EPITA - Practical Programming

Security
System
Laboratory of Epita

C Programming - 02

Overview

1. And Then Came The Pointers !
    a. A Gentle Introduction To Pointers
    b. How And Where ?
    c. Arithmetic
2. Of Strings And Arrays
    a. Strings In C ?
    b. Arrays

And Then Came The Pointers !

LSE
Security System
Laboratory of Epita

Pointers ?

- Pointers are probably one of the most important concept in programming.
- As any expressive features, pointers are also a dangerous tool. Most software failures come from pointers missues.
- Unfortunately, pointers seem also to be the hardest concept to learn.

LSE Security System
Laboratory of Epita

A pointer is no more than an array index in the memory.

Memory

```c
int  *ptr;
ptr = p;
```

```c
*ptr = 42;
```

1	2	...	P	...	End
		...	42	...	

Basic Operations

- ➢ **Dereference:** access the memory cell
- ➢ **Reference:** retrieve the address of the left-value

```c
#include <stdio.h>

int main() {
  int   x = 0;
  int  *p;
  p = &x; // Reference
  *p = 42; // Dereference
```

```c
  printf("p = %p\n",p);
  printf("*p = %d", *p);
  printf("\nx = %d\n", x);
  return 0;
}
```

LSE Security System Laboratory of Epita

Basic Operations

```
> clang -Wall -Wextra -std=c99 -o ptr ptr.c
> ./ptr
p = 0x7fff131f5d08
*p = 42
x = 42
```

Errors ?

- **Invalid address dereference:** immediate failure (bus error, segmentation fault ...)
- **Data structure error:** lead to invalid behavior
- **Memory Corruption:** memory is (almost) uniformly accessible, you can thus modify any memory cell, leading to various kind of errors and security threats.

LSE Security System
Laboratory of Epita

Classical Mistakes

- Use after free
- Out-of-bound access
- Buffer overflow
- Dereferencing NULL pointers
- ...

LSE
Security System
Laboratory of Epita

Using Pointers

LSE
Security
System
Laboratory of Epita

Passing As References

```
void swap(int *a, int *b) {
    int                 c;
    c = *a;
    *a = *b;
    *b = c;
}
```

Passing As References

```c
void euclid(int a, int b, int *q, int *r) {
  *q = a/b;
  *r = a%b;
}
```

LSE Security System
Laboratory of Epita

Passing As Reference

```
unsigned qpower(unsigned a, unsigned b, unsigned *count) {
  if (!b) return 1;
  *count += 1;
  return (b%2 ? a : 1) * qpower(a*a, b>>1, count);
}
```

Arithmetic

Pointers Are Integer

- Addition and subtraction are allowed on pointer
- In the original B (and BCPL) model, pointers were indexes to words in memory and thus p+1 was the next value in memory.
- C preserves this semantics: p+1 is not the next byte but the next value, depending on the type of the pointed type.

Pointers Are Integer

```c
#include <stdio.h>
int main() {
  int                      i=0, *p1;
  char                     c='a', *p2;
  p1 = &i; p2 = &c;
  printf("p1\t: %p\n(p1+1)\t: %p\n",p1,p1+1);
  printf("p2\t: %p\n(p2+1)\t: %p\n",p2,p2+1);
  return 0;
}
```

```
p1      : 0x7fff77938bd8
(p1+1)  : 0x7fff77938bdc
p2      : 0x7fff77938bcf
(p2+1)  : 0x7fff77938bd0
```

LSE Security System Laboratory of Epita

Base Of Arithmetic

address	0	1	2	3	4	5	6	7
pointer	p				p+1			
int	42				43			

Beware ...

```c
#include <stdio.h>
int main() {
  int                    i=0, *p1;
  char                   c='a', *p2;
  p1 = &i; p2 = &c;
  printf("p1\t: %p\n(p2+1)\t: %p\n",p1,p2+1);
  *(p2+1) = 'b'; // b = 98
  printf("i\t: %d\n",i);
  return 0;
}
```

```
p1       : 0xffe61a44
(p2+1)   : 0xffe61a44
i        : 98
```

LSE Security System
Laboratory of Epita

More …

- byte by byte arithmetic requires pointer to char.
- void* can't be used in arithmetic.
- Operation between pointers of different kinds are not permitted.

Arrays

Pointers to Array

- An array is just a pointer
- It’s supposed to point on a memory area containing uniform data
- Arithmetics provides array’s cell access.

Arrays and Pointers

```c
int main() {
  int   t0[] = {0,1,2,3,4,5,6,7};
  int  *t1;
  t1 = t0;
  printf("t0: %p\nt1: %p\n", t0, t1);
  printf("*t0: %d\n*t1: %d\n", *t0, *t1);
  return 0;
}
```

Arrays and Arithmetic

`t[i]` and `*(t + i)`

- Array index is a shorthand
- Any pointer can be used as array
- Fun fact: `t[i] == i[t]`

Iterating

```c
int sum(int t[], size_t len) {
  int   r = 0;
  for (size_t i=0; i<len; ++i)
    r += t[i];
  return r;
}
```

```c
int sum(int t[], size_t len) {
  int   r = 0;
  int  *end = t + len;
  for (int *i = t; i != end; ++i)
    r += *i;
  return r;
}
```

LSE Security System Laboratory of Epita

Static or Dynamic ?

```
void statordyn(void) {
  int           t0[8];
  int          *t1;
  t1 = malloc(8 * sizeof (int));
  printf("sizeof (t0): %zu\n", sizeof (t0));
  printf("sizeof (t1): %zu\n", sizeof (t1));
}
```

