## today

mon 23 sep 2002

- homework #1 — due today
- homework #2 — out today
- quiz #1 — next class
  - 30-45 minutes long
  - one page of notes
  - topics: C
- advanced data types
- dynamic memory allocation
- structured data types (array, struct)



## advanced data types (1) — typedef.

- defining your own types using `typedef`

```
typedef short int smallNumber;
typedef unsigned char byte;
typedef char String[100];

smallNumber x;
byte b;
String name;
```



## advanced data types (2) — typedef.

- defining your own boolean:

```
typedef char boolean;
#define FALSE 0
#define TRUE 1
```

- generally works, but beware:

```
check = x > 0;
if ( check == TRUE ) {...}
```

- if `x` is positive, `check` will be non-zero, but may not be == 1



## advanced data types (3) — enum.

- define new integer-like types as *enumerated* types:

```
enum weather { rain, snow=2, sun=4 };

typedef enum {
  Red, Orange, Yellow, Green, Blue, Violet
} Color;
```

- look like C identifiers (names)
- are listed (enumerated) in definition
- treated like integers
  - start with 0 (unless you set value)
  - can add, subtract — e.g., `color + weather`
  - cannot print as symbol automatically (you have to write code to do the translation)

## advanced data types (4) — enum.

- just fancy syntax for an ordered collection of integer constants:

```
typedef enum {
   Red, Orange, Yellow
} Color;
```

is like

```
#define Red 0
#define Orange 1
#define Yellow 2
```

- here's another way to define your own boolean:

```
typedef enum {False, True} boolean;
```



## advanced data types (5) — data objects.

- C does not have Objects in the OOP sense (like Java and C++ do)
- but C has *data objects* — i.e., variables

```
short int x;
char ch;
float pi = 3.1415;
float f, g;
```

- scope
  - variables defined in `{ }` block are active only in block — e.g., *local*
  - variables defined outside a block are *global* (persist during program execution)
  - *static* variables may be declared outside a block, but are not globally visible



## advanced data types (6) — data objects.

- variables must be declared before they are used
- we have used variables within `main()` and within functions
- *global variables*
  - are declared *outside* main() and outside any function, usually at the top of the program file, after any #'s (preprocessor directives)
  - can be "seen" anywhere
- *local variables*
  - are declared within a program block or function
  - they can only be seen inside the block in which they are defined
  - function arguments are local to the function they are passed to



## advanced data types (7) — usage.

- a variable is conceptually a container that can hold a value
- default value is (mostly) undefined — you should treat it as a random number
- the compiler may warn you about uninitialized variables, but not as reliably as Java
- variables are always passed *by value*, but you can pass the *address* of a variable to a function:

```
scanf( "%d%f", &x, &f );
```

## advanced data types (8) — sizes.

- every data object in C has:
  - a name and data type (specified in definition)
  - an address (its relative location in memory)
  - a size (number of bytes of memory it occupies)
  - visibility (which parts of program can refer to it)
  - lifetime (period during which it exists)
- Unlike scripting languages and Java, all C data objects have a fixed size over their lifetime
  - *except dynamically created objects*
- size of object is determined when object is created:
  - global data objects at compile time (data)
  - local data objects at run-time (stack)
  - dynamic data objects by programmer (heap)



## dynamic memory allocation (1).

- *malloc()* allocates a block of memory:

```
void *malloc( size_t size );
```

- lifetime of the block is until memory is freed, with *free()*:

```
void free( void *ptr );
```

- example:

```
int *dynvec, num_elements;
printf( "how many elements do you want to enter? " );
scanf( "%d", &num_elements );
dynvec = (int *)malloc( sizeof(int) * num_elements );
```



## dynamic memory allocation (2).

- memory leaks — memory allocated that is never freed:

```
char *combine( char *s, char *t ) {
  u = (char *)malloc( strlen(s) + strlen(t) + 1 );
  if ( s != t ) {
    strcpy( u, s );
    strcat( u, t );
    return u;
  }
  else {
    return 0;
  }
} /* end of combine() */
```

- `u` should be freed if `return 0;` is executed
- but you don't need to free it if you are still using it!



