int main () {
    struct student {
        char* name;
        char* surname;
        unsigned int number;
        unsigned short mailbox;
        float gpa;
    } studs[5];

    studs[0].name = "Jane";
    studs[0].surname = "Doe";
    studs[0].number = 1234567;
    studs[1].name = "John";
    studs[1].surname = "Smith";
    studs[1].number = 7654321;
    cout << studs[1].name << " "
         << studs[1].surname
         << " ("
         << studs[1].number
         << ")\n";
}
```

## C STRUCTURES

---

- typedef `type alias` establishes `alias` as an alias for `type`
- Useful between others for using C structures in a more convenient manner:

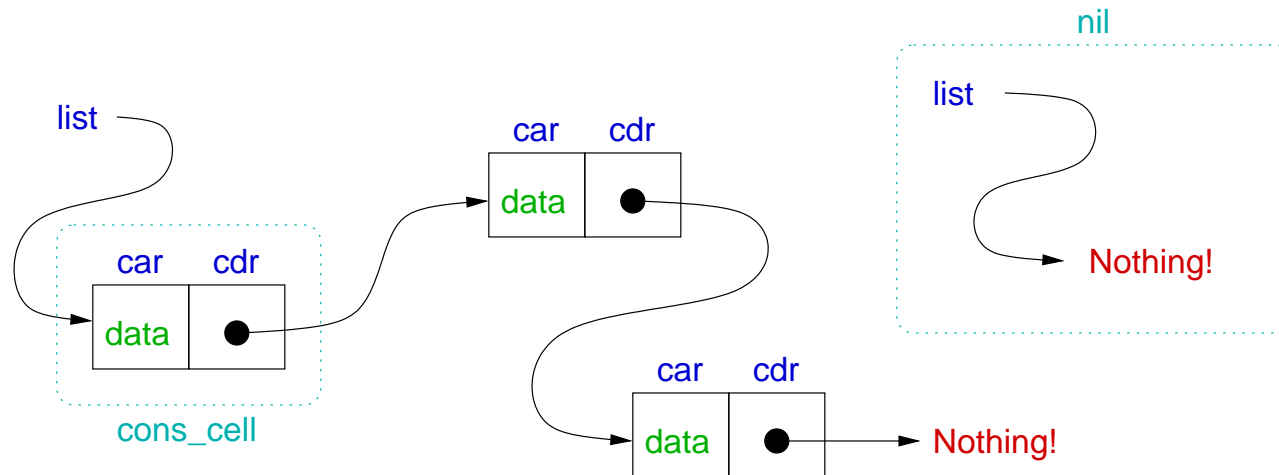
```
struct student {  
    char* name;  
    char* surname;  
    unsigned int number;  
    unsigned short mailbox;  
    float gpa;  
};
```

```
typedef struct student_t {  
    char* name;  
    char* surname;  
    unsigned int number;  
    unsigned short mailbox;  
    float gpa;  
} student;
```

## POINTERS TO STRUCTURES

---

- Let's do something useful: a (linked) list (of integers).



- Interesting operations:

<code>cons</code>	adds an integer to the list
<code>car</code>	returns the first element of a list
<code>cdr</code>	returns a list without its first element
<code>null</code>	returns true iff the list is empty

## LINKED LIST

---

```
// Header: contains data structures and
// function declarations (prototypes)
#ifndef __LIST_H
#define __LIST_H

struct cons_cell {
    int car;
    cons_cell* cdr;
};

typedef cons_cell* list;
// bad programming practice!

const list nil = 0;

int null (list);
list cons (int, list = nil);
int car (list);
list cdr (list);