LSE
Security System
Laboratory of Epita

Static or Dynamic

```
sizeof  (t0):  32
sizeof (t1): 8
```

- ➢ size is part of the static array type
- ➢ Dynamic array are seen as pointer (8bytes = 64bits)

LSE Security System
Laboratory of Epita

Static Array

```c
void staticarray(void) {
  int   t0[] = {0,1,2,3,4,5,6,7};
  printf("t0  : %p\n", t0);
  printf("&t0 : %p\n", &t0);
  int  *c = t0;
  for (;c != t0 + 8; ++c) {
    printf("c    : %p ", c);
    printf("- *c: %d\n", *c);
  }
}
```

```
t0   : 0x7fff55dec000
&t0 : 0x7fff55dec000
c    : 0x7fff55dec000 - *c: 0
c    : 0x7fff55dec004 - *c: 1
c    : 0x7fff55dec008 - *c: 2
c    : 0x7fff55dec00c - *c: 3
c    : 0x7fff55dec010 - *c: 4
c    : 0x7fff55dec014 - *c: 5
c    : 0x7fff55dec018 - *c: 6
c    : 0x7fff55dec01c - *c: 7
```

LSE Security System
Laboratory of Epita

array, &array and &array[0]

Static arrays:

- array is equivalent to &array[0] (almost)
- &array is a pointer an array (different type)
- sizeof (array) != sizeof (&array)

Dynamic arrays:

- array is equivalent to &array[0]
- &array is a pointer to a pointer
- sizeof (array) == sizeof (void*)

LSE Security System
Laboratory of Epita

More on static

- static arrays are not exactly pointers
- same behavior most of the time
- array variable can not be assigned
- Type for address of array:

```
int array[8];
int (*array)[8];
```

More Than One Dimensions ?

Static

- Fixed size
- *Real* multi-dimension array
- Specific type

Dynamic

- Array of arrays
- No regularity check
- Multiple allocations
- Dynamic size

LSE Security System

Laboratory of Epita

Static Matrix

```c
int mat[3][3] = {
  {1,2,3},
  {4,5,6},
  {7,8,9}
};

void print_mat(int mat[3][3]) {
  for (size_t i = 0; i < 3; ++i) {
    for (size_t j = 0; j < 3; ++j)
      printf("%d ", mat[i][j]);
    printf("\n");
  }
}
```

- Fixed Size
- Size part of the type
- No equivalent dyn types

LSE Security System
Laboratory of Epita

Dynamic Matrix

```
int** build_matrix(size_t dim) {
  int          **mat;
  mat = malloc(dim * sizeof (int *));
  for (size_t i = 0; i < dim; ++i)
    mat[i] = malloc(dim * sizeof (int));
  return mat;
}
void print_dynmat(int **mat, size_t dim) {
  for (size_t i = 0; i < dim; ++i) {
    for (size_t j = 0; j < dim; ++j)
      printf("%02d ", mat[i][j]);
    printf("\n");
  }
}
```

- Dynamic size
- Array of pointers
- May not be uniform

LSE Security System
Laboratory of Epita

One Used as Many

- Use 1-D array as n-D array
- Translate index manually
- Only one allocation
- Can use static or dynamic arrays

One Used as Many

```
int* build_lin_matrix(size_t dim) {
  int          *mat;
  mat = malloc(dim * dim * sizeof (int));
  for (size_t i=0; i < dim*dim; ++i)
    mat[i] = i + 1;
  return mat;
}

void print_linmat(int *mat, size_t dim) {
  for (size_t i = 0; i < dim; ++i) {
    for (size_t j = 0; j < dim; ++j)
      printf("%02d ", mat[i * dim + j]);
    printf("\n");
  }
}
```

Of Strings And Arrays

LSE
Security System
Laboratory of Epita

Characters

- Characters are simply 8bits (1 byte) integer (signed or not, depending on architecture): `char`
- Each char can be used as an integer or as a character.
- Character literals (a single character between single quote) are transformed into their ASCII value at compile time.
- Since characters are treated as integer, you have nice properties like: `'a' + 1 == 'b'`

LSE Security System
Laboratory of Epita

Letters …

```c
#include <stdio.h>

int main()
{
  for (unsigned i=0; i<13; ++i) {
    printf("'%c' == %u\t",'a' + i, 'a' + i);
    printf("'%c' == %u\n",'a'+13+i, 'a'+13+i);
  }
  return 0;
}
```

Letters …

```
'a' == 97  'n' == 110
'b' == 98  'o' == 111
'c' == 99  'p' == 112
'd' == 100 'q' == 113
'e' == 101 'r' == 114
'f' == 102 's' == 115
'g' == 103 't' == 116
'h' == 104 'u' == 117
'i' == 105 'v' == 118
'j' == 106 'w' == 119
'k' == 107 'x' == 120
'l' == 108 'y' == 121
'm' == 109 'z' == 122
```

Pointer To Char

- Representation of characters strings was one of the issue in B.
- The choice made is somehow confusing: there's no dedicated type for strings, they are simply arrays of characters, but, there's a syntax for string literals.
- In practice: strings are array of characters with a marker (the null character) at the end. String literals are built at compile time and replaced with the corresponding pointer.

LSE Security System
Laboratory of Epita

Array Of Char

```c
int main() {
  char                     *s = "a string";
  printf("s = %p\ns = \"%s\"\n",s,s);
  for (size_t i=0; i<9; ++i)
    printf("| %u ", s[i]);
  printf("|\n");
  return 0;
}
```

```
s = 0x400702
= "a string"
97 | 32 | 115 | 116 | 114 | 105 | 110 | 103 | 0 |
```

LSE Security System Laboratory of Epita