

POINTERS

- What is a pointer?
 - The [index](#) of a book contains pointers.
 - A [URL](#) (e.g., <http://turing.ubishops.ca/home/csc218>) is a pointer.
 - A [street address](#) is a pointer.
 - What is then a [forwarding address](#)?

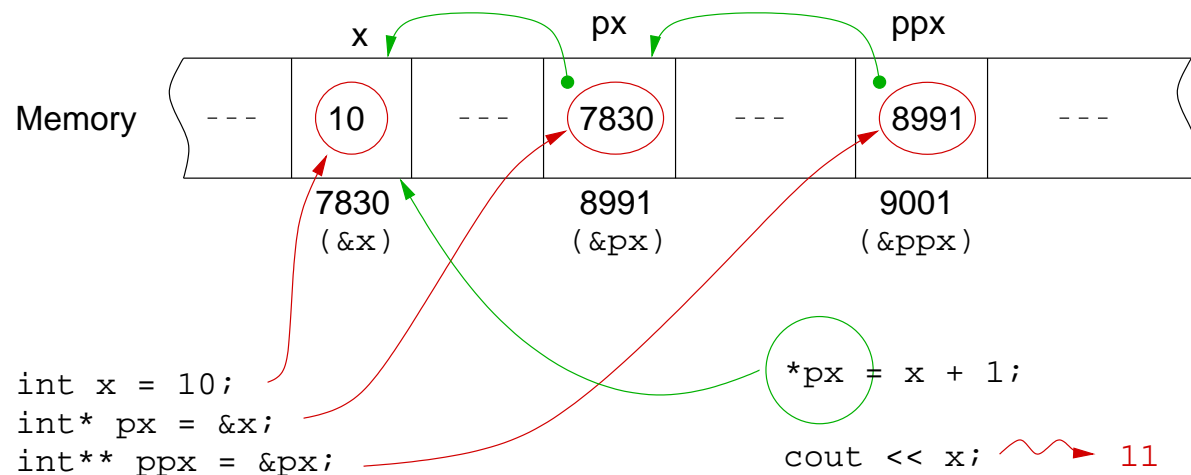
POINTERS

- What is a pointer?
 - The **index** of a book contains pointers.
 - A **URL** (e.g., <http://turing.ubishops.ca/home/csc218>) is a pointer.
 - A **street address** is a pointer.
 - What is then a **forwarding address**?
 - * a **pointer to a pointer**!
- OK, so what is a (C++) pointer?
 - Computer memory contains data which can be accessed using an address.
 - * A pointer is such an address, nothing more.
 - If you want, computer memory is like an **array** holding data.
 - * A pointer then is an **index** in such an array.
 - What are in fact pointers?

POINTERS (CONT'D)

- Pointers can (just as array indices) be stored in variables.
- If we have some type d , then

$d \text{ } vx;$	\rightarrow	vx is a variable of type d
$d^* \text{ } px;$	\rightarrow	px is a (variable holding a) pointer to a variable of type d
$\&vx$	\rightarrow	denotes the address of vx (i.e., a pointer , of type d^*)
$*px$	\rightarrow	denotes the value from the memory location pointed at by px , of type d (we thus dereference px)



WHAT POINTERS REALLY ARE

- Since a pointer is an address, it is usually represented internally as `unsigned int`.
- Do we need a type for a pointer?
 - Why?
 - Always?

WHAT POINTERS REALLY ARE

- Since a pointer is an address, it is usually represented internally as unsigned int.
- Do we need a type for a pointer?
 - Why?
 - Always?

