Lecture 5
C Programming Language

Summary of Lecture 5

- Pointers
- Pointers and Arrays
- Function arguments
- Dynamic memory allocation
- Pointers to functions
- 2D arrays

Addresses and Pointers

- Every object in the computer has an **address**
- **Some** of the objects in a C program may be referenced through the address of their location in memory
  - Expressions like **&var**, are evaluated to the **address** of **var**.
- The **address operator &** cannot be applied to objects that have a temporary location in the memory (explicit constants, compound expressions)
- Addresses can be stored in variables of type **pointer to...**

Addresses and Pointers

- When pvar is a pointer variable carrying an address, the **dereferencing** (or **indirection**) operator * is used to extract the value stored in that address (via the expression *pvar)
- The dereferencing operator * is also used for the **declaration** of pointer type variables.
- Example:

```
int i, *pi;          /* pi - a pointer to integer  */
                     /* in other words, *pi is int */
i = 3; pi = &i;      /* now (*pi == 3)             */
*pi = 2;             /* now (i == 2)               */
```

<u>Memory Image:</u>

- After line 2, above:

  Address 0x6414 Address 0x6480

  i [3] [0x6414] pi

- After line 3, above:

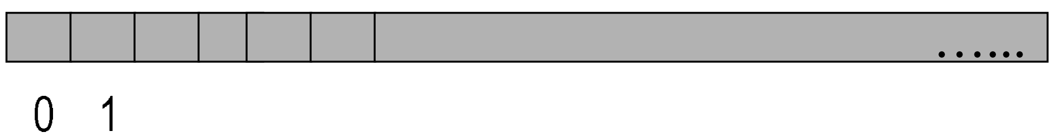
  Address 0x6414 Address 0x6480

  i [2] [0x6414] pi

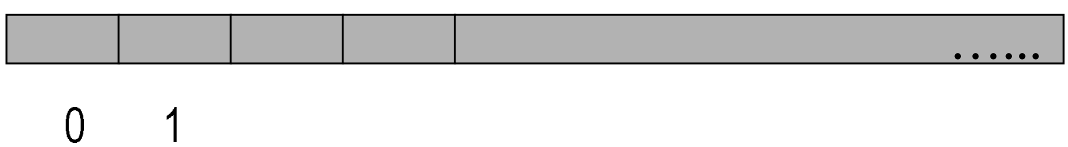
Addresses and Pointers

- In order to derefernce a pointer, it **must be known** to which **type** it refers
- Objects of different types may occupy spaces of different size, e.g. char, int, float, double. Example:

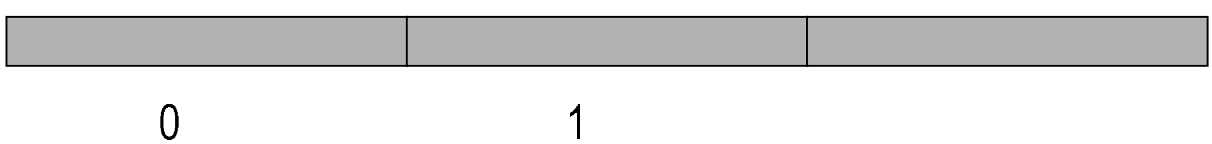
char c[N];   char *pc = &c[0] ; (*(pc+1)==c[1]);



int i[N];   int *pi = &i[0]; (*(pi+1)==i[1]);



double d[N]; double *pd = &d[0];(*(pd+1)==d[1]);



- It is illegal to compare two pointers, unless they are known to point to a single object (e.g. an array), or NULL. Illegal comparisons are sometimes possible, but the results may be surprising.

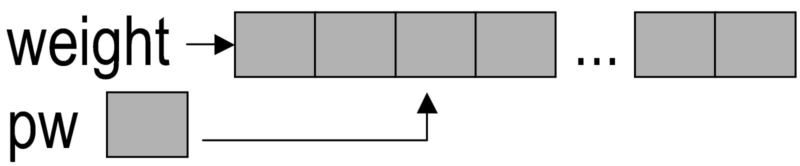
Pointers and Arrays

- For the declaration: double weight[LEN], *pw; the following holds:
- weight[i] is an expression of type **double** that refers to the value stored in the ith entry of the array
- weight is an expression of type **pointer to double** that refers to the address of the first element of the array.
  - This means that weight==&(weight[0]) is always TRUE.
- Fact: The C compiler always translates an "array expression" like weight[i] into the equivalent "pointer expression" *(weight+i)
- After assigning pw = weight, the expression pw[2] has the value weight[2]
- The main difference between pw and weight: weight is constant (cannot be assigned to) and pw is a variable !

Pointers and Arrays cont.

- Example:

pw = weight + 2;



pw++;   (now pw[0] is weight[3])

BUT - weight++ is illegal !!!

