Programming in C

Based on the Original Slides from
Politehnica International- Computer Engineering
Lecture Slides
Outline

• Pointers
  – Pointers and Addresses
  – Pointers and Function Arguments
  – Pointers and Arrays
  – Pointer Arithmetics
  – Pointers and strings
  – Dynamic memory allocation
  – Pointer arrays. Pointers to pointers
  – Multidimensional arrays and pointers
  – Structures and pointers
Pointers and addresses

- A pointer is a variable whose value is a memory address
  - `int count = 10;`
  - `int *int_pointer;`
  - `int_pointer = &count;`
- The **address operator** has the effect of assigning to the variable `int_pointer`, not the value of `count`, but a **pointer** to the variable `count`.
- We say that `int_ptr"points to" count`
- The values and the format of the numbers representing memory addresses depend on the computer architecture and operating system. In order to have a portable way of representing memory addresses, we need a different type than integer!
- To print addresses: `%p`
Lvalues and Rvalues

- There are two “values” associated with any variable:
  - An "lvalue" (left value) of a variable is the value of its address, where it is stored in memory.
  - The "rvalue" (right value) of a variable is the value stored in that variable (at that address).

- The lvalue is the value permitted on the left side of the assignment operator '==' (the address where the result of evaluation of the right side will be stored).

- The rvalue is that which is on the right side of the assignment statement

- a=a+1
Declaring pointer variables

type * variable_name;

- it is not enough to say that a variable is a pointer. You also have to specify the type of variable to which the pointer points!
  - int * p1; // p1 points to an integer
  - float * p2; // p2 points to a float

- Exception: generic pointers (void *) indicate that the pointed data type is unknown
  - may be used with explicit type cast to any type (type *)
  - void * p;
Indirection (dereferencing) operator *

- To reference the contents of `count` through the pointer variable `int_pointer`, you use the **indirection operator**, which is the asterisk `*` as an unary prefix operator. `*int_pointer`
- If a pointer variable `p` has the type `t *`, then the expression `*p` has the type `t`

```
// Program to illustrate pointers
#include <stdio.h>
int main (void)
{
    int count = 10, x;
    int *int_pointer;
    int_pointer = &count;
    x = *int_pointer;
    printf (“count = %i, x = %i
”, count, x);
    return 0;
}
```
Example: pointers

// Program to illustrate pointers
#include <stdio.h>
int main (void)
{
    int count = 10;
    int *ip;
    ip = &count;
    printf ("count = %i, *ip = %i\n", count, *ip);
    *ip = 4;
    printf ("count = %i, *ip = %i\n", count, *ip);
    return 0;
}
Using pointer variables

• The value of a pointer in C is meaningless until it is set pointing to something!

```c
int *p;
*p = 4;
```

Severe runtime error !!! the value 4 is stored in the location to which p points. But p, being uninitialized, has a random value, so we cannot know where the 4 will be stored!

• How to set pointer values:
  – Using the address operator

```c
int *p;
int x;
p = &x;
*p = 4;
```

– Using directly assignments between pointer variables

```c
int *p;
int *p1;
int x;
p1 = &x;
p = p1;
*p = 4;
```
NULL pointers

• Values of a pointer variable:
  – Usually the value of a pointer variable is a pointer to some other variable
  – Another value a pointer may have: it may be set to a null pointer
• A null pointer is a special pointer value that is known not to point anywhere.
• No other valid pointer, to any other variable, will ever compare equal to a null pointer!
• Predefined constant NULL, defined in <stdio.h>
• Good practice: test for a null pointer before inspecting the value pointed!

```c
#include <stdio.h>

int *ip = NULL;

if(ip != NULL)  printf("%d\n", *ip);
if(ip )  printf("%d\n", *ip);
```
**const and pointers**

