Pointers in C

Xiao Jia

Shanghai Jiao Tong University

November 22, 2011

Outline

- Overview of pointers
- Pointers in depth
- Q & A

* If you can't read this, move closer!

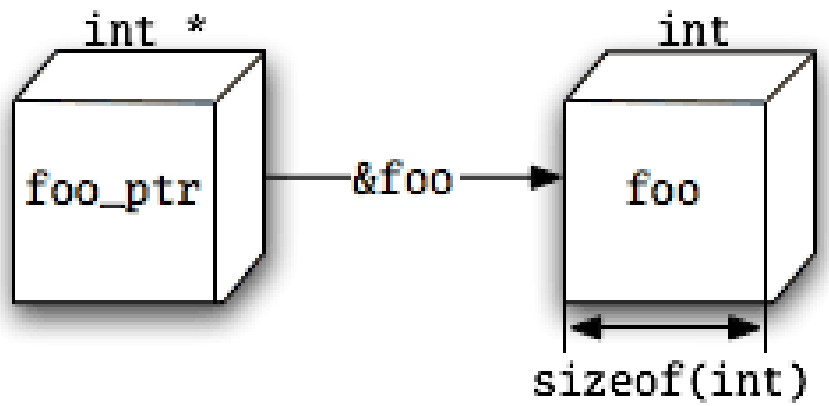
Overview of pointers

- Definition & declaration
- Assignment & dereferencing
- Arrays
- Pointer arithmetic
- Indexing
- Structures and unions
- Multiple indirection
- `const`
- Function pointers

Definition

A pointer is a memory address.

```
int foo;
int *foo_ptr = &foo;
```



Declaration

```
int* ptr_a, ptr_b;
```

```
int *ptr_a;
int  ptr_b;
```

```
int *ptr_a, ptr_b;
```

```
int ptr_b, *ptr_a;
```

Declaration

```
int *ptr_a, *ptr_b;
```

```
int ((not_a_pointer)), (*ptr_a), (((*ptr_b)));
```

This is useful for declaring function pointers (described later).

Assignment & dereferencing

```
int foo;
int *foo_ptr = &foo;


foo_ptr = 42;


int bar = *foo_ptr;


*foo_ptr = 42; //Sets foo to 42
```

Arrays

```
int array[] = { 45, 67, 89 };
```

The variable `array` is an extra-big box: three `int`s' worth of storage.

```
array == &array == &array[0]
```

- `array`
- pointer to `array`
- pointer to the first element of `array`

Pointer arithmetic

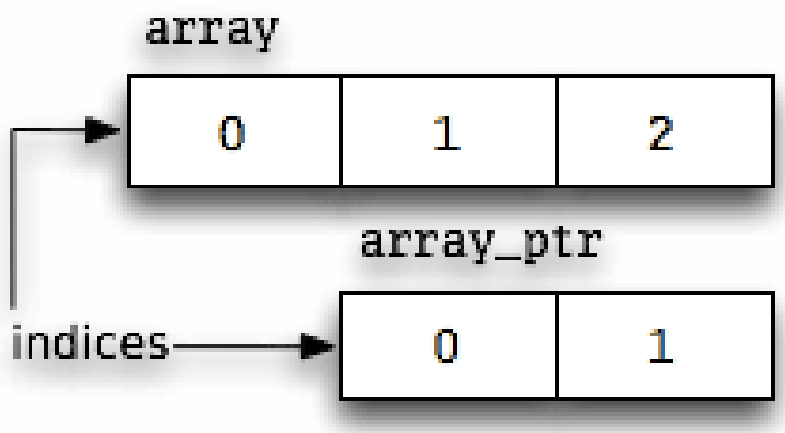
```
int *array_ptr = array;
printf(" first element: %d\n", *(array_ptr++));
printf("second element: %d\n", *(array_ptr++));
printf(" third element: %d\n", *array_ptr);
```

```
 first element: 45
second element: 67
 third element: 89
```

`int` pointers are incremented or decremented by `sizeof(int)` bytes. `void` pointers are incremented or decremented by 1 byte since `sizeof(void)` is illegal.

