C Pointers and parameter passing

Victor Eijkhout, Susan Lindsey

Fall 2024
last formatted: December 3, 2024

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.
(until you get pretty advanced)
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.

Code:

```
// pointer/coutpoint.cpp
int i;
cout << "address of i, decimal: "
     << (long)&i << '\n';
cout << "address of i, hex     : "
     << std::hex << &i << '\n';
```

Output:

```
address of i,
    ↪decimal:
    ↪140732703427524
address of i, hex
    ↪:
    ↪0x7ffee2cbcbc4
```

2. Same in C

Using purely C:

Code:

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

Output:

```
address of i:
    ↪140732690693076
 same in hex:
    ↪7ffee2097bd4
```

3. 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;
// exactly the same:
int *addr = &i;
```

Now `addr` contains the memory address of `i`.

4. Dereferencing

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

Code:

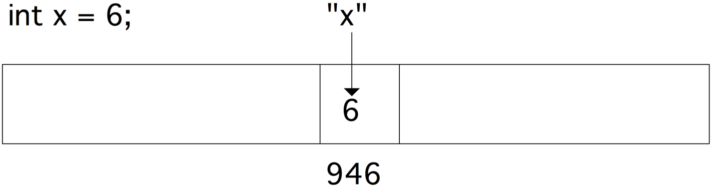
```
// pointer/cintpointer.cpp
int i;
int* addr = &i;
i = 5;
cout << *addr << '\n';
i = 6;
cout << *addr << '\n';
```

Output:

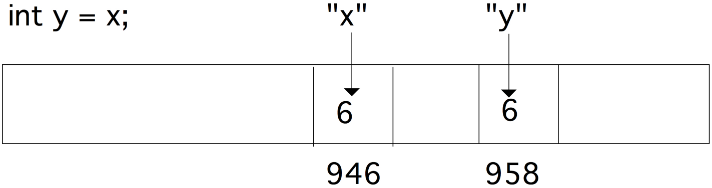
```
5
6
```

5. illustration

int x = 6;

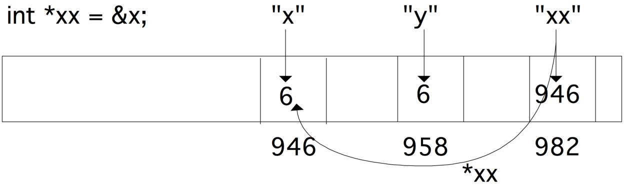


int y = x;

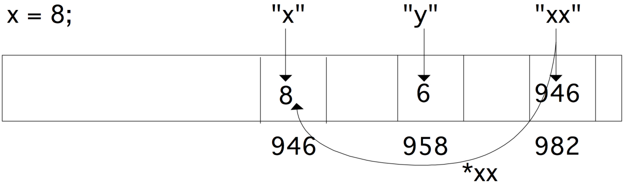


6. illustration

`int *xx = &x;`



`x = 8;`



7. Star stuff

Equivalent:

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

Addresses and parameter passing

8. 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;
}
```

9. 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;
}
```

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

Arrays and pointers

10. Array and pointer equivalence

Array and memory locations are largely the same:

Code:

```
// pointer/arrayaddr.cpp
double array[5] =
      {11,22,33,44,55};
double *addr_of_second =
      &(array[1]);
cout << *addr_of_second << '\n';
array[1] = 7.77;
cout << *addr_of_second << '\n';
```

Output:

```
22
7.77
```

11. Array passing to function

When an array is passed to a function, it behaves as an address:

Code:

```
// pointer/arraypass.cpp
void set_array( double *x,int
     size) {
  for (int i=0; i<size; ++i)
    x[i] = 1.41;
};
    /* ... */
  double array[5] =
     {11,22,33,44,55};
  set_array(array,5);
  cout << array[0] << "...." <<
     array[4] << '\n';
```

Output:

```
1.41....1.41
```

Note that these arrays don't know their size, so you need to pass it.

12. Size of an array

There is a `sizeof` function but beware:

Code:

```
// c/carray.c
void stat_f( int stat[] ) {
  printf(".. in function:
     %lu\n",sizeof(stat));
}
    /* ... */
  int stat[23];
  printf("Size of stat[23]:
     %lu\n",sizeof(stat));
  stat_f( stat );
```

Output:

```
Size of stat[23]: 92
.. in function: 8
```

(This is an example of pointer decay)

Multi-dimensional arrays

13. 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.

Dynamic allocation

14. Problem with static arrays

Create an array with size depending on something:

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

This Does Not Work

15. Declaration and allocation

Now dynamic allocation:

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

Don't forget:

```
delete array;
```

16. Allocation, C vs C++

C allocates in bytes:

```
double *array;
array  = (double*) malloc( 25*sizeof(double) );
```

C++ allocates an array:

```
double *array;
array  = new double[25];
```

Don't forget:

```
free(array); // C
delete array; // C++
```

17. De-allocation

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

```
free(array);
delete(array);
```

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

18. 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.

19. Memory leaks

```
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!