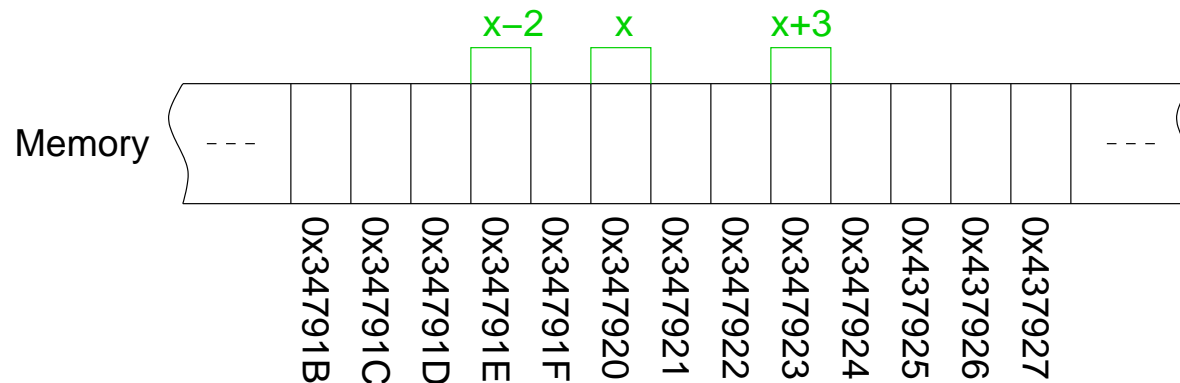
```
int x=10;
void* p = &x;
int * pi;
float* pf;
pi = (int*)p;
pf = (float*)p;
cout << "Pointer " << p << " holds the int: " << *pi
      << " ...and the float: " << *pf << "\n";
```

- Special pointer (of type void*): NULL (really, 0), which points to nothing.

POINTER ARITHMETIC

- The types of pointers do matter:
 1. We know what we get when we **dereference** a pointer
 2. We can do meaningful **pointer arithmetic**

```
int i=10;          long j=10;
int *x = &i;       long *y = &j;
int *x1 = x + 3;   long *y1 = y + 3;
int *x2 = x - 2;   long *y2 = y - 2;
```

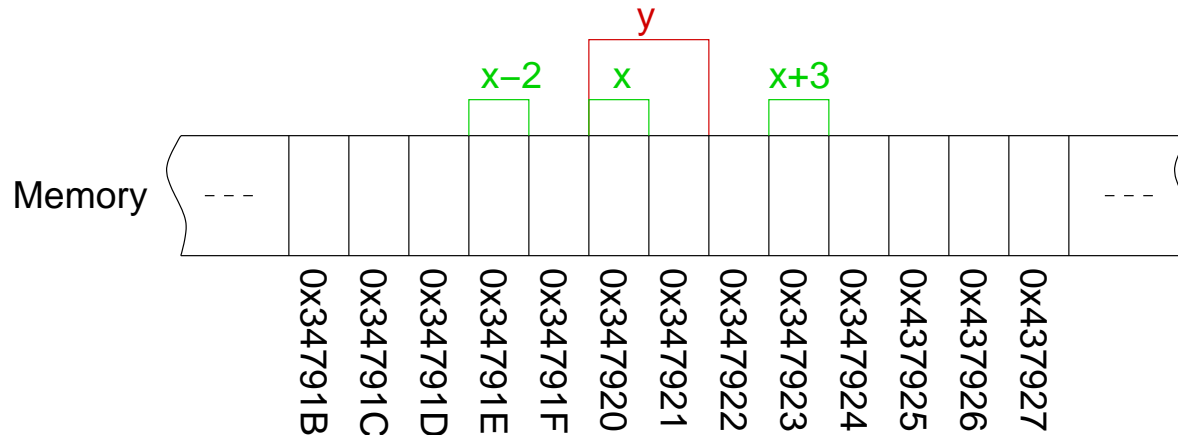


- **Meaningful** pointer arithmetic?!?

POINTER ARITHMETIC

- The types of pointers do matter:
 1. We know what we get when we **dereference** a pointer
 2. We can do meaningful **pointer arithmetic**

```
int i=10;          long j=10;
int *x = &i;       long *y = &j;
int *x1 = x + 3;   long *y1 = y + 3;
int *x2 = x - 2;   long *y2 = y - 2;
```