## dynamic memory allocation (3).

- note: `malloc()` does not initialize data
- you can allocate and initialize with "calloc":

```
void *calloc( size_t nmemb, size_t size );
```

  - calloc allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero.
- you can also change size of allocated memory blocks with "realloc":

```
void *realloc( void *ptr, size_t size );
```

  - realloc changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized.
- these are all functions in `stdlib.h`
- for more information: `unix$ man malloc`

## structured data types (1).

- structured data types are available as:

| object | property |
|---|---|
| `array [ ]` | enumerated; indexed from 0 |
| `struct` | names and types of fields |
| `union` | made up of multiple elements, but only one exists at a time; each element could be a native data type, a pointer or a struct |



## structured data types (2) — arrays.

- "arrays" are defined by specifying an element type and number of elements
  - statically:

```
int vec[100];
char str[30];
float m[10][10];
```

  - dynamically:

```
int *dynvec, num_elements;
printf( "how many elements do you want to enter? " );
scanf( "%d", &num_elements );
dynvec = (int *)malloc( sizeof(int) * num_elements );
```

- for an array containing N elements, indeces are 0..N-1
- stored as a linear arrangement of elements
- often similar to pointers



## structured data types (3) — arrays.

- C does not remember how large arrays are (i.e., no length attribute, unlike Java)
- given:

```
int x[10];
x[10] = 5;  /* error! */
```

- ERROR! because you have only defined x[0]..x[9] and the memory location where x[10] is can become something else...
- `sizeof x` gives the number of bytes in the array
- `sizeof x[0]` gives the number of bytes in one array element
- thus you can compute the length of `x` via:

```
int length_x = sizeof x / sizeof x[0];
```



## structured data types (4) — arrays.

- when an array is passed as a parameter to a function:
  - the size information is not available inside the function
  - array size is typically passed as an additional parameter

```
printArray( x, length_x );
```

  - or as part of a struct (best practice; object-like)

```
typedef struct {
  int x[10];
  int length_x;
} Array;
Array ax;
ax.length_x = 10;
printArray( ax );
```

  - or globally

```
#define VECSIZE 10
int x[VECSIZE];
```

## structured data types (5) — arrays.

- array elements are accessed using the same syntax as in Java: `array[index]`
- C does not check whether array index values are sensible (i.e., no bounds checking)
- e.g., `x[-1]` or `vec[10000]` will not generate a compiler warning!
- if you're lucky, the program crashes with

```
Segmentation fault (core dumped)
```



## structured data types (6) — arrays.

- C references arrays by the address of their first element
- array is equivalent to `&array[0]`
- you can iterate through arrays using pointers as well as indexes:

```
int *v, *last;
int sum = 0;
last = &x[length_x-1];
for ( v = x; v <= last; v++ )
  sum += *v;
```



## structured data types (7) — arrays.

- example:

```
#include <stdio.h>
#define MAX 12
int main( void ) {
  int x[MAX]; /* declare 12-element array */
  int i, sum;
  for ( i=0; i<MAX; i++ ) { x[i] = i; }
  /* here, what is value of i? of x[i]? */
  sum = 0;
  for ( i=0; i<MAX; i++ ) { sum += x[i]; }
  printf( "sum = %d\n",sum );
} /* end of main() */
```



## structured data types (8) — arrays.

- another example:

```
#include <stdio.h>
#define MAX 10
int main( void ) {
  int x[MAX]; /* declare 10-element array */
  int i, sum, *p;
  p = &x[0];
  for ( i=0; i<MAX; i++ ) { *p = i + 1;  p++; }
  p = &x[0];
  sum = 0;
  for ( i=0; i<MAX; i++ ) {  sum += *p;  p++; }
  printf( "sum = %d\n",sum );
} /* end of main() */
```

## structured data types (9) — 2D arrays.

- 2-dimensional arrays

```
int weekends[52][2];
```

```
 [0][0]  [0][1] [1][0] [1][1] [2][0] [2][1] [3][0] ...
   ↑
weekends
```

- you can use indices or pointer math to locate elements in the array
  - `weekends[0][1]`
  - `weekends+1`
- `weekends[2][1]` is same as `*(weekends+2*2+1)`, but NOT the same as `*weekends+2*2+1` (which is an integer)!