Indexing

```
int array[] = { 45, 67, 89 };
int *array_ptr = &array[1];
printf("%d\n", array[0]);       // 45
printf("%d\n", array_ptr[1]);   // 89
```



Structures and unions

```
struct foo {
  size_t   size;
  char     name[64];
  int      answer_to_ultimate_question;
  unsigned shoe_size;
};

struct foo my_foo;
my_foo.size = sizeof(struct foo);
```

Structures and unions

```
struct foo *foo_ptr = &my_foo;
(*foo_ptr).size = new_size;
foo_ptr->size = new_size;

struct foo **foo_ptr_ptr = &foo_ptr;
(*foo_ptr_ptr)->size = new_size;
(**foo_ptr_ptr).size = new_size;
```

Multiple indirection

```
int    a =  3;
int   *b = &a;
int  **c = &b;
int ***d = &c;
```

```
  *d ==   c
 **d ==  *c ==  b
***d == **c == *b == a == 3
```

const

```
const int *ptr_a;
int const *ptr_a;


int const *ptr_a;
    // int is const; cannot do *ptr_a = 42

int *const ptr_b;
    // can change *ptr_b; cannot do ptr_b++
```

Function pointers

Consider `strcpy`.

```
enum { str_length = 18U };
char src[str_length] = "This is a string.",
     dst[str_length];

strcpy(dst, src);
```

Declaring function pointers

```
char *strcpy(char *dst, const char *src);
    // Just for reference

char *(*strcpy_ptr)(char *dst, const char *src);
    // Pointer to strcpy-like function

strcpy_ptr =  strcpy;
strcpy_ptr = &strcpy;

//strcpy_ptr = &strcpy[0];
```

Parameter names are optional

```
char *(*strcpy_ptr_noparams)(char *, const char *) =
    strcpy_ptr;

strcpy_ptr =
    (char *(*)(char *, const char *))my_strcpy;

char *(**strcpy_ptr_ptr)(char *, const char *) =
    &strcpy_ptr;
```

Array of function-pointers

```
char *(*strcpies[])(char *, const char *) =
    { strcpy, strcpy, strcpy };

strcpies[0](dst, src);
```

Declaring exercise

Declare the following in a single line:

- a function `f` with no parameters returning an `int`
- a function `fip` with no parameter specification returning a pointer to an `int`
- a pointer `pfi` to a function with no parameter specification returning an `int`

(taken from C99 standard)

Declaring exercise

Declare the following in a single line:

- a function `f` with no parameters returning an `int`
- a function `fip` with no parameter specification returning a pointer to an `int`
- a pointer `pfi` to a function with no parameter specification returning an `int`

(taken from C99 standard)

```
int f(void), *fip(), (*pfi)();
```

Function returning a function pointer

```
char *(*get_strcpy_ptr(void))(char *dst,
                              const char *src);

strcpy_ptr = get_strcpy_ptr();
```

typedef

```
typedef char *(*strcpy_funcptr)(char *, const char *);

strcpy_funcptr strcpy_ptr = strcpy;
strcpy_funcptr get_strcpy_ptr(void);
```

Summary

- Declaring

```
void (*foo)(int);
```

- Initializing

```
void foo();
func_ptr = foo;
func_ptr = &foo;
```

- Invoking

```
func_ptr(arg1, arg2);
(*func_ptr)(arg1, arg2);
```

Pointers in depth

- What is a pointer?
- Pointer types and arrays
- Pointers and strings
- Pointers and structures
- Multi-dimensional arrays
- Dynamic allocation of memory
- When to use pointers?

What is a pointer?

```
int j, k, *ptr;
k = 2;
j = 7;
k = j;
ptr = &k;
*ptr = 7;
```

- What is a variable?
- What is an address?
- What is an object?

What is a pointer?

```
int j, k;
k = 2;
j = 7;
k = j;
```

- What is lvalue?
- What is rvalue?

Object & lvalue

```
int j, k;
k = 2;
j = 7;
k = j;
```

- An **object** is a named region of storage
- An **lvalue** is an expression referring to an object

Pointer types

```
int *ptr;
char *str;
double *dptr;
```

- What is the size of a pointer?

Pointer types

```
int *ptr;
*ptr = 2;
```

- What is the problem with the code above?

Pointer types

```
int *ptr, k;
ptr = &k;          // What is the value of ptr?
*ptr = 10;         // What is the value of k?
ptr++;
*ptr = 11;
```

- What is the problem with the code above?

Pointers and arrays

```
int my_array[] = {1, 23, 17, 4, -5, 100};
int *ptr;
ptr = &my_array[0];

ptr = my_array;

my_array = ptr;  // It's a named region of storage!
```

- What is the problem with the code above?
- What is the difference between `ptr` and `my_array`?