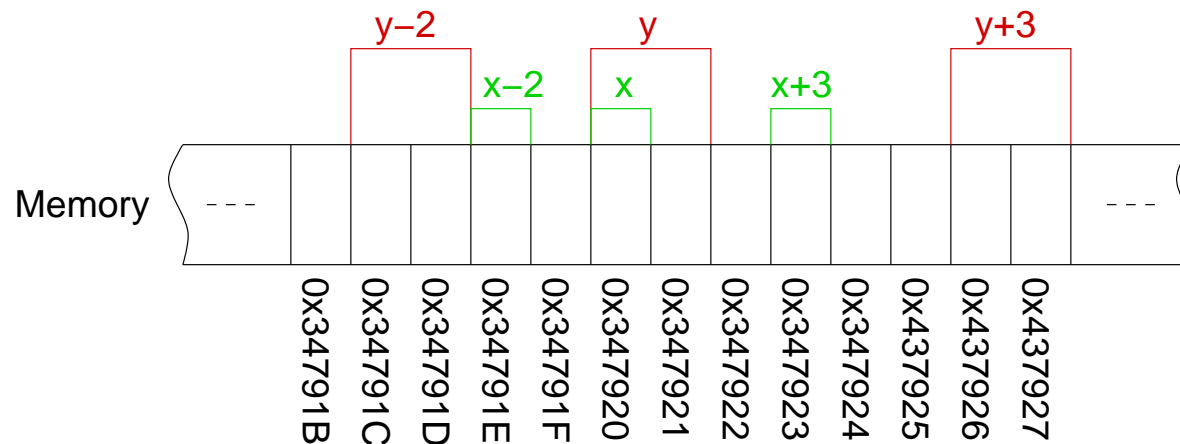


- **Meaningful** pointer arithmetic?!?

POINTER ARITHMETIC

- The types of pointers do matter:
 1. We know what we get when we **dereference** a pointer
 2. We can do meaningful **pointer arithmetic**

```
int i=10;          long j=10;
int *x = &i;       long *y = &j;
int *x1 = x + 3;   long *y1 = y + 3;
int *x2 = x - 2;   long *y2 = y - 2;
```

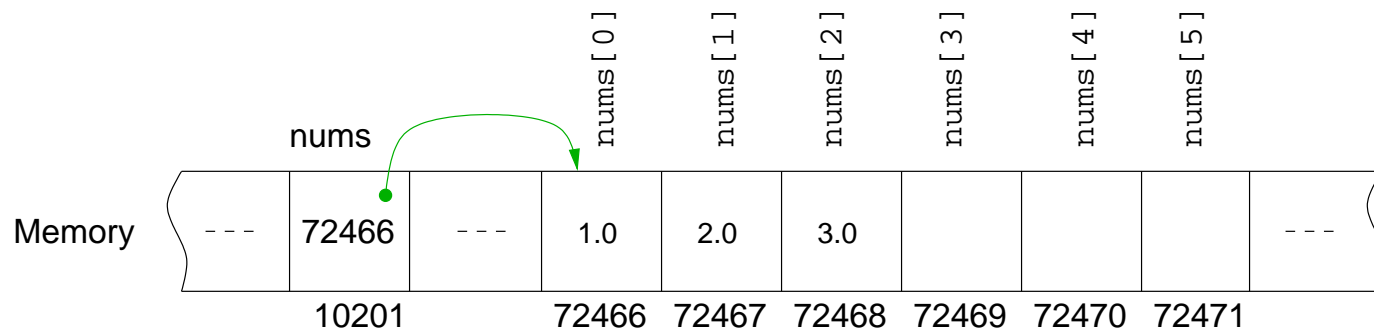


- **Meaningful** pointer arithmetic?!?

ARRAYS AND POINTERS

- An array is just a **pointer** to its content:

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



- In addition, when you declare an array (contiguous) **memory space** is also reserved to hold its elements.

- What do they all mean?

```
float nums[6] = {1,2,3};  
float* p1 = nums;  
float* p2 = nums + 3;
```

```
int nums[6] = {1,2,3};  
int* p1 = nums;  
int* p2 = nums + 3;
```

ARRAYS VERSUS POINTERS

- The following declarations mean **almost** the same thing:

```
int* numsP;  
int  numsA[20];
```

- Because we have:

```
numsA[2] = 17;    →    Good  
numsP[2] = 17;    →    Disaster!
```

- Prize for the most uninformative error message goes to “Segmentation fault.”

