

# Pointers and references

Victor Eijkhout and Carrie Arnold and Charlie Dey

Fall 2017

# Pointers and addresses

# C and F pointers

C++ and Fortran have a clean reference/pointer concept: a reference or pointer is an 'alias' of the original object

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

If you're writing C++ you should not use it.  
if you write C, you'd better understand it.

# 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. C style:

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

and C++:

```
int i;  
cout << "address of i, decimal: " << (long)&i << endl;  
cout << "address of i, hex      : " << std::hex << &i << endl;
```

# 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;
```

# Dereferencing

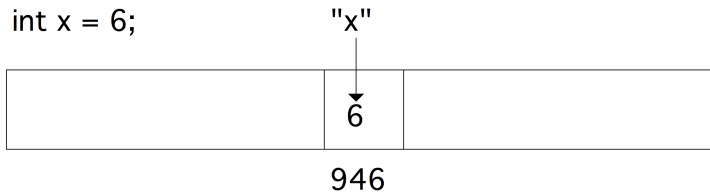
Using `*addr` 'dereferences' the pointer: gives the thing it points to; the value of what is in the memory location.

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

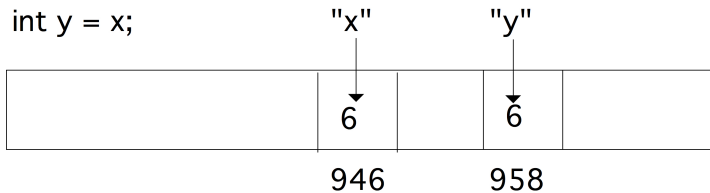
This will print 5 and 6:

## illustration

int x = 6;

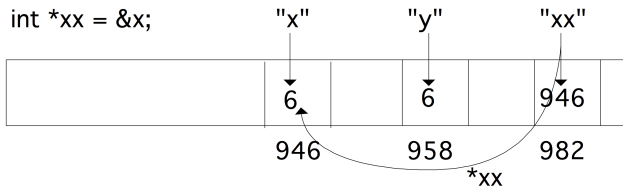


int y = x;

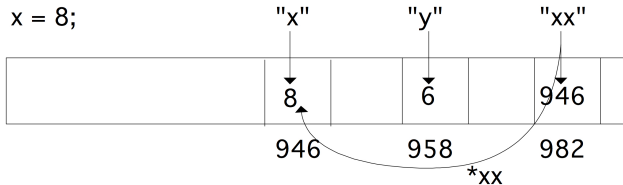


# illustration

int \*xx = &x;



x = 8;





# Star stuff

Equivalent:

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

# Arrays and pointers

# 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;
```

## Multi-dimensional arrays

# Multi-dimensional arrays

After

```
double x[10][20];
```

a row `x[3]` is a `double*`, so is `x` a `double**`?

Was it created as:

```
double **x = new double*[10];  
for (int i=0; i<10; i++)  
    x[i] = new double[20];
```

No: multi-d arrays are contiguous.

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

Hint: write C++ code, then insert stars where needed.

# Reference: change argument

```
void f( int &i ) { i += 1; };  
int main() {  
    int i = 2;  
    f(i); // makes it 3
```

# Reference: save on copying

```
class BigDude {  
private:  
    vector<double> array(5000000);  
}  
  
int main() {  
    BigDude big;  
    f(big); // whole thing is copied
```

Instead write:

```
void f( BigDude &thing ) { .... };
```

Prevent changes:

```
void f( const BigDude &thing ) { .... };
```

# 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);
```

The C++ `vector` does not have this problem, because it obeys scope rules.

# Memory leak1

```
void func() {  
    double *array = new double[large_number];  
    // code that uses array  
}  
int main() {  
    func();  
};
```

- The function allocates memory
- After the function ends, there is no way to get at that memory
- $\Rightarrow$  *memory leak*.



# Memory leak2

```
for (int i=0; i<large_num; i++) {  
    double *array = new double[1000];  
    // code that uses array  
}
```

Every iteration reserves memory, which is never released: another *memory leak*.

Your code will run out of memory!