Pointers and strings

```
char my_string[40];
my_string[0] = 'A';
my_string[1] = 'c';
my_string[2] = 'm';
my_string[3] = '\0';

char my_string[40] = {'A', 'c', 'm', '\0'};

char my_string[40] = "Acm";

char *my_string = "Acm";

const char *my_string = "Acm";
```

Implementing strcpy

```
char *my_strcpy(char dest[], char src[]) {
  int i = 0;
  while (src[i] != '\0') {
    dest[i] = src[i];
    i++;
  }
  dest[i] = '\0';
  return dest;
}
```

Pointers and structures

```
struct Man      { int age;                    };
struct Superman { Man man_part; int power; };

void print_man(void *p) {
  cout << "Age: " << ((Man *)p)->age << endl;
}

void print_superman(void *p) {
  print_man(p);
  cout << "Power: " << ((Superman *)p)->power << endl;
}
```

Pointers and structures

```
struct Man       a = { 25     };
struct Superman b = { a, 250 };

print_man(&a);           // Age: 25
print_superman(&b);      // Age: 25
                         // Power: 250
b.man_part.age++;

print_man(&a);           // Age: 25
print_superman(&b);      // Age: 26
                         // Power: 250
```

Arrays of length zero

```
struct line {
  int  length;
  char contents[0];
};

struct line *this_line = (struct line *)
  malloc( sizeof(struct line) + this_length );

this_line->length = this_length;
strcpy(this_line->contents, this_contents);
```

Arrays of length zero

```
struct foo { int x; int y[]; };
struct bar { struct foo z;   };

struct foo a = { 1, {2, 3, 4} };          // Valid.
struct bar b = { { 1, {2, 3, 4} } };      // Invalid.
struct bar c = { { 1, {} } };             // Valid.
struct foo d[1] = { { 1, {2, 3, 4} } };   // Invalid.
```

Multi-dimensional arrays

```
int multi[5][10];

multi[row][col]

*(*(multi + row) + col)
// *(multi + row)  -> X
// *(X + col)
```

- `&multi == 100`
- `sizeof(int) == 4`
- `&multi[3][5] == ???`

Allocate & release an int

```
int *p = (int *) malloc(sizeof int);
*p = 100;
free(p);
```

```
int *p = new int;
*p = 100;
delete p;
```

Allocate & release a 1-dimension array

```
int *a, i;
a = (int *) malloc(10 * sizeof(int));
for (i = 0; i < 10; i++) {
  a[i] = i;
}
free(a);
```

```
int *a = new int[10];
for (int i = 0; i < 10; i++) {
  a[i] = i;
}
delete[] a;
```

Allocate a 2-dimension array

```
int **a = new int*[10];
for (int i = 0; i < 10; i++) {
  a[i] = new int[20];
  for (int j = 0; j < 20; j++) {
    a[i][j] = i + j;
  }
}
```

Release a 2-dimension array

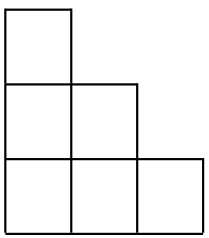
```
for (int i = 0; i < 10; i++) {
  delete[] a[i];
}
delete[] a;
```

Allocate a 3-dimension array

```
int ***a = new int**[10];
for (int i = 0; i < 10; i++) {
  a[i] = new int*[20];
  for (int j = 0; j < 20; j++) {
    a[i][j] = new int[30];
    for (int k = 0; k < 30; k++) {
      a[i][j][k] = i + j + k;
    }
  }
}
```

Allocate a fluctuated 2-dimension array

```
int **a = new int*[10];
for (int i = 0; i < 10; i++) {
  a[i] = new int[i + 1];
  for (int j = 0; j <= i; j++) {
    a[i][j] = i + j;
  }
}
```



When to use pointers?

- Indirect addressing
- Dynamic (run-time) addressing
- Polymorphism
- Pointers vs. references
  - Pointers may be `NULL`
  - References have to be valid (but may not if misused)
  - As parameters, small objects should behave like `int`s, e.g. `std::string`.
- Resource management
  - Must NOT have memory leaks
  - Acquiring and releasing tend to behave in a well-nested fashion
  - Across the borders of functions/methods, use smart pointers

Q & A

Thank you for listening!

Any questions?

References

- Ted Jensen. *A Tutorial on Pointers and Arrays in C*. Sept., 2003.
- Peter Hosey. *Everything you need to know about pointers in C*. Jan. 16, 2010.
- Cprogramming.com. *Function Pointers in C and C++*.