Applied Programming and Computer Science, DD2325/appcs15

PODF, Programmering och datalogi för fysiker, DA7011

Autumn 2015

Lecture 4, C programming: pointer, array, struct, list

A. Maki, C. Edlund

Pointer variables

Pointer variables are intended for pointing at locations in memory rather than storing values.

They can point to any objects of a specific type, including basic types, arrays, pointers and functions.

Declaration: `type *x;`

Initially *x* points to *NULL* (=0, no address). *x* can point to any variable of correct type.

The object pointed to can be accessed by the (indirection) dereference operator $*$.

The address can be obtained by the address operator &.

Pointer variables (cont.)

```
#include <stdio.h>

int main() {
  int alpha;
  int *beta;

  alpha = 1;
  beta = &alpha;
  printf("The value %d is stored at addr %u.\n", alpha, &alpha);
  printf("The value %d is stored at addr %u.\n", *beta, beta);
  printf("The value %d is stored at addr %u.\n", beta, &beta);
}

Output:
% gcc -o test test.c
% ./test
The value 1 is stored at addr 1584483176.
The value 1 is stored at addr 1584483176.
The value 1584483176 is stored at addr 1584483168.
%
```

Pointer variables (cont.)

```
#include <stdio.h>

void print(int i, int j, int *k) {
  printf("%d  %d  %d\n", i, j, *k);
}

int main() {
  int i=0, j=1, *k;

  print(i,j,k);  /* Run-time Error; k = NULL */
  k  = &i;
  print(i,j,k);  /* 0 1 0 */
  k  = &j;
  print(i,j,k);  /* 0 1 1 */
  j  = 2;
  print(i,j,k);  /* 0 2 2 */
  *k = 3;
  print(i,j,k);  /* 0 3 3 */
}
```

Pointer variables (cont.)

```
#include <stdio.h>

int main() {
  int i=1, j=2;
  int *maxvalue;
  if (i >= j)
    maxvalue = &i;
  else
    maxvalue = &j;
  printf("Max %d stored at %p.\n", *maxvalue, maxvalue);
}
```

Output:

```
Max 2 stored at 0xbffff458.
```

Since *maxvalue* points to the memory location occupied by *j*, changing *j* will also change $*maxvalue$ (but not *maxvalue*).
Similarly, changing $*maxvalue$ will change *j*.

Passing arguments by reference

Changes made to a parameter inside a function does not affect the value of the corresponding argument.
A function accepting a pointer as parameter can however change the value of the referenced object using the dereference operator.

Example: $R^2_{polar} \rightarrow R^2_{Cartesian}$

```
int polar2cart(double r, double phi, double *x, double *y) {
  if (r < 0)
    return 0;

  *x = r*cos(phi);
  *y = r*sin(phi);
  return 1;
}
```

Function call:

```
if (polar2cart(R, PHI, &X, &Y) != 1)
  fprintf(stderr, "Error in conversion\n");
```

Scope (are the printouts as expected?)

```
# include <stdio.h>
int e = 5;

void func(int a, int *b, int c){
  int d = 140;
  a = a + 100;
  *b = *b + 200;
  c = c + 300;
  d = d + 400;
  e = e + 500;
  printf("Inside func \n %d %d %d %d %d\n\n", a, *b, c, d, e);
}
main(){
  int a = 10, *b, c = 30, d = 40;
  b = &a;
  printf("1st printout: \n%d %d %d %d %d\n\n", a, *b, c, d, e);
  func(e, &a, *b);
  printf("Last printout: \n%d %d %d %d %d\n\n", a, *b, c, d, e);
}
```

Summary (pointers)

- Pointers are intended for pointing at locations in memory holding type-specific data.
- & - get address of variable ($maxvalue = \&i;$)
- $*$ - value at address ($*b = *b + 200;$)

One-Dimensional Arrays

Some concepts in mathematics cannot be represented in a natural way using the types we've seen so far. One such example is vectors. All basic types can be extended to be vector-valued (in the terminology of computer science aggregates of basic types or arrays).

A one-dimensional array consisting of n elements of the same type can be declared by $\boxed{\text{type name}[n];}$ and each element in the array accessed by $name[i]$, where i goes from 0 to $n-1$.
Note that the numbering of the components is different from the one normally used in mathematics (starting from 0 instead of 1).

NB! If you make a mistake when indexing arrays ($i < 0 || i >= n$) the compiler will not give a warning, but the program will compute the wrong result or crash (segmentation fault).

