Pointers

Young W. Lim

2020-11-06 Fri

Outline

Based on

"Self-service Linux: Mastering the Art of Problem Determination", Mark Wilding
"Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

0. Pointer

- every pointer has a type
- every pointer has a value
- pointers are created with the & operator
- pointers are dereferenced with the * operator
- arrays and pointers are closely related
- pointers can also point to functions

1. Pointer Type

- Every pointer has a type
  - the data (object) type which a pointer points
  - pointer type : int *, double *
  - object tyupe : int, double
  - pointers : p, q
  - if the object type is T,
  - then the pointer type is *T
  - void * : a generic pointer
  - malloc returns void * pointer
  - must be type casted

2. Pointer Value

- the value is an address of the object
- which the pointer points to
- NULL value : the pointer does not point to anywhere

3. Pointer Creation with &

- & operator can be applied to any lvalue c expression
- which can appear on the left side of an asignment
- variables
- the elements of structures, unions, and arrays
- &a, &A[i], &S.i, &U->m

4. Pointer Dereference with *

- * operator returns a value of the object
- which the pointer points to

```
int * ip[2];
...
* ip
** ip
```

```
union uni { int t; char v; } u;
union uni *up;
...
up->v
```

5. Arrays and Pointers

- the name of an array can be referenced as a pointer variable
- but it is not lvalue and cannot be changed

```
a[3]
```

```
*(a+3)
```

6. Function Pointers (1)

- a powerful capability for storing and passing refereces to code
- which can be invoked in some other part of the program

```
void fun (int *xp) {
  void (*f) (int *) = fun;

  ...
  if (--(*xp)>0) f(xp);  // recursive call
}
```

6. Function Pointers (2)

```
08048414 <fun>:
08048414: 55              push %ebp
08048415: 89 e5           mov %esp, %ebp
08048417: 83 ec 1c        sub $0x1c, %esp
08048417: 57              push %edi
```

6. Function Pointers (3)

- it helps to read it starting form the inside (starting with f)
  and working outward
  thus, we see that f is a pointer, as indicated by (*f)
- it is a pointer to a function that has a single int * as
  an argument as indicated by (*f)(int *)
- finally we see that it is a pointer to a function
  that takes an int * as an argument and returns void

```
void (*f) (int *);

void *f(int *);

(void *) f(int *);
```

Pointer example code (1)

```
struct str { // example structures
  int t;
  char v;
}

union uni { // example unions
  int t;
  char v;
}

int g = 15;
```

Pointer example code (2)

```
void fun(int* xp)
{
  void (*f)(int *) = fun;  // f is a function pointer

  // allocation structure on stack
  struct str s = {1, 'a'}; // structure initialization

  // allocation union from heap
  union uni *up = (union uni *) malloc(sizeof(union uni));

  // locally declared array
  int *ip[2] = {xp, &g};
```

Pointer example code (3)

```
  up->v = s.v+1;

  printf("ip = %p, *ip = %p, **ip = %d \n",
          ip, *ip, **ip);

  printf("ip+1 = %p, ip[1] = %p, *ip[1] = %d \n",
          ip+1, ip[1], *ip[1]);

  printf("&s.v = %p s.v = '%c'\n", &s.v, s.v);

  printf("&up-> = %p, up->v = '$c'\n" &up->v, up->v);

  printf("f = %p \n", f);

  if (--(*xp) > 0) f(xp);  // recursive call of fun
}
```

Pointer example code (4)

```
int test()
{
  int x = 2;

  fun(&x);
  return x;
}
```

Pointer example code (5)

```
ip     = 0xbfffefa8,  *ip   = 0xbfffefe4,  **ip = 2
ip+1   = 0xbfffefac,  ip[1] = 0x804965c,   *ip[1] = 15
&s.v   = 0xbfffefb4,  s.v   = 'a'
&up->v = 08049760,    up->v = 'b'
f      = 0x804814
ip     = 0xbffef68,   *ip   = 0xbfffefe4,  **ip = 1
ip+1   = 0xbfffef6c   ip[1] = 0x804965c,   *ip[1] = 15
&s.v   = 0xbfffef74,  s.v   = 'a'
&up->v = 0x8049770,   up->v = 'b'
f      = 0x8048414
```

Pointer example code (6)

```
ip[0] = xp, *xp = x = 2
ip[1] = &g, g=15
s  in stack frame
up points to area in heap
f points to code for fun
ip in new frame, x = 1
ip[1] same as before
s in new frame
up points to new area in heap
f points to code for fun
```

Pointer example code (7)

- the function is executed twice
  - 1st by the direct call from `test`
  - 2nd by the indirect, recursive call
- those starting with `0xbfffef` point to locations on the stack, while the rest are part of the global storage (`0x804965c`), part of the executable code (`0x8048414`), or locations on the heap (`0x8049760` and `0x8049770`)

Pointer example code (8)

- array `ip` is instantiated twice
  once for each call to `fun`
- the second value (`0xbfffef68`) i smaller than
  the first value (`0xbfffefa8`), because
  the stack grows downward
- the contents of the array, however, are the same
  in both cases
- element 0 (`*ip`) is a pointer to variable `x`
  in the stack frame for `test`
- element 1 is a pointer to global variable `g`

Pointer example code (9)

- we can see that structure `s` is instantiated twice,
  both times on the stack
  while the union pointed to by variable `up`
  is allocated on the heap

Pointer example code (10)

- finally variable `f` is a pointer to function `fun`
- in the disassembled code,
  we find the following as the initial code for `fun`

```
08048414 <fun>:
 8048414 : 55                    push %ebp
 8048415 : 89 e5                 mov  %esp, %ebp
 8048417 : 83 ec 1c              sub  $0x1c, %esp
 804841a : 57                    push %edi
```

- the value `0x8048414` printed for pointer `f` is
  exactly the address of the first instruction
  in the code for `f`