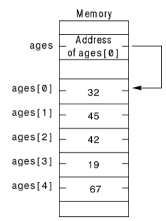

C-Programming, continued

Functions
Arrays

The if statement (and comparisons)

- The following variety might be used
  - `if (i < 2)`
  - `if (i <= 2)`
  - `if (i >= -1)`
  - `if (i == 4)      // note difference between == and =`
  - `if (x == 1.0)`
  - `if (fabs(x) < 10.0)`
  - `if (i < 8 && i > -5)           // && = and`
  - `if (x > 10.0 || x < -10.0)     // || = or`
- Remember to double up ==, &&, ||

Functions do stuff

```
#include <stdio.h>
#include <math.h>

double gauss(double x, double amplitude, double center, double sigma);

int main()
{
  double gaussval1,gaussval2;
  double xval=-1.4,ampl=100.0,ctr=0.1,sig=2.0;

  gaussval1 = gauss(1.5,10.0,0.0,2.0);
  gaussval2 = gauss(xval,ampl,ctr,sig);

  printf("Gaussval1 = %f; Gaussval2 = %f\n",gaussval1,gaussval2);

  return 0;
}

double gauss(double x_val, double amplitude, double center, double sigma)
{
  return amplitude*exp(-0.5*pow((x_val-center)/sigma,2));
}
```


Functional Notes

- In the previous program:
  - we have a function declaration before `main()`, specifying the argument types, with temporary (descriptive is good) names
    - not strictly necessary, but aids in checking errors during compilation
  - we can pass either numerical arguments or variables (or a mix)
  - names don't have to match from declaration to use in `main()` to names in function()
    - but have to match within function (note `x` vs. `xval` vs. `x_val`)
  - can pass any number of arguments of any type into function
  - limited to a single value out

UCSD: Physics 121; 2012

Arrays

- We can hold more than just one value in a variable
  - but the program needs to know how many places to save in memory
- Examples:

```
int i[8], j[8]={0}, k[]={9,8,6,5,4,3,2,1,0};
double x[10], y[10000]={0.0}, z[2]={1.0,3.0};
char name[20], state[]="California";
```

  - we can either say how many elements to allow and leave them unset; say how many elements and initialize all elements to zero; leave out the number of elements and specify explicitly; specify number of elements and contents
  - character arrays are strings
  - strings must end in `'\0'` to signal the end
  - must allow room: `char name[4]="Bob"`
    - fourth element is `'\0'` by default

Winter 2012 5

UCSD: Physics 121; 2012

Indexing Arrays

```
int i,j[8]={0},k[]={2,4,6,8,1,3,5,7};
double x[8]={0.0},y[2]={1.0,3.0},z[8];
char name[20],state[]="California";

for (i=0; i<8; i++)
{
  z[i] = 0.0;
  printf("j[%d] = %d, k[%d] = %d\n",i,j[i],i,k[i]);
}
name[0]='T';
name[1]='o';
name[2]='m';
name[3] = '\0';
printf("%s starts with %c and lives in %s\n",name,name[0],state);
```

- Index array integers, starting with zero
- Sometimes initialize in loop (`z[]` above)
- String assignment awkward outside of declaration line
  - `#include <string.h>` provides "useful" string routines

Winter 2012 6

UCSD: Physics 121; 2012

Memory Allocation in Arrays

- `state[]="California";` →

C	a	l	i	f	o	r	n	i	a	\0

- `name[11]="Bob";` →

B	o	b	\0							

  - empty spaces at the end could contain any random garbage
- `int i[] = {9,8,7,6,5,4,3,2};` →

9	8	7	6	5	4	3	2

  - indexing `int[8]` is out of bounds, and will either cause a segmentation fault (if writing), or return garbage (if reading)

Winter 2012 7

UCSD: Physics 121; 2012

#define to ease the coding

```
#define NPOINTS 10
#define NDIMS    3

int main()
{
  int shots[NPOINTS],hits[NPOINTS],flag[NDIMS];
  double coords[NDIMS][NPOINTS],time_hit=[NPOINTS];
…
}
```

- `#define` comes before the function definitions, up with the `#include` statements
  - note no semi-colons
  - just a text replacement process: any appearance of NPOINTS in the source code is replaced by 10
  - Convention to use all CAPs to differentiate from normal variables or commands
  - Now to change the number of points processed by that program, only have to modify one line

Winter 2012 8

Multi-Dimensional Arrays

```
int i,j,arr[2][4];

for (i=0; i<2; i++){
  for (j=0; j<4; j++){
    arr[i][j] = 4+j-2*i;
  }
}
```

i \ j	0	1	2	3
0	4	5	6	7
1	2	3	4	5

in memory space:

4	5	6	7	2	3	4	5

- C is a row-major language: the first index describes which row (not column), and arranged in memory row-by-row
  - memory is, after all, strictly one-dimensional
- Have the option of treating a 2-D array as 1-D
  - `arr[5] = arr[1][1] = 3`
- Can have arrays of 2, 3, 4, … dimensions

Arrays and functions

- How to pass arrays into and out of functions?
- An array in C is actually handled as a “pointer”
  - a pointer is a direction to a place in memory
- A pointer to a double variable’s address is given by the & symbol
  - remember this from `scanf` functions
- For an array, the name is already an address
  - because it’s a block of memory, the name by itself doesn’t contain a unique value
  - instead, the name returns the address of the first element
  - if we have `int arr[i][j];` `arr` and `&arr[0]` and `&arr[0][0]` mean the same thing: the address of the first element
- By passing an address to a function, it can manipulate the contents of memory directly, without having to pass bulky objects back and forth explicitly

Example: 3x3 matrix multiplication

```
void mm3x3(double a[], double b[], double c[])

// Takes two 3x3 matrix pointers, a, b, stored in 1-d arrays nine
// elements long (row major, such that elements 0,1,2 go across a
// row, and 0,3,6 go down a column), and multiplies a*b = c.
{

  double *cptr;
  int i,j;

  cptr = c;

  for (i=0; i<3; i++){
    for (j=0; j<3; j++){
      *cptr++ = a[3*i]*b[j] + a[3*i+1]*b[j+3] + a[3*i+2]*b[j+6];
    }
  }
}
```


mm3x3, expanded

- The function is basically doing the following:

```
*cptr++ = a[0]*b[0] + a[1]*b[3] + a[2]*b[6];
*cptr++ = a[0]*b[1] + a[1]*b[4] + a[2]*b[7];
*cptr++ = a[0]*b[2] + a[1]*b[5] + a[2]*b[8];

*cptr++ = a[3]*b[0] + a[4]*b[3] + a[5]*b[6];
*cptr++ = a[3]*b[1] + a[4]*b[4] + a[5]*b[7];
*cptr++ = a[3]*b[2] + a[4]*b[5] + a[5]*b[8];

*cptr++ = a[6]*b[0] + a[7]*b[3] + a[8]*b[6];
*cptr++ = a[6]*b[1] + a[7]*b[4] + a[8]*b[7];
*cptr++ = a[6]*b[2] + a[7]*b[5] + a[8]*b[8];
```

Notes on mm3x3

- The function is made to deal with 1-d instead of 2-d arrays
  - 9 elements instead of 3x3
  - it could have been done either way
- There is a pointer, `*cptr` being used
  - by specifying `cptr` as a double pointer, and assigning its address (just `cptr`) to `c`, we can stock the memory by using "pointer math"
  - `cptr` is the address; `*cptr` is the value at that address
  - just like `&x_val` is an address, while `x_val` contains the value
  - `cptr++` bumps the address by the amount appropriate to that particular data type
  - `*cptr++ = value;` assigns value to `*cptr`, then advances the `cptr` count

Using mm3x3

```
#include <stdio.h>

void mm3x3(double a[], double b[], double c[]);

int main()
{
  double a[]={1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
  double b[]={1.0, 2.0, 3.0, 4.0, 5.0, 4.0, 3.0, 2.0, 1.0};
  double c[9];

  mm3x3(a,b,c);

  printf("c = %f  %f  %f\n",c[0],c[1],c[2]);
  printf("    %f  %f  %f\n",c[3],c[4],c[5]);
  printf("    %f  %f  %f\n",c[6],c[7],c[8]);

  return 0;
}
```

- passing just the names (addresses) of the arrays
  - defining a and b, just making space for c
  - note function declaration before main

Another way to skin the cat

```
double a[3][3]={{1.0, 2.0, 3.0},
                {4.0, 5.0, 6.0},
                {7.0, 8.0, 9.0}};
double b[3][3]={{1.0, 2.0, 3.0},
                {4.0, 5.0, 4.0},
                {3.0, 2.0, 1.0}};
double c[3][3];

mm3x3(a,b,c);
```

- Here, we define the arrays as 2-d, knowing that in memory they will still be 1-d
  - we will get compiler warnings, but the thing will still *work*
  - not a recommended approach, just presented here for educational purposes
  - Note that we could replace `a` with `&a[0][0]` in the function call, and the same for the others, and get no compiler errors