Example: $[1.0\,,\,0.5\,,\,0.1]^T \in R^3$ is defined by

```
double v[3];
v[0] = 1.0;   /* x-component */
v[1] = 0.5;   /* y-component */
v[2] = 0.1;   /* z-component */
```

Example:
Computing $\|x\|_2$ where $x \in R^{100}$ and $x_i = i$ (numbered from zero):

```
double x[100];
double l2norm = 0;
int    i;

for (i=0 ; i < 100 ; i++)
  x[i] = i;

for (i=0 ; i < 100 ; i++)
  l2norm += x[i] * x[i];

l2norm = sqrt(l2norm);
```

Multi-dimensional arrays

Arrays can be extended to 2 (matrices), 3 or even more dimensions, $\boxed{\text{type name}[n_1][n_2]\ldots[n_r]}$ and the elements accessed by

$name[i_1][i_2]\ldots[i_r]$ where $0 \leq i_j < n_j$.
Multi-dimensional arrays can be thought of as arrays of arrays.

Example:

The array $double\ x[10][3][5]$; can be thought of as belonging to $R^{10\times3\times5}$, and its elements accessed by $x[i_1][i_2][i_3]$, where

$$0 \leq i_1 < 10$$
$$0 \leq i_2 < 3$$
$$0 \leq i_3 < 5$$

Multi-d arrays & efficiency

It is convenient to store matrices as two-dimensional arrays, but not advisable* for computations due to inefficiency. A matrix

$$X \equiv \begin{bmatrix} x_{0,0} & \cdots & x_{0,n-1} \\ \vdots & \ddots & \vdots \\ x_{m-1,0} & \cdots & x_{m-1,n-1} \end{bmatrix} \in R^{m\times n}$$

is normally stored in a one-dimensional array $double\ x[lda * n]$, with the mapping from matrix to array defined by $x_{i,j} \rightarrow x[i + j * lda]$.

lda is the leading dimension of the matrix, satisfying $lda \geq m$ (remnant from Fortran). The leading dimension is the distance between the first element in column j and the first element in column $j+1$ in the array.

N.B. This format is used in almost every numerical library working with dense matrices.

* "Introduction to High Performance Computing".

Arrays as function arguments

Arrays can be used as function arguments but a function cannot return an array.

Arrays are always passed by reference, that is, all changes to the parameter will also affect the argument.

We can/must also supply the function information on the length of the array. For one dimensional arrays, a function declaration may look like

return-type function-name(int length, type parameter[]);

or equivalently

return-type function-name(int length, type *parameter);

(Other parameters declared as before.)

Example: Computing $\|\mathbf{x}\|_\infty$

```
#include <stdio.h>

double maxnorm(int n, double *x) {
  /* Compute max | x[i] |, 0 <= i < n */
  int    i;
  double nrm = -1;

  for (i=0 ; i<n ; i++)
    if (fabs(x[i]) > nrm)
      nrm = fabs(x[i]);

  return nrm;
}

main()
{
  double lista[7] = {7.23, -2, 13, -4.23, -23.42, 18.2, 1};
  printf("the maxnorm is: %f\n", maxnorm(7, lista));
}
```

Pointers as return values

A function can return a pointer variable.

Example: Determining $\max x_i$, $0 \leq i < n$

```
double* max(double *a, int n) {
  int i;
  double *p; // a local variable pointing to a non-local object.
  p = &a[0];
  for (i=1 ; i<n ; i++)
    if (a[i] > *p)  p = &a[i];
  return p;
}
main() {
  double a[4] = {2.3, -3.12, 32423.3, 3},
         *b;
  b = max(a, 4);
  printf("Maximum is: %lf \n", *b);
}
```

The pointer should never point to a local object since memory is freed when the variable exits scope (unpredictable contents).

Structure definitions

Structures are collections of values (members), possibly of different types, used for storing related data.

```
struct {
          type_1 member_1;
          ...
          type_n member_n
} identifier;
```

defines a structure variable named identifier with n members.

The value of a member is accessed through identifier.member.

Structure definitions (cont.)

Example, point in R^3

```
struct {double x, y, z;} point;
point.x = 1.0;
point.y = 0.3;
point.z = -0.5;
```

A structure can be initialized in the same way as an array,

```
struct {
        char name[128];
        double x, y, z;
} point = {"origin", 0, 0, 0};
```

Structure definitions (cont.)