#endif // __LIST_H
```

```
// C++ file: contains
// function definitions
#include "list.h"

int null (list cons) {
    return cons == nil;
}

list cons (int car, list cdr) {
    list new_cons = new cons_cell;
    new_cons -> car = car;
    // (*new_cons).car = car;
    new_cons -> cdr = cdr;
    // (*new_cons).cdr = cdr;
    return new_cons;
}

int car (list cons) {
    return cons -> car;
}

list cdr (list cons) {
    return cons -> cdr;
}
```

## DYNAMIC MEMORY ALLOCATION

---

- `new` allocates memory for your data. The following are (somehow) equivalent:

```
char message[256];      char* pmessage;  
                        pmessage = new char[256];
```

- **Exception:**

- \* `message` takes care of itself (i.e., gets deleted when it is no longer in use),  
whereas

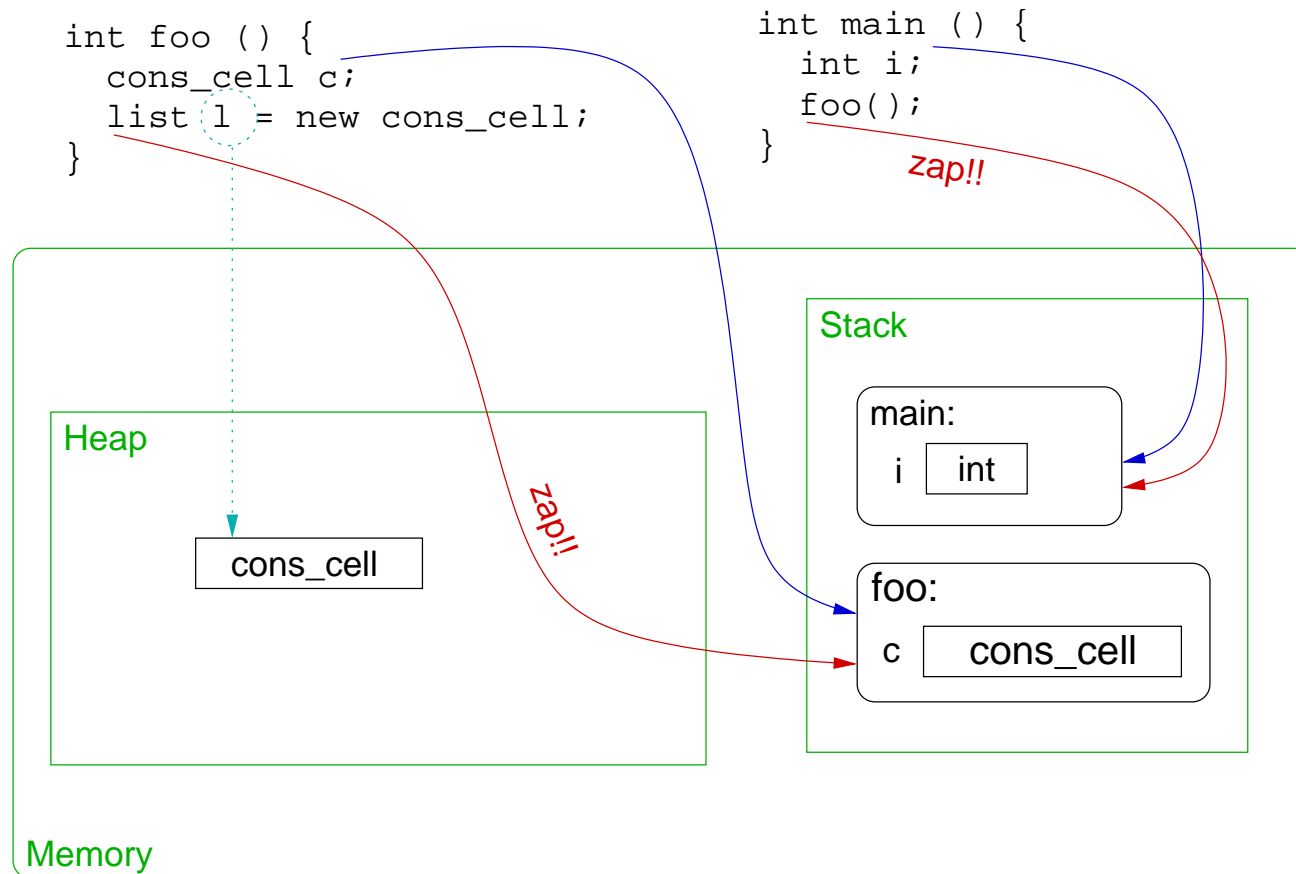
- \* `pmessage` however must be explicitly deleted when it is no longer needed:

```
delete[] pmessage;
```

- **Perrils of `not` using `new`:**

```
list cons (int car, list cdr) {      #include "list.h"  
    cons_cell new_cons;  
    new_cons.car = car;  
    new_cons.cdr = cdr;  
    return &new_cons;  
}  
  
int main () {  
    list bad = cons(1);  
    cout << car(bad);    → Boom!  
}
```

# DYNAMIC MEMORY MANAGEMENT



Conclusion: ✓ `foo` returns `l` ✗ `foo` returns `c`

## USING LINKED LISTS

---

```
list rmth (list cons, int which) {
    list place = cons;
    for (int i = 0; i < which - 1; i++) {
        if (null(place))
            break;
        place = place -> cdr;
    }
    if (! null(place) ) {
        if (null(cdr(place)))
            place -> cdr = nil;
        else
            place -> cdr = cdr(place -> cdr);
    }
    return cons;
}
```

## MEMORY LEAKS

---

- A good example:

```
list rmth (list cons, int which) {
    list place = cons;
    for (int i = 0; i < which - 1; i++) {
        if (null(place))
            break;
        place = place -> cdr;
    }
    if (! null(place) ) {
        if (null(cdr(place)))
            place -> cdr = nil;
        else
            place -> cdr = cdr(place -> cdr);
    }
    return cons;
}
```

- If you create something using `new` then you **must eventually delete it** using `delete`.
- But you should not leave **stale pointers** behind either!

## SAY NO TO MEMORY LEAKS

---

```
list rmth (list cons, int which) {
    list place = cons;
    for (int i = 0; i < which - 1; i++) {
        if (null(place))
            break;
        place = place -> cdr;
    }
    if (! null(place) ) {
        if (null(cdr(place)))
            place -> cdr = nil;
        else {
            list to_delete = cdr(place);
            place -> cdr = cdr(place -> cdr);
            delete to_delete;
        }
    }
    return cons;
}
```

## THE PERILS OF DELETE

---

- Thou shall not leak memory, but also:
- Thou shall not leave **stale pointers** behind.

```
char* str = new char[128];  
strcpy(str, "hello");  
char* p = str;  
delete [] p;
```

→ allocate memory for `str`  
→ put something in there ("hello")  
→ `p` points to the same thing  
→ "hello" is gone,  
   `str` is a **stale pointer!!**

- Thou shall not **dereference deleted pointers**.

```
strcpy(str, "hi");
```

→ `str` **already deleted!!**

- Thou shall not delete a pointer more than once.

```
delete str;
```

→ `str` **already deleted!!**

- You can however **delete null pointers as many times as you wish!**

## THE PERILS OF DELETE

---

- Thou shall not leak memory, but also:
- Thou shall not leave **stale pointers** behind.

```
char* str = new char[128];  
strcpy(str, "hello");  
char* p = str;  
delete [] p;
```

→ allocate memory for `str`  
→ put something in there (“hello”)  
→ `p` points to the same thing  
→ “hello” is gone,  
    `str` is a **stale pointer**!!

- Thou shall not **dereference deleted pointers**.

```
strcpy(str, "hi");
```

→ `str` **already deleted**!!

- Thou shall not delete a pointer more than once.

```
delete str;
```

→ `str` **already deleted**!!

- You can however **delete null pointers as many times as you wish!**
- So **assign zero to deleted pointers** whenever possible (**not a panaceum**)