C Pointers and parameter passing

Victor Eijkhout, Susan Lindsey

COE 322 Fall 2021
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.
1. 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:

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

Output

```
[pointer] printfpoint:
address of i: 140732781906948
same in hex: 7ffee7794c04
```

and C++:

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

Output

```
[pointer] coutpoint:
address of i, decimal : 140732704635892
address of i, hex : 0x7ffee2de3bf4
```
2. Address types

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

You can create variables of this type:

```c
int i;
int* addr = &i;
```
3. Dereferencing

Using \texttt{*addr} ‘dereferences’ the pointer: gives the thing it points to; the value of what is in the memory location.

Code:
\begin{verbatim}
int i;
int* addr = &i;
i = 5;
cout << *addr << endl;
i = 6;
cout << *addr << endl;
\end{verbatim}

Output
[pointer] cintpointer:
5
6
4. illustration

int x = 6;

"x"

\[ \begin{array}{c|c}
\hline
& 6 \\
\hline
946 & \\
\hline
\end{array} \]

int y = x;

"x"  "y"

\[ \begin{array}{c|c|c}
\hline
& 6 & 6 \\
\hline
946 & 958 & \\
\hline
\end{array} \]
5. illustration

```c
int *xx = &x;

x = 8;
```
6. Star stuff

Equivalent:

• \texttt{int* addr}: \texttt{addr} is an int-star, or

• \texttt{int *addr}: \texttt{*addr} is an int.
Addresses and parameter passing
7. C++ pass by reference

C++ style functions that alter their arguments:

```cpp
void inc(int &i) { i += 1; }
int main() {
    int i=1;
    inc(i);
    cout << i << endl;
    return 0;
}
```
8. 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:

```c
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:

```cpp
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;
}
```

Hint: write C++ code, then insert stars where needed.
Arrays and pointers
Array and pointer equivalence

Array and memory locations are largely the same:

Code:
```cpp
    double array[5] = {11,22,33,44,55};
    double *addr_of_second = &(array[1]);
    cout << *addr_of_second << endl;
    array[1] = 7.77;
    cout << *addr_of_second << endl;
```

Output

```
[pointer] arrayaddr:
22
7.77
```
10. Array and pointer equivalence

Array and memory locations are largely the same:

Code:

```cpp
double array[5] = {11, 22, 33, 44, 55};
double *addr_of_second = &(array[1]);
cout << *addr_of_second << endl;
array[1] = 7.77;
cout << *addr_of_second << endl;
```

Output
[pointer] arrayaddr:
22
7.77
Multi-dimensional arrays
11. Multi-dimensional arrays

After

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

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

Was it created as:

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

No: multi-d arrays are contiguous.
Dynamic allocation
12. Problem with static arrays

```c
if ( something ) {
    double ar[25];
} else {
    double ar[26];
}
ar[0] = // there is no array!
```
13. Declaration and allocation

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

(Size in doubles, not in bytes as in C)
14. De-allocation

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

\texttt{delete(array);}

The C++ \texttt{vector} does not have this problem, because it obeys scope rules.
15. Memory leak

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
• ⇒ memory leak.
16. Memory leaks

```cpp
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!