- **With pointers, there are two things to consider:**
  - whether the pointer will be changed
  - whether the value that the pointer points to will be changed.
- Assume the following declarations:
  ```
  char c = 'X';
  char *charPtr = &c;
  ```
- The pointer variable charPtr is set pointing to the variable c.
- If the pointer variable is always set pointing to c, it can be declared as a const pointer as follows:
  ```
  char * const charPtr = &c;
  ```
- If the location pointed to by charPtr will not change through the pointer variable `charPtr`, that can be noted with a declaration as follows:
  ```
  const char *charPtr = &c;
  ```
- ```
  charPtr = &d;   // not valid !!!
  *charPtr = 'Y'; // not valid !!!
  ```
Pointers and Function Arguments

- Recall that the C language passes arguments to functions by value (except arrays).
- There is no direct way for the called function to alter a variable in the calling function.

```c
void swap(int x, int y) /* WRONG */
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

swap(a, b); // Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above swaps copies of a and b.
```
Pointers and Function Arguments

- If it is necessary that a function alters its arguments, the caller can pass pointers to the values to be changed.
- Pointer arguments enable a function to access and change variables in the function that called it.

```c
void swap(int *px, int *py) /* interchange *px and *py */{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int a=3, b=5;
swap(&a, &b);
```
Pointers and arrays

• In C, there is a strong relationship between pointers and arrays
• Any operation that can be achieved by array subscripting can also be done with pointers

```c
int a[10];

int *pa;
pa=&a[0];
Or
pa=a;
```

The value of a variable of type array is the address of element zero of the array.
The name of an array is a synonym for the location of the initial element.
Pointers and Arrays

- If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before.
- If pa points to a[0], *(pa+1) refers to the contents of a[1], pa+i is the address of a[i], and *(pa+i) is the contents of a[i].
- The value in a[i] can also be written as *(a+i). The address &a[i] and a+i are also identical.
- These remarks are true regardless of the type or size of the variables in the array a!
Arrays are **constant** pointers

The name of an array is a CONSTANT having as a value the location of the first element.
You cannot change the address where the array is stored!
An array's name is equivalent with a **constant** pointer

**OK.** Pointers are variables that can be assigned or incremented

**Syntax errors !!!**

Arrays as parameters

• When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.

• As formal parameters in a function definition, char s[] and char *s are equivalent; The latter is preferred because it says more explicitly that the variable is a pointer.

• It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray.

• For example, if a is an array, f(&a[2]) and f(a+2) both pass to the function f the address of the subarray that starts at a[2]. Within f, the parameter declaration can read

  • f(int arr[]) { ... }

  or

  • f(int *arr) { ... }
Example: Arrays as parameters

```c
void print1(int tab[], int N) {
    int i;
    for (i=0; i<N; i++)
        printf("%d ",tab[i]);
}

void print2(int tab[],int N) {
    int * ptr;
    for (ptr=tab; ptr<tab+N; ptr++)
        printf("%d ", *ptr);
}

void print3(int *tab,int N) {
    int * ptr;
    for (ptr=tab; ptr<tab+N; ptr++)
        printf("%d ", *ptr);
}

void print4(int *tab,int N) {
    int i;
    for (i=0; i<N; i++, tab++)
        printf("%d ", *tab);
}

void main(void) {
    int a[5]={1,2,3,4,5};
    print1(a,5);
    print2(a,5);
    print3(a,5);
    print4(a,5);
}
```

**The formal parameter can be declared as array or pointer!**
**In the body of the function, the array elements can be accessed through indexes or pointers!**
Example: Arrays as parameters

/* strlen: return length of string s */
int strlen(char *s)
{
    int n;
    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}

char array[100]="Hello, world";
char *ptr="Hello, world";

strlen("Hello, world"); /* string constant */
strlen(array); /* char array[100]; */
strlen(ptr); /* char *ptr; */
Example: Arrays as parameters

```c
int strlen(char *s)
{
    if (*s=='\0')
        return 0;
    else
        return 1 + strlen(++s);
}
```

The recursive call gets as parameter the subarray starting with the second element.
Pointer arithmetic

• All operations on pointers take into account the size of the pointed type (sizeof(T))!

