CS 403: Pointers, References, and Memory Management

Stefan D. Bruda

Fall 2021

http://xkcd.com/138/

Pointers

- What is a pointer?
  - The index of a book contains pointers
  - A URL (e.g., `http://cs.ubishops.ca/home/cs403`) is a pointer
  - A street address is a pointer
  - What is then a forwarding address?

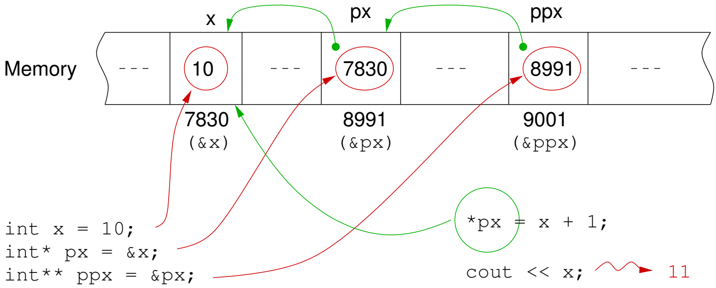
POINTERS

- What is a pointer?
  - The index of a book contains pointers
  - A URL (e.g., `http://cs.ubishops.ca/home/cs403`) is a pointer
  - A street address is a pointer
  - What is then a forwarding address? $\rightarrow$ a pointer to a pointer!
- OK, so what is then a (C/C++) pointer?
  - Often need to refer to some object without making a copy of the object itself
  - Computer memory contains data which can be accessed using an address
    - A pointer is nothing more than such an address
  - Alternatively, computer memory is like an array holding data
    - A pointer then is an index in such an array
  - What are pointers physically?

Pointers

- What is a pointer?
  - The index of a book contains pointers
  - A URL (e.g., `http://cs.ubishops.ca/home/cs403`) is a pointer
  - A street address is a pointer
  - What is then a forwarding address? → a pointer to a pointer!
- OK, so what is then a (C/C++) pointer?
  - Often need to refer to some object without making a copy of the object itself
  - Computer memory contains data which can be accessed using an address
    - A pointer is nothing more than such an address
  - Alternatively, computer memory is like an array holding data
    - A pointer then is an index in such an array
  - What are pointers physically?
    - Since a pointer is an address, it is usually represented internally as `unsigned int`
  - Pointers can (just as array indices) be stored in variables.

d `vx;`	$\rightarrow$	`vx` is a variable of type *d*
d∗ `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`)



POINTER TYPES

- Do we need a type for a pointer?
  - Why?
  - Always?

POINTER TYPES

- Do we need a type for a pointer?
  - Why? → So that we know what is the type of the thing we are referring to
  - Always? → Yes, unless the pointer does not point to anything (`void*`)

```
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;
```

Pointer Types

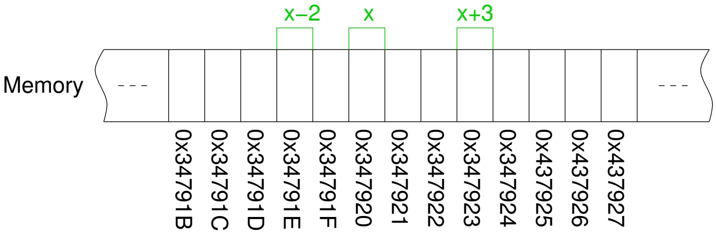
- Do we need a type for a pointer?
  - Why? → So that we know what is the type of the thing we are referring to
  - Always? → Yes, unless the pointer does not point to anything (`void*`)

```
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;
```

- There is nothing preventing a pointer to point to garbage
- Special pointer (of what type?): `NULL` or `0`, which points to nothing

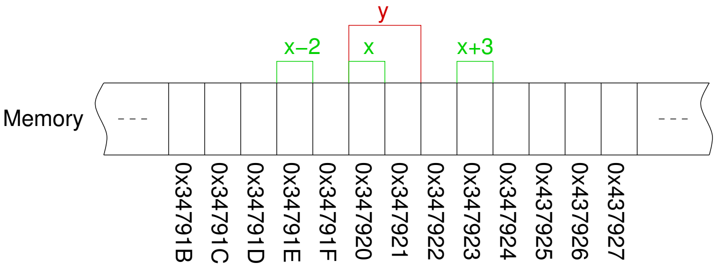
- The types of pointers do matter:
  1. We know what we get when we dereference a pointer
  2. We can do meaningful pointer arithmetic
     - Meaningful pointer arithmetic?!?

```
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;
```