## structured data types (10) — struct.

- `struct` is similar to a field in a Java object definition
- it's a way of grouping multiple data types together
- components can be any type (but not recursive)
- accessed using the same syntax struct.field

```
int main() {
  struct {
    int x;
    char y;
    float z;
  } rec;
  rec.x = 3;
  rec.y = 'a';
  rec.z = 3.1415;
  printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} /* end of main() */
```



## structured data types (11) — struct.

- variables of `struct` types can be declared in two ways:
  - using a tag associated with the `struct` definition
  - wrapping the `struct` definition inside a `typedef`
- example:

```
int main() {
  struct record {
    int x;
    char y;
    float z;
  };
  struct record rec;
  rec.x = 3;
  rec.y = 'a';
  rec.z = 3.1415;
  printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} /* end of main()
```



## structured data types (12) — struct.

- another example:

```
int main() {
  typedef struct {
    int x;
    char y;
    float z;
  } Record;
  Record rec;
  rec.x = 3;
  rec.y = 'a';
  rec.z = 3.1415;
  printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} /* end of main()
```

## structured data types (13) — struct.

- overall size of `struct` is the sum of the elements, plus padding for alignment
- given previous 3 examples:
  `sizeof( rec )` → 12
- but, it depends on the size and order of content (e.g., `int`s need to be aligned on word boundaries, since size of char is 1 and size of int is 4):

```
struct {
  char x;
  int y;
  char z;
} s1;
/*  x    y    z     */
/* |----|----|----| */
/* sizeof s1 -> 12  */
```

```
struct {
  char x, y;
  int z;
} s2;
/*  xy   z     */
/* |----|----| */
/* sizeof s2 -> 8  */
```



## structured data types (14) — struct.

- pointers to structs are common — especially useful with functions (as arguments to functions or as function type)
- two notations for accessing elements: `(*sp).field` or `sp->field`
  (note: *sp.field doesn't work)

```
struct xyz {
 int x, y, z;
};
struct xyz s;
struct xyz *sp;
...
s.x = 1;
s.y = 2;
s.z = 3;
sp = &s;
(*sp).z = sp->x + sp->y;
```



## structured data types (15) — extended example p1.

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

#define NAME_LEN 40

struct person {
  char  name[NAME_LEN+1];
  float height;
  struct { /* nested structure */
    int day;
    int month;
    int year;
  } birthday;
};

void printPerson( struct person * ); /* prototype */
```



## structured data types (16) — extended example p2.

```
int main( void ) {
  struct person suzanne;    /* declare one */
  struct person class[120]; /* declare an array */
  /* store info in one */
  strcpy( suzanne.name,"suzanne" );
  suzanne.height = 60;
  suzanne.birthday.day = 16;
  suzanne.birthday.month = 5;
  suzanne.birthday.year = 1988;
  /* store info in the array */
  strcpy( class[0].name,"alex" );
  class[0].height = 48;
  class[0].birthday.day = 9;
  class[0].birthday.month = 5;
  class[0].birthday.year = 1995;
  strcpy( class[1].name,"jen" );
  class[1].height = 55;
```

structured data types (17) — extended example p3.

```
  class[1].birthday.day = 14;
  class[1].birthday.month = 4;
  class[1].birthday.year = 1992;
  /* print them... */
  printPerson( &suzanne );
  printPerson( &class[0] );
  printPerson( &class[1] );
} /* end of main() */

void printPerson( struct person *p ) {
  printf( "name   = [%s]\n",p->name );
  printf( "height = %5.2f inches\n",p->height );
  printf( "birthday = %02d/%02d/%4d\n",p->birthday.day,
          p->birthday.month,p->birthday.year );
}
```