Pointers and Function-Arguments

- In C, function arguments are passed only by value
- This means that a variable that is used as an argument at a function call will always retain its value when the function returns to the caller
- By using pointers as function arguments this restriction is overcome (Re. Swap function)
- Example:

```
void Swap(int *a,int *b)
{
        int t = *a; *a = *b; *b = t;
}
int i = 3, j = 4;
Swap(&i,&j);
```

- The call to Swap() is a **call by reference**
- In both cases (Swap with/without pointers) **a** and **b** are local variables, and are initialized to the values of the function's arguments.

Arrays as Function arguments

- **Array parameters are an exception to the "call by value" rule in C**
- When an array is used as an argument at a function call, the entire array is not copied, but its address only is passed

  Examples:

  - "int vec[SIZE];": Function calls func(vec) and func(&vec[0]) are synonymous.
  - In "func" declaration, func(int *arr) and func(int arr[]) are synonymous too.
  - If only part of the array is transferred at function call: func(vec+2) and func(&vec[2]) are synonymous.
  - For multidimensional arrays, the above is true for the first (leftmost) index only.

Find the Error

- ```
  int *pa;
  ...
  *pa = 1;
  int *pb = pa
  ```

- consider:
  ```
  int a;
  int *pa , *pb;
  pa = &a;
  *pa = 1;
  ```
- consider:
  ```
  int *pa;
  int a = 3;
  pa = &a;
  printf(“%d”,*pa);
  pa = 1;
  printf(“%d”,*pa);
  ```
  ```

Double Indirection Review

- Consider:
  int a;
  int * pa;
  int **ppa;
- what type is &a ?
- int * pa
- What type is &pa ?
- Int **ppa

- After pa = &a, Are these statements correct:

  *ppa = pa;
  *ppa = &a;
  int ** ppa = &&a;

Dynamic Memory Allocation

- C allows general purpose dynamic memory allocation on the heap, restricted only by the amount of memory available at run time
- There are three predefined functions for this: (in /usr/include/stdlib.h)

```
void * malloc(num_bytes_to_allocate);
void * calloc(num_of_obj, size_of_obj);
void * realloc(old_ptr, new_size_in_bytes);
```

- If memory allocation fails, these functions return a NULL pointer.
- Since these functions return a pointer to void, when allocating memory use conversion:

```
int *pi = (int *) malloc(5*sizeof(int));  /* or: */
int *pi = (int *) calloc(5,sizeof(int));
pi = (int *) realloc(pi, 10*sizeof(int));
```

Dynamic Memory Allocation

- Why do we need dynamic memory allocation?
  - When the size of the array is passed as argument to the program
- IMPORTANT !!!
  After you finished using the variable which you dynamically allocated, FREE the memory:
  ```
  void free(void *);
  ```
  Usage:
  ```
  free(vec);
  ```
  If you don’t - you will experience **Memory Leak** - no free memory !
- **Dangling Pointer** - a pointer that points to a memory that is unreserved.
  (Example: allocate pointer and then reference to another location).

Dynamic Memory Allocation (cont)

- Example:

```
int * vec;
if ((vec=(int*) malloc(ARR_LNG*sizeof(int)))==NULL)
{
        fprintf(stderr,”cannot allocate\n”);
        exit(1);
}
if ((vec = (int*)realloc(vec,
NEW_ARR_LNG*sizeof(int)))
        ==NULL)
 {
        fprintf(stderr,”cannot allocate\n”);
        exit(1);
}
```

- Dynamically allocated memory (only) can be returned to the system using the function “void *free(old_ptr)”

Dynamic Memory Allocation (cont)

- Bad Example:

```
int * vec, *new_vec;
if ((vec=(int*) malloc(ARR_LNG*sizeof(int)))==NULL)
{
        fprintf(stderr,”cannot allocate\n”);
        exit(1);
}
if ((new_vec=(int*)realloc(vec,NEW_ARR_LNG*sizeof(int))
        ==NULL)
 {
        fprintf(stderr,”cannot allocate\n”);
        exit(1);
}
/* now, vec points to nowhere */
```

Pointers to Functions

- There are cases where there is a **function call** in a command but there is no prior knowledge **which function** is to be called
- Example:
```
void *v1, *v2;
if (compare(v1,v2) == 0) { …
```
v1, v2 may point to integers or strings or other types. An appropriate compare function should be called, according to the type of the objects pointed to by v1, v2.
- Solution:
```
enum {INT, STR}
int (*compare)(void*, void*); /*pointer to function*/
...
switch(type){
   case INT: compare = &num_compare; break;
   case STR: compare = &strcmp; break;
}
if ((*compare)(v1,v2) == 0) { /*or “compare(v1,v2)” */
```

Pointers to Functions

- Another case of using a pointer to function is when a function is used as an argument, passed to another function (a sub-case of the last case)
- Example:
  A definition of a function which is used as an argument:

```
void  string_manipulation(char s[], char (*chr_mnp)(char))
{
        int i;
        for (i=0; i<strlen(s);i++)
              s[i] = chr_mnp(s[i]);
             /* here “chr_mnp” is a given op. on a char*/
}