A structure can be associated with a tag,

struct tag { ...};

Using the tag, a structure variable can be declared as:
`struct tag name;`

E.g.

```
struct fruit_tag {elements ...} plum, apple, pears;
struct fruit_tag    IS A SHORTHAND OF   struct {elements ...}

struct fruit_tag orange, grape;
```

Compare this with:

```
struct fruit_tag {int weight, price};
struct fruit_tag plum, apple, pears; (orange, grape if you like)
```

Structure definitions (cont.)

It is also convenient to use type definitions, e.g.

```
typedef struct {
  double re;
  double im;
} Complex;

main() {
  Complex x, y;
}
```

(A typedef can be used to simplify the declaration for a struct or pointer type, and to eliminate the need for the struct key word.)

```
typedef struct my_tag {int i; ...} my_type;
```

E.g.

```
typedef struct veg_tag {int weight, price} Veg;
```

Pointers to structures

It is sometimes necessary to have pointers to structures.
The members of a structure can be accessed by either of

```
(*pointer_to_struct).member
pointer_to_struct->member
```

Example:

Computing complex conjugate:

```
void conj(Complex *p){
  p->im = -p->im;
}
```

or

```
void conj(Complex *p){
  (*p).im = -(*p).im;
}
```

Working with structures

Structures can be used as arguments to and return values from functions.

Structures can also be used in other structure definitions (nested structures), e.g.

```
#define MAXDEGREE 10

typedef struct {
  int     degree;
  Complex coeff[MAXDEGREE+1];
} polynomial;
```

The assignment operator works for structures.
Arithmetic operators are not defined.

Dynamic memory allocation

Memory can be allocated during program execution using the functions `malloc` and `calloc` (stdlib.h)

pointer variable = malloc(size_t size); pointer variable = calloc(size_t nmemb, size_t size);

`calloc` initializes the block by setting all bits to 0.

Both these functions allocate memory and return a pointer to the memory block, or `NULL` if not enough memory is available.

`size_t` is defined in stdlib.h, and is equivalent to an unsigned int.

`nmemb` represents the number of elements, and `size` the size of each element.

Deallocating storage

When a memory block is no longer needed, it should be deallocated so that it can be reused for other purposes.

void free(void *p);

The memory is deallocated, but p still points to the same memory address. Modifying the memory at p is an error since that memory is no longer in our control. (p is a dangling pointer.)

`free` cannot be used to free memory from any other pointer than one returned by some alloc-routine.

Example:

```
void Heap() {
  int* intPtr;

//Allocates local pointer local variable (but not its pointee)

  intPtr = malloc(sizeof(int));
  *intPtr = 42;

//Allocates heap block and stores its pointer in local variable.
//Dereferences the pointer to set the pointee to 42.

  free(intPtr);

//Deallocates heap block making the pointer bad.
//The programmer must remember not to use the pointer
//after the pointee has been deallocated.
}
```

Self-referential structures

A structure with a pointer member that points to the structure itself is called a self-referential structure.

Example:

```
struct listNode
{
  char data;
  struct listNode *nextPtr;
};

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;
```

With structures like these, one can create dynamic data types like linked list, stack and queue.

Linked list

```
main()
{
  ListNodePtr startPtr = NULL;
  char item;
  int noOfNodes = 0;

  printf("Write data: ");
  scanf("\n%c",&item);
  while (item != 'q')
    {
      insert(&startPtr, item);
      printList(startPtr);
      printf("Write data: ");
      scanf("\n%c",&item);
      noOfNodes ++;
    }
  printf("%d\n", noOfNodes);
  printList(startPtr);
}
```

Linked list (cont.)

Insert a value of character type first in a list.

```
void insert(ListNodePtr *sPtr, char value)
{
  ListNodePtr newPtr;
  newPtr = (ListNode *) malloc(sizeof(ListNode));

  if (newPtr != NULL){
    newPtr->data = value;
    newPtr->nextPtr = *sPtr;
    *sPtr = newPtr;
  }
  else
    printf("Out of memory!! \n\n");
}
```

Linked list (cont.)

Write all elements in the linked list on screen,

```
void printList (ListNodePtr currPtr)
{
  if (currPtr == NULL)
    printf("The list is empty! \n");
  else
    {
    printf("The elements in the list: ");
    while (currPtr != NULL){
      printf("%c  --  ", currPtr->data);
      currPtr = currPtr->nextPtr;
      }
    }
  printf("\n\n");
}
```