Tour of C

Data Types, Arrays, Strings and Pointers in C

Data Types

- char
- short
- int
- long
- float
- double
- long double
- signed/unsigned - char, short, int, long

Like Java, Like C

- Operators same as in Java
  - Arithmetic
    - int i = i+1; i++; i--; i*=2;
    - +, -, *, /, %
  - Relational and logical
    - <, >, <=, >=, ==, !=
    - &&, ||, &, |, !
- Syntax same as in Java
  - if() { } else { }
  - while() { }
  - do { } while ();
  - for(i = 1; i <= 100; i++) { }
  - switch() {case 1: …}
  - continue; break;

Arrays in C

Array Declaration

- The dimension must be a constant or a constant expression (needs to be determined at compile time)

```
define BUFFSIZE 256
define MAXSIZE 12
int main()
{
  int size = 6;
  char buffer[BUFFSIZE]; // elegant declaration
  char buffer[256]; // NOT an elegant declaration
  int dice[BUFFSIZE / 2]; // constant integer expression
  float rollValue[size]; // NOT valid - size is not a constant
  ...
```

Array Initialization

- Providing too many initial values causes a compile-time error:
  ```
  int count[MAX] = {5, 7, 2, 4, 8}; // ERROR!
  ```

- If the number of initial values listed is less than the capacity of the array, the remaining elements are automatically initialized to 0. Thus
  ```
  int count[5] = {2};
  ```
  is equivalent to
  ```
  int count[5] = {2,0,0,0,0};
  ```
Array Indices

Logically, valid indices for an array range from 0 to \( \text{MAX}-1 \), where \( \text{MAX} \) is the dimension of the array.

```
int A[6];
```

stands for


Logically, there is no \( A[6]! \)

Memory

```
```

Out-of-Bounds Array Indices (1)

A common logical error in processing arrays is exceeding the valid index range:

```
#define MAX 100
int someArray[MAX];
someArray[MAX] = 0; // will compile
// but run time ERROR!
```

What happens when a statement uses an array index that is out of bounds?

```
int i;
for(i = 0; i <= MAX; i++)
someArray[i] = 0;
```

No automatic checking of array indices at run time!

Out-of-Bounds Array Indices (2)

The memory location \( \text{someArray}[100] \) may:

1. Store a variable declared in your program - that variable will be altered. Since there is no statement that directly assigns a value to that variable, this effect seems very mysterious when debugging.

2. Not be allocated for the use of your program. The result depends on the operating system you are using
   - Some operating systems, such as Windows 95/98 do not carefully monitor memory accesses and your program may corrupt a value that actually belongs to another program
   - Other operating systems, such as Windows NT and Unix, will detect that a memory access violation has occurred and suspend or kill your program

Lack of Aggregate Array Operations

Aggregate operations refer to operations on an array as a whole, as opposed to operations on individual array elements.

```
#define MAX 100
int x[MAX];
int y[MAX];
```

```
// There are no aggregate operations on arrays:
Assignment  x = y;              Error!
Comparison  if (x == y) ...       Error!
I/O         printf("%d", x);       Error!
Arithmetic x = x + y;           Error!
```

String Declarations

Unlike Java, there is no String data type in C

Strings in C are simply arrays of characters terminated with 0 (character \'\0\')

```
char some[10]; // need to specify max size
```

// one way:
```
char mag[6] = {'W', 'e', 'y', ' ', '!', '\0'};
```

// another way (last 2 places unused):
```
char mag[8] = "Hey !"; // no \'\0' \\
```

// or
```
char mag[] = "Hey !";    // no \'\0'
```

--- memory for 6 characters (5 plus the null char \'\0\') automatically allocated
Memory Representation

- `char some[10];`
  
  - `char msg[6] = {'H','e','y',' ','!','\0'};`
  - `char msg[8] = "Hey !";`
  - `char msg[] = "Hey !";`

Reading Into a String (1)

- `#define MAX_BUFFER 20
  char buffer[MAX_BUFFER];
  scanf("%s", buffer); // or
  gets(buffer, MAX_BUFFER);`

  - What if the array is not large enough to hold the input?
    - Characters will be stored into memory locations past the end of the array
    - Will result in run-time memory access violation error!