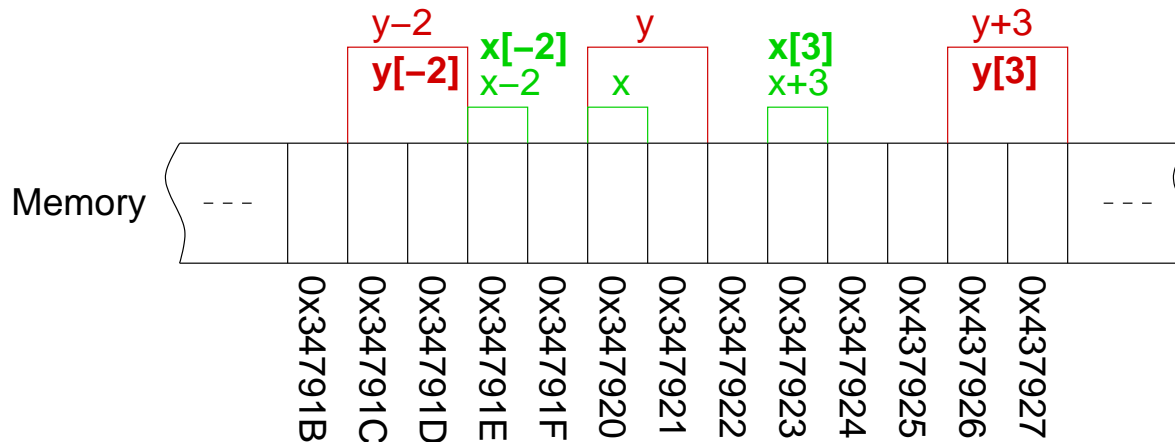
- But it is **perfectly good** to do:

```
int numsP[] = {1,2,3};
```

- In other words, you do not have to provide the dimension for an array if you initialize it at the moment of declaration (e.g., by providing a literal array).

ARRAY SUBSCRIPTS

- We access elements in an array precisely as we do it in Java:
 - `cout << x[6];` prints the seventh element of `x`
 - `x[5] = 20;` assigns 20 to the sixth element of `x`
- The subscript operator `[]` is in fact implemented using pointer arithmetic
 - `x[5]` is a shorthand for (and thus a **perfect equivalent** to) `&x+5`.
 - the subscript operator works with **any** pointer, not just with arrays.
 - it does correct pointer arithmetic so that we access the intended element



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) { // translate, by the way, is a OVERLOADED FUNCTION
    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

ARRAYS, POINTERS, AND FUNCTIONS (CONT'D)

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

int translate(char a) { // still overloaded...
    if (a == 'A') a = '5'; else a = '0';
    return a;
}

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

5
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** (messy, error prone syntax).

- **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);
```

more useful:

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

Used for **bulky arguments** (still messy syntax).

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
```

CALL BY REFERENCE (CONT'D)

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);
}
```

→ no more messy syntax!

g++ -Wall foo.cc

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

CALL BY REFERENCE (CONT'D)

foo.cc

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

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

int increment1 (const int& i) {
    int r = i + 1;
    return r;
}

int main () {
    int n = 0;
    increment(n);
    cout << n << "\n";
    n = increment1(n);
    cout << n << "\n";
}
```

output

```
1
2
```

CALLING CONVENTIONS IN C++ AND JAVA

- The following are the implicit calling conventions:

What	Java	C++
Primitive types (int, float, etc.)	value	value
Arrays	reference	value
Objects	reference	value

- In C++ **everything is passed by value unless explicitly stated otherwise**.
Arrays are apparently passed by reference, but only because of the array structure (pointer + content).
- In Java there is no other way to pass arguments than the implicit one.
- In C++ you can request that an argument be passed by reference by either passing a pointer to the actual argument or by saying explicitly that you want to pass the argument by reference.

C STRINGS

- There is no special type for strings.
 - Instead, strings are simply arrays of characters.
 - * Literal strings can be written surrounded by double quotes though.

```
char message[20] = "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**, so 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., `==`) ([why?](#)).
 - 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` (see `man strcpy`)
 - Length of a string: `strlen` (see `man strlen`)
- Just do not forget to include the appropriate header:
`#include <string.h>`

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;  
};
```