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

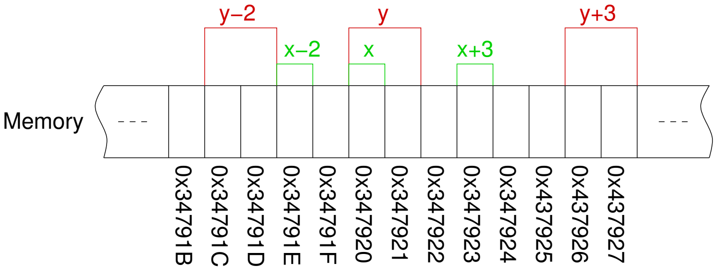
```
int i=10;                long j=10;
int *x = &i;             long *y = &j;
int *x1 = x + 3;         long *y1 = y + 3;
int *x2 = x - 2;         long *y2 = y - 2;
```



POINTER ARITHMETIC

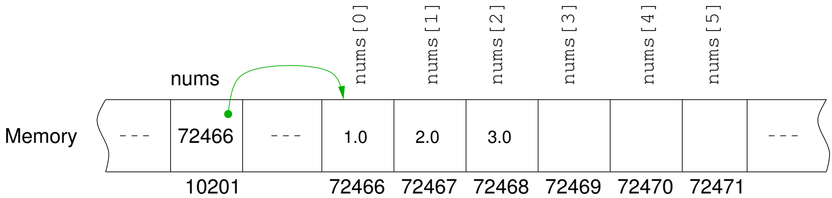
- The types of pointers do matter:
  1. We know what we get when we dereference a pointer
  2. We can do meaningful pointer arithmetic
     - Meaningful pointer arithmetic?!?

```
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;
```



C Arrays and Pointers

- A C 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 reserved to hold its elements
  - Pointer arithmetic goes hand in hand with C arrays

```
float nums[6] = {1,2,3};
float* p1 = nums;
float* p2 = nums + 1;
cout << nums[1] << " " << p1[1] << " " << p2[1];  // 2.0 2.0 3.0
```

  - In fact `arr[index]` is perfectly equivalent with `*(arr+index)`

C Arrays versus Pointers

- The following declarations mean almost the same thing:

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

  - Indeed:

    `numsA[2] = 17;` $\rightarrow$ Good

    `numsP[2] = 17;` $\rightarrow$ Disaster!

  - Prize for the most uninformative error message goes to "`Segmentation fault`."
- However it is perfectly good to do: `int numsP[] = {1,2,3};`
- In other words, one do not have to provide the dimension for an array if one initializes it at the moment of declaration (e.g., by providing a literal array)

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";
}
```

```
#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";
}
```

Output:

```
A
5 0 5 0 0
```

```
#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";
}
```

Output:
5
5 0 5 0 0

POINTERS AND FUNCTIONS

- In C and C++ an argument can be passed to a function using:
  - Call by value: the value of the argument is passed; argument changes will not be seen outside the function

    ```
    int aFunction(int i);
    ```
  - Call by reference: the pointer to the argument is passed to the function; argument changes will be seen outside 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);
    ```
    - Used for bulky arguments (still messy syntax)

CALL BY REFERENCE POINTER

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

foo.cc → no more messy syntax!

```
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 reference ‘i’
```

References and Calling Conventions

- A reference is just like a pointer, but with a nicer interface
  - An alias to an object, but it hides such an indirection from the programmer
  - Must be typed and can only refer to the declared type (`int &r;` can only refer to `int`, etc)
  - Must always refer to something in C++ but may refer to the `null` object in Java
  - Explicit in C++ (unary operator `&`) implicit in Java for all objects and nonexistent for primitive types
- 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 (by declaring the respective parameter as a reference)
    - C 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

```
struct cons_cell {
  int car;
  cons_cell* cdr;    // must use pointers, else the type is infinitely recursive
};

typedef cons_cell* list;

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` 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`:

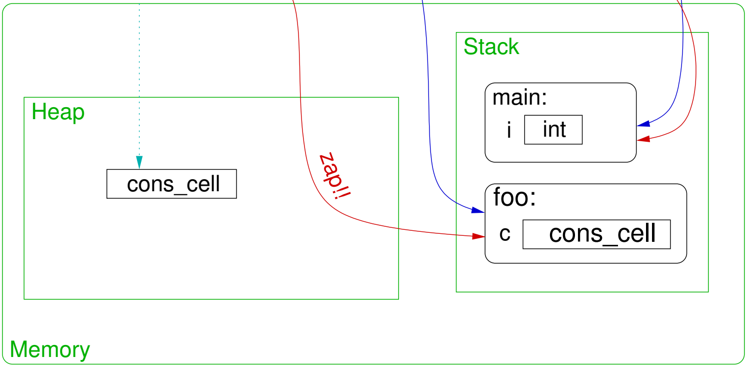
```
list cons (int car,
           list cdr = nil) {
  cons_cell new_cons;
  new_cons.car = car;
  new_cons.cdr = cdr;
  return &new_cons;
}
```

```
int main () {
  list bad = cons(1);
  cout << car(bad);    → Boom!
}
```

```
cons_cell* foo () {
  cons_cell c;
  list l = new cons_cell;
  return ...;
}
```

```
int main () {
  int i;
  foo();
}
```

zap!!



Conclusion: ✓ foo returns l ✗ foo returns c

Say No to Memory Leaks

- If you create something using `new` then you must eventually delete it using `delete`

```
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

- An object and all the pointers to it (when they are dereferenced) alias the same location
  - Assigning a value through one channel will affect all the other channels
  - Memory management through one channel will affect all the other channels as well
- Thou shall not leak memory, but also:
- Thou shall not leave stale (dangling) pointers behind

```
char* str = new char[128];    → allocate memory for str
strcpy(str,"hello");          → put something in there ("hello")
char* p = str;                → p points to the same thing
delete [] p;                  → "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

- An object and all the pointers to it (when they are dereferenced) alias the same location
  - Assigning a value through one channel will affect all the other channels
  - Memory management through one channel will affect all the other channels as well
- Thou shall not leak memory, but also:
- Thou shall not leave stale (dangling) pointers behind

```
char* str = new char[128];    → allocate memory for str
strcpy(str,"hello");          → put something in there ("hello")
char* p = str;                → p points to the same thing
delete [] p;                  → "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)

```
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;
```

Exogenous data

// Delete dimitri

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

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??

POINTERS AS FIRST-ORDER C++ OBJECTS

- They are objects that look and feel like pointers, but are smarter
- Look like pointers = have the same interface that pointers do

- Smarter = do things that regular pointers don't

Pointers as First-Order C++ Objects

- They are objects that look and feel like pointers, but are smarter
- Look like pointers = have the same interface that pointers do
  - They need to support pointer operations like dereferencing (`operator*`) and indirection (`operator ->`)
  - An object that looks and feels like something else is called a proxy object, or just proxy
- Smarter = do things that regular pointers don't

Pointers as First-Order C++ Objects

- They are objects that look and feel like pointers, but are smarter
- Look like pointers = have the same interface that pointers do
  - They need to support pointer operations like dereferencing (`operator*`) and indirection (`operator ->`)
  - An object that looks and feels like something else is called a proxy object, or just proxy
- Smarter = do things that regular pointers don't
  - Such as memory management

Pointers as First-Order C++ Objects

- They are objects that look and feel like pointers, but are smarter
- Look like pointers = have the same interface that pointers do
  - They need to support pointer operations like dereferencing (`operator*`) and indirection (`operator ->`)
  - An object that looks and feels like something else is called a proxy object, or just proxy
- Smarter = do things that regular pointers don't
  - Such as memory management
- Simplest example: `auto_ptr`, included in the standard C++ library.
  - header: `<memory>`