Reading Into a String (2)

- Better:
  - `#define MAX_BUFFER 20
    char buffer[MAX_BUFFER];
    fgets(buffer, MAX_BUFFER, stdin);`

  - `fgets` is similar to `gets`, but:
    - It takes a third argument, in our case standard input
    - If stores into buffer no more than MAX_BUFFER chars (extra characters are ignored), so memory violation error won't occur

Functions for Manipulating Strings

- C provides a large number of functions for manipulating strings. Four important ones:
  - `strlen(s)`
    - Returns the length of s
  - `strcpy(toS, fromS)`
    - Copies fromS to toS (toS must be large enough)
  - `strcmp(s1, s2)`
    - Returns 0 if s1 == s2
    - Returns an integer < 0 if s1 < s2
    - Returns an integer > 0 if s1 > s2
  - `strtok` - Read the Sun manual pages to find out what this function does

What are Pointers?

- A pointer is a variable that holds the address of another variable (object)

  - Suppose that we have an integer variable
    - `int i;`
    - And wish to have a pointer point to this variable. Thus we need to know the memory address of i.

  - How do we know the address of i? ...

    - `&i` is the address of i. The operator `&` is called the ADDRESS-OF operator.

Pointers in C
We can declare that a pointer \texttt{iPtr} points to an int by saying
\begin{verbatim}
int * iPtr;
\end{verbatim}

Suppose that we have:
\begin{verbatim}
int i = 5;
int j = 7;
\end{verbatim}

We can make \texttt{iPtr} point to \texttt{i} by assigning to \texttt{iPtr} the memory location where \texttt{i} is stored. Thus
\begin{verbatim}
iPtr = &i;
\end{verbatim}
sets \texttt{iPtr} to point to \texttt{i}.

We can also initialize \texttt{iPtr} at the point of declaration:
\begin{verbatim}
int i;
int * iPtr = &i;
\end{verbatim}

Here is a common error:
\begin{verbatim}
int i;
int * iPtr = i;  // ERROR: i is not an address
\end{verbatim}

When declaring several pointer variables in one statement - the asterisk does not distribute throughout the statement:
\begin{verbatim}
int * p, q;
\end{verbatim}
equivalent to
\begin{verbatim}
int * p;
int * q;
\end{verbatim}

\begin{verbatim}
int * p, * q;
\end{verbatim}
equivalent to
\begin{verbatim}
int * p;
int * q;
\end{verbatim}

The value of the data being pointed at is obtained by using the operator \texttt{*}.

If \texttt{p} is a pointer value, then
\begin{verbatim}
*p
\end{verbatim}
refers to the variable pointed to by \texttt{p}. Since reference is another name for address, the operator \texttt{*} is called dereference operator.

A dereferenced pointer behaves exactly like the variable it points to.
Uninitialized Pointers

Suppose that we have the following declarations:

```c
int i;
int * iPtr;
*iPtr = 100;
```

What is the value of `iPtr`? **Undefined.** What could happen?
- `iPtr` could hold an address that does not make sense at all, causing your program to crash if dereferenced.
- `iPtr` could point to an address which is accessible. Then the assignment `*iPtr = 100;` would accidentally change some other data, which could result in a crash at a later point. This is a tough error to detect since the cause and symptom may be widely separated in time.

The null Pointer

The value of a pointer can be:
- some garbage (pointer unassigned)
- the address of some variable (e.g., `int * p = &i;`)
- the constant 0 (the null pointer, points to absolutely nothing)

```c
somePointer = 0;
```

This statement does not cause `somePointer` to point to memory location zero; it guarantees that `somePointer` does not point to anything.

The null pointer is a special pointer value that a program can test for:

```c
if (somePointer == 0) ...
```

Arrays and Pointers

An array name is basically a constant pointer

Consider the declaration:

```c
char a[3];
```

The compiler allocates three integers for the array object. These are referenced as `a[0]`, `a[1]`, `a[2]` and occupy a contiguous block of memory.

The value of `a` is exactly `&a[0]`, the address of the first integer in the array
Arrays are NOT Pointers

- Declaring an array sets aside space for its elements
  ```c
  char a[5];
  ```

- Declaring a pointer variable sets aside only space to hold the variable
  ```c
  char * p;
  ```

- You can change a pointer variable, but not the address of an array
  ```c
  char b[6];
  p = b;  // OK
  a = b;  // ERROR!
  ```

Other resources

- A very good tutorial on pointers and arrays in C:
  http://pw1.netcom.com/~tjensen/ptr/pointers.htm