• *Valid pointer operations:*
  – Assignment between pointers of the same type
  – Addition/ subtraction between a pointer and an integer
  – Comparison between two pointers that point to elements of the same array
  – Subtraction between two pointers that point to elements of the same array
  – Assignment or comparison with zero (NULL)
Pointer arithmetic

• **Increment/decrement:** if \( p \) is a pointer to type \( T \), \( p++ \) increases the value of \( p \) by \( \text{sizeof}(T) \) (\( \text{sizeof}(T) \) is the amount of storage needed for an object of type \( T \)). Similarly, \( p-- \) decreases \( p \) by \( \text{sizeof}(T) \);

```c
T tab[N];
T * p;
int i;
p=&tab[i];
p++; // p contains the address of tab[i+1];
```

• **Addition/subtraction with an integer:** if \( p \) is a pointer to type \( T \) and \( n \) an integer, \( p+n \) increases the value of \( p \) by \( n*\text{sizeof}(T) \). Similarly, \( p-n \) decreases \( p \) by \( n*\text{sizeof}(T) \);

```c
T tab[N];
T * p;
p=tab;
p=p+n; // p contains the address of tab[n].
```
Pointer arithmetic

• **Comparison of two pointers.**
• *If p and q point to members of the same array*, then relations like `==`, `!=`, `<`, `>`, `>=`, etc., work properly.
  – For example, `p < q` is true if `p` points to an earlier element of the array than `q` does.
• Any pointer can be meaningfully compared for equality or inequality with zero.
• **Pointer subtraction**:  
• *if p and q point to elements of the same array*, and `p<q`, then `q-p+1` is the number of elements from `p` to `q` inclusive.

• The behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array.
Example: pointer subtraction

```c
/* strlen: return length of string s */
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return p - s;
}
```
Example: middle of array

• The initializers for low and high are now pointers to the beginning and just past the end of the table.
• The computation of the middle element can no longer be simply
  • mid = (low+high) / 2    /* WRONG */
• because the addition of pointers is illegal. Subtraction is legal, however, so high-low is the number of elements, and thus
  • mid = low + (high-low) / 2
• sets mid to the element halfway between low and high.
Character pointers

- A string constant: "I am a string"
- is an array of characters. In the internal representation, the array is terminated with the null character '\0' so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.
- a string constant is accessed by a pointer to its first element
- char *pmessage;
- pmessage = "now is the time";
- assigns to pmessage a pointer to the character array. This is not a string copy; only pointers are involved!
Character pointers

char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */

- amessage is an array, just big enough to hold the sequence of characters and '\0' that initializes it. Individual characters within the array may be changed but amessage will always refer to the same storage.
- pmessage is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.
Precedence of operators

- \*p++ increments p after fetching the character that p points to
  char *p=“hello” ;
  printf(“%c”,*p++); // displays h

- *++p increments p before fetching the character that p points to
  char *p=“hello” ;
  printf(“%c”,*++p); // displays e
Example: Character pointers and functions

*/ strcpy: copy t to s; array subscript version */
void strcpy(char s[], char t[])
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}

/* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
    while ((s = t) != '\0')
    {
        s++;
        t++;
    }
}
Example: Character pointers and functions

/* strcpy: copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
    
}

/* strcpy: copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
    
}
Dynamic memory allocation

• Whenever you define a variable in C you are reserving one or more locations in the computer’s memory to contain the values that will be stored in that variable. The C compiler automatically allocates the correct amount of storage for you.

• It is frequently desirable to be able to dynamically allocate storage while a program is running:

• Suppose you have a program that is designed to read in a set of data from input (standard input or file) into an array in memory. Suppose, however, that you don’t know how much data is in the file until the program starts execution. You have three choices:
  – Define the array to contain the maximum number of possible elements at compile time.
  – Use a variable-length array to dimension the size of the array at runtime.
  – Allocate the array dynamically using one of C’s memory allocation routines.
malloc()

- `<stdlib.h>`
- `void * malloc(int n);`
- `malloc` allocates `n` bytes of memory and returns a pointer to them if the allocation was successful, NULL otherwise
- The pointer returned by `malloc` is of the generic type `void *`; it has to be converted to a concrete pointer type

```c
#include <stdlib.h>
char *line;
int linelen = 100;
line = (char *) malloc(linelen);
/* incomplete -- malloc's return value not checked */
getline(line, linelen);
```
Examples: malloc

```c
char *somestring, *copy;
...
copy = (char *) malloc(strlen(somestring) + 1);
/* incomplete -- malloc's return value not checked */
strcpy(copy, somestring);

int *ip = (int *) malloc(100 * sizeof(int));
```
Checking what malloc returns!