 /* Use of that function: */
char str[10] = “aBcD”;
...
string_manipulation(str,tolower);
```

Dynamic Memory Allocation - Example

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define LNG1 (6)
#define LNG2 (7)

void string_manipulation(char *s, char (*chr_mnp)(char))
{
        while (*s != '\0')  {
          *s  = chr_mnp(*s);
          s++;
        }
        return;
}
```

Dynamic Memory Allocation - Example

```
void main()
{
  char *str, *new_str, *str1;
  str = (char*) malloc(LNG2);
  while (printf("Enter 6-char string\n"), gets(str)) {
        printf("  %s ==>> ",str);
        string_manipulation(str,tolower);
        printf("%s\n",str);
  }
  new_str = (char *)realloc(str,LNG2+LNG1);
  printf("new_str=%s\n",new_str);
  printf("        str=%s\n", str);
  string_manipulation(str,toupper);
  printf("new_str=%s\n",new_str);
  printf("        str=%s\n", str);
  str1 = (char *) malloc(LNG2);
  printf("        str1=%s\n", str1);
  string_manipulation(str1,tolower);
  printf("        str1=%s\n", str1);
  free(str1); free(new_str);
}
```

Dynamic Memory Allocation - Example

A Sample Output

```
Enter 6-char string
 CDfsER ==>> cdfser
Enter 6-char string
 ZXYABC ==>> zxyabc
Enter 6-char string
new_str = zyxabc
        str = zyxabc   This is a FREE memory area
new_str = zyxabc
        str = ZYXABC A FREE memory is manipulated
       str1 = ZYXABC This area is ALLOCATED again
       str1 = zyxabc  and manipulated by an old pointer
```

1D Arrays

- Fixed (on stack) and Dynamic (on Heap) array are treated exactly the same, accept declaration and allocation:
- <u>Allocation:</u>
  <u>For dynamic:</u> <u>For fixed:</u>
  ```
  int * vec;                    int vec[100]; /* that's it*/
  vec = (int*)malloc(sizeof(int)*100);
  ```
- <u>Access:</u>
  `vec[70] = 1;` or:
  `*(vec+70) = 1;`
- <u>Initialization example:</u>
  Inefficient:
  ```
  for (i=0;i<100;i++) vec[i] = 0;
  int *ptr = vec;
  int *end = vec + 99; *end = 0; /* or end = vec+100 */
  while (ptr != end)   *ptr++ = 0;
  ```
- <u>Passing to a function</u>
  Function prototype:
  `void foo(int *ptr);` **or** `void foo(int ptr[]);`
  Function call: `foo(vec);`

Fixed 2D Arrays

- Arrays allocated on the stack
- Allocation:
  int fixed[50][100];
- Access:
  fixed[5][10] = 1; or:
  fixed[0][5*100+9] = 1; or:
  fixed[1][4*100+9] = 1; etc..
- Initialization example:
  Inefficient :

```
for (i=0;i<50;i++) for (j=0;j<100;j++)
        fixed[i][j] = 0;
int *ptr = fixed[0];
int *end = fixed[49]+99; *end = 0; /* or end = fixed+100 */
while (ptr != end)   *ptr++ = 0;
```

- Passing to a function
  Function prototype:
  void foo(int fixed[50][100]);
  Function call:
  foo(vec);

Dynamic 2D Arrays

- Allocated on the stack - more efficient, flexible
- Allocation:
```
int  **dynamic;
dynamic  =(int**)malloc(sizeof(int *)*50);
dynamic[0] = (int*)malloc(sizeof(int)*50*100);
for (i=1;i<50;i++) dynamic[i] = dynamic[i-1]+100;
```
  Access:
```
dynamic[5][10] = 1; or:
 dynamic[0][5*100+9] = 1;   or:
 dynamic[1][4*100+9] = 1;   etc..
```
- Initialization example:
```
int *ptr = dynamic[0];
int *end = dynamic[49]+99; /* or end = dynamic + 100 */
*end = 0;
while (ptr != end)   *ptr++ = 0;
```
- Passing to a function

  Function prototype:
```
void foo(int ** ptr);
```
  Function call:
```
foo(dynamic);
```

2D Arrays

- Here's an array with 10x20 elements:
  int arr[10][20];
- arr is now the same as &(arr[0][0])
- There are 10 rows and 20 columns
- the data is stored in row sequential format,
  so arr[2][5] is the same as arr[0][2*20+5]
- Name   Type   Same as
  arr
  arr[0]
  arr[2]
  arr[2][5]
- Example:

```
int **ppa;
ppa = arr;      /* points to arr[0] */
ppa = arr[2]   /* points to arr[2] */
*((*ppa)+3) = 7;
**(ppa+3) = 7;
```

More on Arrays

- Arrays must be explicitly initialized , they are not automatically initialized to 0 upon allocation
- Don’t forget to free the allocated memory to the array. Free is done in the reverse order of allocation:
  free(dynamic[0]);
  free(dynamic);
- For dynamically allocated array “dynamic”, indexing dynamic[j][k] requires no multiplication.
- Dynamic 3D arrays are defined as an array of pointers which point to dynamic 2D arrays.