Computer Programming

Pointers

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 6

Contents

Simplified memory model

- all code and data is stored in memory
- the standard unit for memory is byte
- we can view the memory as an array of bytes
- the address of a variable is defined as the position of its lowest byte in the memory
- usually, the lowest byte is stored at the lowest position (little endian)

Simplified memory model - example

int x = 1000000007;
= 0011 1011 1001 1010 1100 1010 0000 $0111_{(2)}$
= 0x 3B 9A CA 07

short s = -3000;
= 1111 0100 0100 $1000_{(2)}$
= 0x F448

char c = ‘a’;
= 97
= 0110 $0001_{(2)}$
= 0x 61

Address	Content
...	
304	0000 0111 = 07
305	1100 1010 = CA
306	1001 1010 = 9A
307	0011 1011 = 3B
...	
412	0110 0001 = 61
...	
512	0100 1000 = 48
513	1111 0100 = F4

Pointers - introduction

Role

- the role of a pointer is to hold a memory address
- the address is just a whole number indicating the position where a variable is located in the memory
- every pointer is linked to the type it points to
- pointer content is not controlled

Utility

- enables access to variables from different functions
- avoids copying of large variables (arrays, structures)

Dangerous

- we can access invalid memory zones
- we can interpret the data from a location erroneously

Pointers - declaration

- use the asterisk * symbol after a type

  `type* pointer_name`

- in this course we will adopt the style where we put the asterisk next to the type = we indicate that the asterisk modifies the type, not the variable
- examples:

```
int* px;
char* pc;
long long int* pll;
```

- we read: px is a pointer to int
- it is recommended to use suggestive names for pointers
  - possibly, start the name of each pointer with p

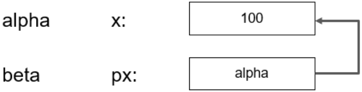
Pointers - initialization

- pointer can and should be initialized
- they should store memory addresses
- this behavior is not enforced by the compiler
- we can store any number in a pointer = not the intended usage
- to obtain the address of a variable we can use the & operator
- using uninitialized pointers can cause run-time errors

Pointers - the *address of* operator &

- the address of a variable can be obtained with the & operator
- this is a unary operator, used before a variable
- note, this symbol has many different meanings in different contexts (bitwise and, logical and if doubled)
- memory addresses can be printed with the %*p* format specifier

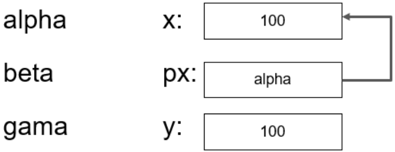
```c
int x;
int* px = &x;
printf("%p", px);
```



Pointers - the *dereference* operator *

- the * operator can be applied on a pointer to obtain the content form the memory location it's pointing to
- this is a unary operator, used before a pointer variable
- note, this symbol has many different meanings in different contexts (multiplication, pointer type modifier)

```c
int x;
int* px = &x;
int y = *px;
```



Pointers - the *dereference* operator *

- it is easy to confuse the dereference operator * and the type modifier *
- their role can only be deduced from context
  - when applied to a pointer variable like `*x` it returns the content
  - when applied after the type like `int*` it changes it into a pointer type
- this overloaded nature of the operator leads to confusion
- one can argue that the * in the type definition operates on the variable name

Frequent mistakes

- the * type modifier applies only to the closest variable
  `int* a, b;`
- if we want to declare multiple pointers, the * must be included before each like:
  `int *a, *b;`
- or we declare each of them individually, like this:
  `int* a; int* b;`
- a local pointer variable is uninitialized
- before accessing the location it points to, it must hold a valid memory address

```
int* p;
scanf("%d", p);
```

Program 6.1 - simple pointers

```c
1 #include <stdio.h>
2
3 int main(){
4   int x = 100;
5   int* px = &x;
6   int* px2 = &x;
7   *px2 = 50;
8   printf("%d\n", x);
9   int y = *px;
10  y /= 2;
11  printf("%d\n", x);
12  *px /= 2;
13  printf("%d\n", x);
14  px = NULL;
15  *px /= 2;
16  printf("%d\n", x);
17  return 0;
18 }
```

- both px and px2 point to the same location
- prints 50, 50, 25
- most likely, run-time error on line 15
- we attempted to dereference a NULL pointer

Pointers as input parameters and return types

Input parameter

- pointer can be used as input parameter types, like
  `void f(int* p)`
- they are sent by value during function call
- because of the dereferencing mechanism, they can be used to modify external data relative to the function

Return type

- returning pointer type from functions is allowed, like
  `char* f(void)`
- however, one should never return the address of a local variable
- after the function call ends, variables are deallocated

Program 6.2 - Interchange

```c
#include <stdio.h>

void inter(int* pa, int* pb){
  int* t = pa;
  pa = pb;
  pb = t;
}

int main(){
  int a = 1, b = 2;
  int* pa = &a;
  int* pb = &b;
  printf("%d %d\n", a, b);
  inter(pa, pb);
  printf("%d %d\n", a, b);
  return 0;
}
```

- the addresses of the variables are sent to the function
- the copies of these are swapped inside the function
- does not affect the variables from *main*