• When malloc is unable to allocate the requested memory, it returns a null pointer. Therefore, whenever you call malloc, it's vital to check the returned pointer before using it!

```c
int *ip = (int *)malloc(100 * sizeof(int));
if(ip == NULL) {
    printf("out of memory\n");
    return; // exits current function
}
```

```c
int *ip = (int *) malloc(100 * sizeof(int));
if(ip == NULL) {
    printf("out of memory\n");
    exit(1); // exits all nested function calls, terminates program
}
```
Dynamic allocation of arrays

- Dynamic allocation of an array of N elements of type TIP:
  - TIP * p;
  - p = (TIP *) malloc(N*sizeof(TIP));
- Pointer p will point to a memory block big enough to hold N elements of type TIP.
- Variable p can be used in the same way as if it was declared:
  - TIP p[N];
Example: dynamic allocation of arrays

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int n;
    int *tab;
    int i;
    printf("Input number of elements: \n");
    scanf("%d", &n);
    if ((tab=(int *)malloc(n * sizeof(int)))==NULL) {
        printf("Memory allocation error !\n");
        exit(1);
    }
    for (i=0; i<n; i++)
        scanf("%d", &tab[i]);
    for (i=0; i<n; i++)
        printf("%d ", tab[i]);
    free(tab);
    return 1;
}
```
Example: dynamic allocation of character strings

void main(void)
{
    char sir1[40];
    char *sir2;
    printf("Enter a string: \n");
    scanf("%40s", sir1);
    if ((sir2=(char *)malloc(strlen(sir1)+1))==NULL) {
        printf("memory allocation error !\n");
        exit(1);
    }
    strcpy(sir2, sir1);
    printf("The copy is: %s \n", sir2);
    ..........
}
Lifetime of dynamic allocated memory

- Memory allocated with malloc lasts as long as you want it to. It does not automatically disappear when a function returns, as automatic variables do:

```c
void fct(void) {
    int *p;
    p=(int*) malloc(10*sizeof(int));
    return;
}
```

The memory area allocated here remains occupied also after the function call is ended! Only the pointer variable p accessing it disappears.
Lifetime example

• Have a function create and return an array of values:

```c
int * fct(void) {
    int *p;
    p=(int*) malloc(10*sizeof(int));
    // fill p with values . . .
    return p;
}
```

```c
int * fct(void) {
    int p[10];
    // fill p with values . . .
    return p;
}
```
free()

- Dynamically allocated memory is deallocated with the free function. If p contains a pointer previously returned by malloc, you can call
  
  ```c
  free(p);
  ```

- which will ``give the memory back'' to the heap of memory from which malloc requests are satisfied.

- the memory you give back by calling free() is immediately usable by other parts of your program. (Theoretically, it may even be usable by other programs.)

- When your program exits, any memory which it has allocated but not freed should be automatically released by the operating system.

- Once you've freed some memory you must remember not to use it any more. After calling \texttt{free(p)} it is probably the case that p still points at the same memory. However, since we've given it back, it's now ``available,'' and a later call to \texttt{malloc} might give that memory to some other part of your program.
Pointer arrays. Pointers to Pointers

• Since pointers are variables themselves, they can be stored in arrays just as other variables can.
• Example: a program that will sort a set of text lines into alphabetic order
• The sorting algorithms will work, except that now we have to deal with lines of text, which are of different lengths, and which, unlike integers, can't be compared or moved in a single operation.
• We need a data representation that will cope efficiently and conveniently with variable-length text lines.
Sorting an array of pointers

- Each line can be accessed by a pointer to its first character. The pointers themselves can be stored in an array.
- Two lines can be compared by passing their pointers to `strcmp`.
- When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.
- This eliminates the twin problems of complicated storage management and high overhead that would go with moving the lines themselves.