```
template <class T> class auto_ptr {
    T* ptr;
 public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr()                 {delete ptr;}
    T& operator*()              {return *ptr;}
    T* operator->()             {return ptr;}
    // ...
};
```

Instead of:	We use:

<pre>void foo()
{
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}</pre>
	<pre>void foo()
	{
	    auto_ptr<MyClass> p(new MyClass);
	    p->DoSomething();
	
	}</pre>
	- `p` now cleans up after itself.

Why Use: Less Bugs

- Automatic cleanup: They clean after themselves, so there is no chance you will forget to deallocate
- Automatic initialization: You all know what a non-initialized pointer does; the default constructor now does the initialization to zero for you
- Stale pointers: As stated before, stale pointers are evil:

```
MyClass* p(new MyClass);
MyClass* q = p;
delete p;
// p->DoSomething();   // We don't do that, p is stale
p = 0;                 // we do this instead
q->DoSomething();      // Ouch, q is still stale!
```

  - Smart pointers can set their content to 0 once copied, e.g.

```
template <class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs) {
    if (this != &rhs) {
        delete ptr;   ptr = rhs.ptr;   rhs.ptr = 0;
    }
    return *this;                                    }
```

Stale Pointers (cont'd)

- The simplistic strategy to "change ownership" may not be suitable; other strategies can be implemented:
  - Deep copy the source into the target
  - Transfer ownership by letting `p` and `q` point to the same object but transfer the responsibility for cleaning up from `p` to `q`
  - Reference counting: count the references to the object and delete it only when the count reaches zero
  - Copy on write: use reference counting as long as the pointer is not modified, and just before it gets modified copy it and modify the copy
- Each strategy has advantages and disadvantages and are suitable for certain kind of applications

Why Use: Exception Safety

- Our old, simple example. . .

```
void foo() {
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

- . . . generates a memory leak (at best!)

Why Use: Exception Safety

- Our old, simple example...

```
void foo() {
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

- ...generates a memory leak (at best!) whenever `DoSomething` throws an exception
- We could of course take care of it by hand (pretty awkward; imagine now that you have some loops threw in for good measure):

```
void foo() {
    MyClass* p(new MyClass);
    try { p = new MyClass;   p->DoSomething();   delete p;   }
    catch (...) {delete p;   throw;   }
}
```

- With smart pointers we just let `p` clean up by itself.

Why Use: Garbage Collection, Efficiency

- C++ typically lacks garbage collection
- But this can be implemented using smart pointers
  - Simplest form of garbage collection: reference counting
  - Other, more sophisticated strategies can be implemented in the same spirit
  - Generally, garbage collection = reference counting + memory compaction

Why Use: Garbage Collection, Efficiency

- C++ typically lacks garbage collection
- But this can be implemented using smart pointers
  - Simplest form of garbage collection: reference counting
  - Other, more sophisticated strategies can be implemented in the same spirit
  - Generally, garbage collection = reference counting + memory compaction
- If the object pointed at does not change, there is no need to copy it; 'nuff said
  - Copying takes both time and space
  - Copy on write is our friend here
  - C++ strings are typically implemented in this manner

```
string s("Hello");
string t = s;        // t and s point to the same
                     // buffer of characters
t += " there!";      // a new buffer is allocated for t before
                     // appending, so that s is unchanged
```

Why Use: STL Containers

- STL containers (such as `vector`) store objects by value
  - So you cannot store objects of a derived type:

```
class Base { /*...*/ };
class Derived : public Base { /*...*/ };

Base b;   Derived d;
vector<Base> v;

v.push_back(b); // OK
v.push_back(d); // no cake
```

WHY USE: STL CONTAINERS

- STL containers (such as `vector`) store objects by value
  - So you cannot store objects of a derived type:

```
class Base { /*...*/ };
class Derived : public Base { /*...*/ };

Base b;   Derived d;
vector<Base> v;

v.push_back(b); // OK
v.push_back(d); // no cake
```

  - You go around this by using pointers:

```
vector<Base*> v;

v.push_back(new Base);       // OK
v.push_back(new Derived);    // OK
// obligatory cleanup, disappears when using smart pointers
for (vector<Base*>::iterator i = v.begin(); i != v.end(); ++i)
    delete *i;
```

How to Choose Your Smart Pointer

- The simplest smart pointer `auto_ptr` is probable suited for local variables
- A copied pointer is usually useful for class members
  - Think copy constructor and you think deep copy
- Due to their nature STL containers require garbage collected pointers (e.g., reference counting)
- Whenever you have big objects you are probably better off using copy on write pointers