C style pointers

Victor Eijkhout and Charlie Dey

spring 2017

Pointers and addresses

C and F pointers

Fortran has a clean pointer concept:
a pointer is an ‘alias’ that can be redirected

C/C++ has a very basic pointer concept:
a pointer is the address of some object
(including pointers)

Memory addresses

If you have an

```
int i;
```

then `&i` is the address of `i`.

An address is a (long) integer, denoting a memory address. Usually it is rendered in *hexadecimal* notation:

```
int i;
printf("address of i: %ld\n",(long)(&i));
printf(" same in hex: %x\n",(long)(&i));
```

Address types

The type of '&i' is int*, pronounced 'int-star', or more formally: 'pointer-to-int'.

You can create variables of this type:

```
int i;
int* addr = &i;
```

Star stuff

Equivalent:

- `int* addr`: `addr` is an int-star, or
- `int *addr`: `*addr` is an int.

Dereferencing

Using *addr 'dereferences' the pointer: gives the thing it points to; the value of what is in the memory location.

```
int i;
int* addr = &i;
i = 5;
cout << *addr;
i = 6;
cout << *addr;
```

This will print 5 and 6:

Array and pointer equivalence

Array and memory locations are largely the same:

```
double array[5];
double *addr_of_second = &(array[1]);
array = (11,22,33,44,55);
cout << *addr_of_second;
```

Pointer arithmetic

pointer arithmetic uses the size of the objects it points at:

```
double *addr_of_element = array;
cout << *addr_of_element;
addr_of_element = addr_of_element+1;
cout << *addr_of_element;
```

Increment add size of the array element, 4 or 8 bytes, not one!

Pointers and parameter passing

C++ pass by reference

C++ style functions that alter their arguments:

```
void inc(int &i) { i += 1; }
int main() {
  int i=1;
  inc(i);
  cout << i << endl;
  return 0;
}
```

C-style pass by reference

In C you can not pass-by-reference like this. Instead, you pass the address of the variable `i` by value:

```
void inc(int *i) { *i += 1; }
int main() {
  int i=1;
  inc(&i);
  cout << i << endl;
  return 0;
}
```

Now the function gets an argument that is a memory address: `i` is an int-star. It then increases `*i`, which is an int variable, by one.

Exercise 1

Write another version of the `swap` function:

```
void swap( /* something with i and j */ {
  /* your code */
}
int main() {
  int i=1,j=2;
  swap( /* something with i and j */ );
  cout << "check that i is 2: " << i << endl;
  cout << "check that j is 1: " << i << endl;
  return 0;
}
```

Dynamic allocation

Problem with static arrays

```
if ( something ) {
  double ar[25];
} else {
  double ar[26];
}
ar[0] = // there is no array!
```

Declaration and allocation

```
double *array;
if (something) {
  array = new double[25];
} else {
  array = new double[26];
}
```

De-allocation

Memory allocated with `new` does not disappear when you leave a scope. Therefore you have to delete the memory explicitly:

```
delete(array);
```

Allocation in C

```
int n;
double *array;
array = malloc( n*sizeof(double) );
if (!array)
  // allocation failed!
```