```
    defghi
    jklmnopqrs
t    abc
```

```
    defghi
t    jklmnopqrs
t    abc
```
Example: main program

#include <stdio.h>
#include <string.h>

#define MAXLINES 5000   /* max #lines to be sorted */
char *lineptr[MAXLINES];  /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void sort(char *lineptr[], int nlines);

int main(void) {
    int nlines;   /* number of input lines read */
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        sort(lineptr, nlines);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}
Example: readlines

```c
#define MAXLEN 1000 /* max length of any input line */
int getline(char *, int);

/* readlines: read input lines */
int readlines(char *lineptr[], int maxlines) {   
    int len, nlines;
    char *p, line[MAXLEN];
    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0) 
        if (nlines >= maxlines ||
            ((p = (char *)malloc(len)) == NULL))
            return -1;
        else {
            line[len-1] = '\0'; /* delete newline */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}
```
Example: getline

/* getline:  read a line into s, return length  */
int getline(char s[],int lim)
{
    int c, i;
    for (i=0; i < lim-1 &&
         (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}
Example: writelines

/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    int i;
    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
Example: sort

void sort (char* v[], int n)
{
    int i, j;
    char *temp;
    for ( i = 0; i < n - 1; ++i )
        for ( j = i + 1; j < n; ++j )
            if ( strcmp(v[i], v[j]) > 0 ) {
                /* swaps only pointers to strings */
                temp = v[i];
                v[i] = v[j];
                v[j] = temp;
            }
}
Pointers vs Multidimensional arrays

- int a[10][20];
- int *b[10];
- a[3][4] and b[3][4] are both syntactically legal references to a single int.
- a is a true two-dimensional array: 200 int-sized locations have been set aside, and the conventional rectangular subscript calculation 20 * row + col is used to find the element a[row, col].
- For b, however, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either statically or with code. Assuming that each element of b does point to a twenty-element array, then there will be 200 ints set aside, plus ten cells for the pointers. The important advantage of the pointer array is that the rows of the array may be of different lengths. That is, each element of b need not point to a 20 element vector; some may point to two elements, some to fifty, and some to none at all.
Structures and pointers

• Structure pointers are just like pointers to ordinary variables. The declaration
  
  ```c
  struct point *pp;
  ```

  says that pp is a pointer to a structure of type struct point. If pp points to a point structure, *pp is the structure, and (*pp).x and (*pp).y are the members.

• To use pp:
  
  ```c
  struct point origin, *pp;
  pp = &origin;
  printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
  ```

• The parentheses are necessary in (*pp).x because the precedence of the structure member operator . is higher then *. The expression *pp.x means *(pp.x), which is illegal here because x is not a pointer.
Pointers to structures

- Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If \( p \) is a pointer to a structure, then
  - \( p->\text{member-of-structure} \)
  - refers to the particular member. So we could write instead
    
    ```
    printf("origin is (%d,%d)\n", pp->x, pp->y);
    ```

- Both \( . \) and \( \rightarrow \) associate from left to right, so if we have
  - ```
    struct rect r, *rp = &r;
    ```
  - then these four expressions are equivalent:
    - \( r.pt1.x \)
    - \( rp->pt1.x \)
    - \( (r.pt1).x \)
    - \( (rp->pt1).x \)
Precedence of operators

- The structure operators . and ->, together with () for function calls and [] for subscripts, are at the top of the precedence hierarchy and thus bind very tightly.
- For example, given the declaration
  ```c
  struct {
    int len;
    char *str;
  } *p;
  ```
  then `++p->len` increments `len`, not `p`, because the implied parenthesization is `+(p->len)`.
- Parentheses can be used to alter binding: `(++p)->len` increments `p` before accessing `len`, and `(p++)->len` increments `p` afterward. (This last set of parentheses is unnecessary.
- `*p->str` fetches whatever `str` points to;
- `*p->str++` increments `str` after accessing whatever it points to (just like `*s++`)
- `(*p->str)++` increments whatever `str` points to
- `*p++->str` increments `p` after accessing whatever `str` points to.