Program 6.3 - Correct interchange

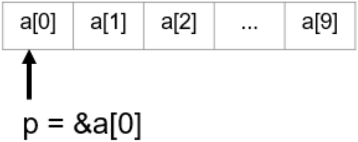
```c
#include <stdio.h>

void inter(int* pa, int* pb){
  int t = *pa;
  *pa = *pb;
  *pb = t;
}

int main(){
  int a = 1, b = 2;
  int* pa = &a;
  int* pb = &b;
  printf("%d %d\n", a, b);
  inter(pa, pb);
  printf("%d %d\n", a, b);
  return 0;
}
```

- the addresses of the variables are sent to the function
- the contents of the variables are changed
- note the usage of the dereference * operator
- works as intended

Pointers and arrays

- they are strongly related concepts
- the name of an array is automatically translated into the address of the first element
- more formally, the name of an array is a constant pointer to the first element
- we cannot change this pointer

```c
int a[10];
int* p = a;
int a0 = *p;
a = p;  //compile error
```



Pointers and arrays

- during function calls, arrays are sent by copying the address of the first element
- information about the array size is lost
- the following are equivalent:

  `void f(int a[10])`

  `void f(int a[])`

  `void f(int* a)`

- the last variant indicates the correct behavior
- for multidimensional arrays the array name is converted to a pointer to an array

```
int a[2][2];
int (*pa)[2] = a;
```

Operations on pointers

- increment/decrement
  - jump to the next/previous memory location
  - based on the size of the underlying type
  - always changes by a multiple of `sizeof(type)`

```c
double a[100];
double* p;
p=&a[10];
printf("%p\n", p);
p++;
printf("%p\n", p);
```

- sizeof(double) is 8
- 0028fc38, the address of 10th element
- 0028fc40 = 0028fc38 + 8 in hexadecimal

Operations on pointers

- adding/subtracting number n
  - changes the memory address by the n * `sizeof(type)`
  - this is equivalent to moving n positions forwards/backwards in an array

```
double a[100];
double* p;
p=&a[1];
printf("%p\n", p);
p = p - 1;
printf("%p\n", p);
p = p + 11;
printf("%p\n", p);
```

- sizeof(double) is 8
- 0028fc38, the address of element from position 1
- 0028fc30 = 0028fc38 - 8 in hexadecimal
- 0028fc88 = 0028fc30 + 11*8 in hexadecimal

Operations on pointers

- pointer difference
  - equal to the difference of the memory addresses stored in them divided by the size of the type
  - equivalent to the difference between the positions of the elements of the array
- pointers can be compared with the usual relational operators
- pointers cannot be added together (enforced by the compiler)

```
double a[100];
double* p = a+8;
double* q = a+10;
int dif = q-p;
printf("%p;%p;%d\n",
    p, q, dif);
printf("%d;%d\n",
    p > q, p+2 == q);
```

- p holds the address of the 8th element
- q holds the address of the 10th element
- prints 0028fc20;0028fc30;2
- prints 0;1

Pointer indexing

- the previous operators can be employed to access elements from an array
- in some cases it produces shorter and clearer code
- it is an alternative and equivalent syntax to using the [] operator

`a[i]` is equivalent to `*(a+i)`

`&a[i]` is equivalent to `a+i`

Program 6.4 - Pointer indexing

```c
#include <stdio.h>

int sum(int* a, int n){
  int s = 0;
  while(n--){
    s += *a;
    a++;
  }
  return s;
}

int main(){
  int a[] = {1, 2, 3};
  int n = sizeof(a) / sizeof(a[0]);
  int s = sum(a, n);
  printf("%d %d\n", s, *(a+n-1));
  return 0;
}
```

- function returns the sum of elements
- add the content from address a
- advance the pointer n times
- prints the sum and the last element

Constant pointers

- they can be declared by inserting the `const` keyword after the pointer type

  `type* const pointer_name = init;`
- similarly to constant variables, they must be initialized
- indicates that the pointer cannot change
- the values from the indicated memory zone can change
- the name of an array is a constant pointer

Pointers to constants

- they can be declared by inserting the `const` keyword before the pointer type

```
const type* pointer_name;
```

- indicates that the data pointed at cannot change
- the pointer can be changed
- string literal names are pointers to constants
- there exist constant pointers to constant data

```
const type* const pointer_name = init;
```

Pointers to void

- pointers can generic = unlinked from data types
- it is useful when we are only interested in the address
- such a pointer is compatible with a pointer to any data type

  `void* pointer_name;`
- we cannot directly dereference a void pointer since we do not know how to interpret the data
- we can cast the pointer to a specific type which can be dereferenced afterwards

```
float x = 0.5f;
void* p = &x;
float y = *((float*)p);
```

Program 6.5 - Reinterpreting memory zones

```c
#include <stdio.h>

int main(){
  int x =
      ('c' << 24)
    + ('p' << 16)
    + ('3' << 8 )
    + '<';
  char* pc = (char*) &x;
  printf("%c%c%c%c\n",
  pc[0], pc[1], pc[2], pc[3]);
  return 0;
}
```

- an int is represented on 4 bytes
- a char is represented on 1 byte
- interpret the 4 bytes of an int as 4 individual bytes of char
- print each byte as char
- little-endian order = least significant byte stored on lowest memory address