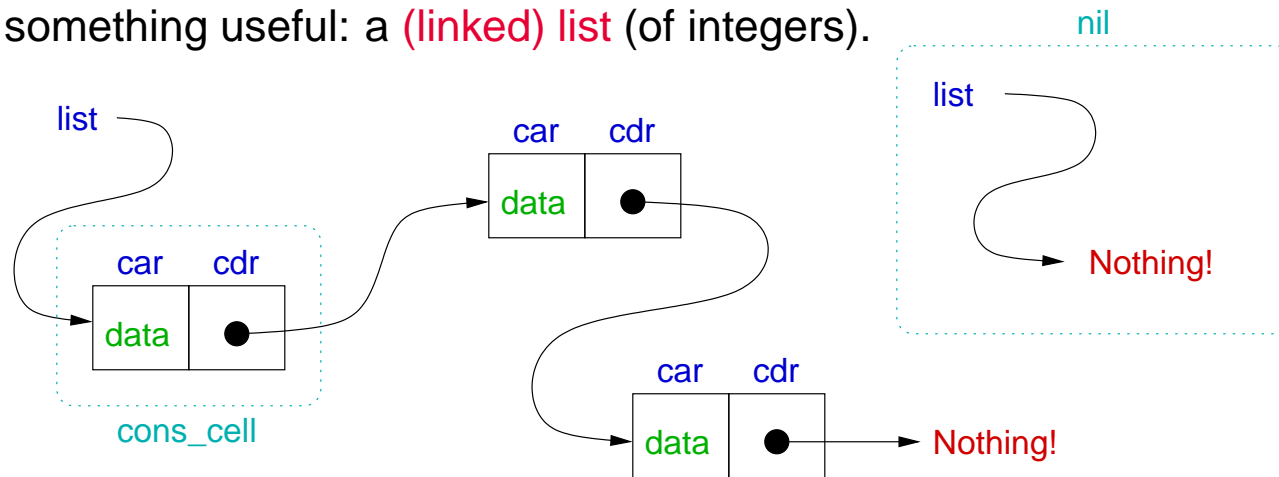
STUDENT STRUCTURE

```
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";
}
```

POINTERS TO STRUCTURES

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



- Interesting operations:

Operation	Meaning
cons	adds an integer to the list
car	returns the first element of a list
cdr	returns a list without its first element
null	returns true iff the list is empty

LINKED LIST

```
struct cons_cell {
    int car;
    cons_cell* cdr;
};
typedef cons_cell* list;    // careful, could be bad programming practice!

const list nil = 0;

int null (list cons) {
    return cons == nil;
}
list cons (int car, list cdr = nil) {
    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;
}
```

NEW (AND DELETE)

- `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:

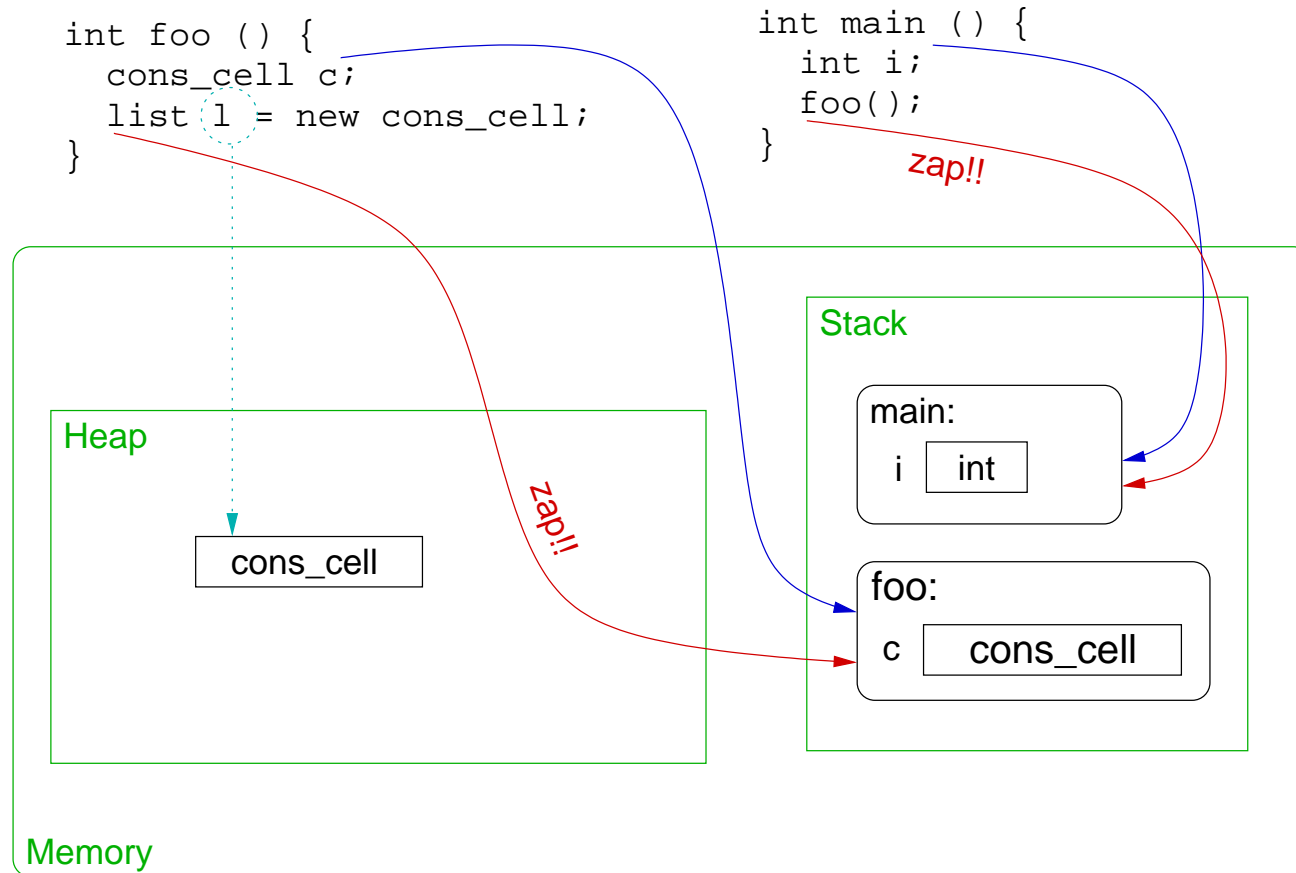
```
delete[] pmessage;
```

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

```
list cons (int car, list cdr = nil) {  
    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;
}
```

USING LINKED LISTS (CONT'D)

```
int main () {
    int elm = -1;
    list lst = nil;
    while (elm != 0) {
        cin >> elm;
        if (elm != 0)
            lst = cons(elm,lst);
    }
    lst = rmth(lst,1);
    lst = rmth(lst,10);
    cout << "List is:\n";
    list iter = lst;
    while (! null(iter) ) {
        cout << car(iter) << "\n";
        iter = cdr(iter);
    }
}
```

```
1
2
3
4
5
0
List is:
5
3
2
1
```

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`.

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;
}
```

STUDENT STRUCTURE, TAKE TWO

- The following won't work. Why? What would happen if it would work?

```
struct student {
    char name[20];
    char surname[20];
    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";
}
```

STUDENT STRUCTURE, TAKE TWO (CONT'D)

- The following **does** work.

```
struct student {
    char name[20];
    char surname[20];
    unsigned int number;
    unsigned short mailbox;
    float gpa;
};

int main () {
    student studs[5];
    strncpy(studs[0].name, "Jane", 20);
    strncpy(studs[0].surname, "Doe", 20);
    studs[0].number = 1234567;
    strncpy(studs[1].name, "John", 20);
    strncpy(studs[1].surname, "Smith", 20);
    studs[1].number = 7654321;
    cout << studs[1].name << " " << studs[1].surname
         << " (" << studs[1].number << ")\n";
}
```

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**)

THE PERILS OF DELETE (CONT'D)

```
struct prof {
    char* name;
    char* dept;
};
char *csc = new char[30];
strcpy (csc, "Computer Science");
prof *stefan, *dimitri, *bruda;
stefan = new prof; dimitri = new prof;
stefan->name = new char[30];
dimitri->name = new char[30];
```

```
strcpy(stefan->name, "Stefan Bruda");
strcpy(dimitri->name, "Dimitri Vouliouris");
```

```
stefan->dept = csc;
dimitri->dept = csc;
```

// Delete dimitri

```
delete dimitri->name;
delete dimitri->dept;
delete dimitri;
```

Exogenous data

OK

???

Indigenous data

// Copy stefan

```
bruda = new prof;
```

// (a) Shallow copying

```
bruda->name = stefan->name;
bruda->dept = stefan->dept;
```

// Can we delete stefan now??

// (b) Deep copying

```
bruda->name = new char[30];
bruda->dept = new char[30];
strcpy(bruda.name, stefan.name);
strcpy(bruda.dept, stefan.dept);
```

// Can we delete stefan now??