C: Arrays, and strings

Department of Computer Science
College of Engineering
Boise State University

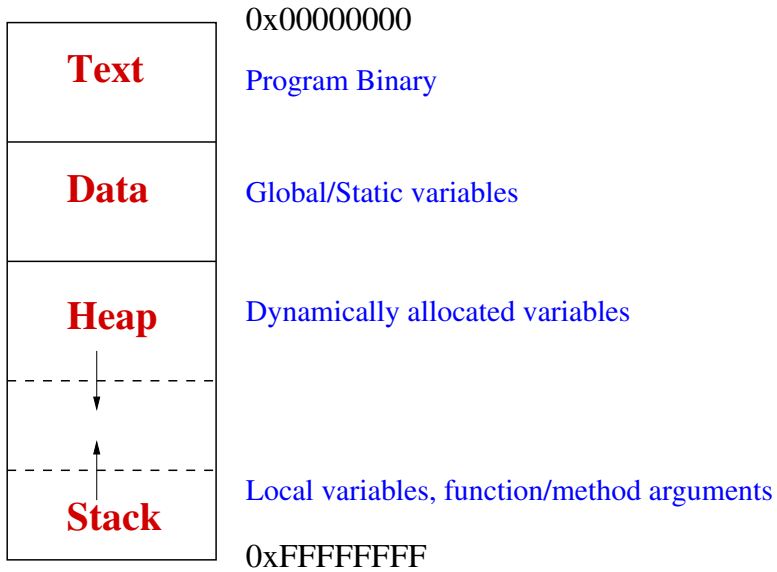
September 11, 2017

1-dimensional Arrays

- Arrays can be statically declared in C, such as:

```
int A[100];
```

The space for this array is declared on the stack segment of the memory for the program (if A is a local variable) or in the data segment if A is a global variable.



- Arrays in C do not contain a length property like in Java
- When arrays are passed as a function argument they decay to a pointer. Thus the code below **COULD** be incorrect if the array below is function parameter.

```
int numOfElefments = sizeof(array) / sizeof(array
[0]);
```

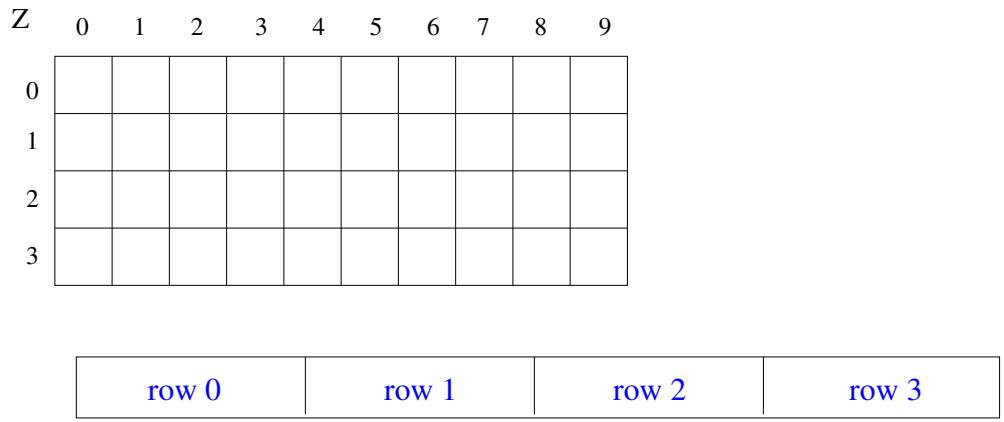
- Here is a good stackoverflow.com posting if you need more info https://stackoverflow.com/questions/33523585/how-do-sizeofarr-sizeofarr0-work

Two-dimensional Arrays

- A 2-dimensional array can be statically allocated in C as shown in the following example:

```
int Z[4][10];
```

- This array is laid out in memory in row major order, that is, it is laid out as a 1d array with row 0 first followed by row 1, row 2 and then row 3.



Row Major Layout

- Some languages use a column major layout for 2-d arrays, which is column 0, then column 1, . . . , and finally column 9 in the above example.

C-style Strings

A C-style string is an array of characters terminated by the null character - `'\0'` (ASCII code = 0).

- There are four ways to declare a string.
  - Modifiable, fixed size array. Compiler determines size based on string literal (e.g. magma takes 6 chars). Use this if you know the length or value of the string at compile-time.

```
char s0[] = "magma";
s0[0] = 'd'; /* legal */
s0 = "magma"; /* illegal */
```

  - Pointer to un-named, static, read-only array.

```
char *s1 = "volcano";
s1[0] = 'm'; /* illegal - read only*/
s1 = "lava"; /* legal */
```

  - Empty, fixed size array.

```
char s2[MAXLEN];
```

  - Uninitialized pointer. Use this if you don't know the length of the potential string until run-time.

```
char *s3;
/* Some later point in your program...*/
s3 = (char *) malloc(sizeof(char) * (strlen(s0) +
1));
```

C-style Strings

- C doesn't provide strings as a basic type so we use functions to do operations with strings...
- The header file is `string.h`
- See man page for `string` for a list of all C string functions.
- Common string manipulation functions: `strlen()`, `strcat()`, `strncat()`, `strcpy()`, `strncpy()`, `strcmp()`, `strncmp()`, `strtok()`, `strsep()`, `strfry()` etc.
- Copying strings. What is wrong with the following? Read man 3 strcpy.

```
strcpy(dest, src); /* not safe */
```

We can solve this using `strncpy()`

```
strncpy(dest, src, MAXLEN); /* safer, but read man page */
```

Better solution is to allocate correct size, then copy.

```
char *dest = (char *) malloc(sizeof(char)*(strlen(src)+1));
strcpy(dest, src);
```

String Copy Example


```c
/* strcpy; copy t to s, array version */
void strcpy(char *s, char *t) {
    int i=0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

```c
/* strcpy; copy t to s, pointer version 1 */
void strcpy(char *s, char *t) {
    int i=0;
    while ((*s = *t) != '\0'){
        s++;
        t++;
    }
}
```

String Copy Example (contd.)


```c
/* strcpy; copy t to s, pointer version 2 */
void strcpy(char *s, char *t) {
    while ((*s++ = *t++) != '\0')
        ;
}
```

```c
/* strcpy; copy t to s, pointer version 3 */
void strcpy(char *s, char *t) {
    while ((*s++ = *t++))
        ;
}
```

Recommend the use of parentheses around assignment used as a boolean

String Comparison

- `strcmp(s, t)` returns negative, zero or positive if `s` is lexicographically less, equal or greater than `t`. The value is obtained by subtracting the characters at the first position where `s` and `t` differ.

```
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) {
    int i;
    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

String Comparison (contd.)

The pointer version of `strcmp`:

```
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) {
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

String Examples

- A String Example: C-examples/strings/strings-ex1.c
- A String Tokenizing Example: C-examples/strings/strings-ex2.c
- A Better String Tokenizing Example: C-examples/strings/strings-ex3.c

Other String Tokenizing Functions

- Use `strsep()` in cases where there are empty fields between delimiters that `strtok()` cannot handle.

Command line arguments

- Recommended prototype for the main function:

```
int main(int argc, char *argv[])
```

or

```
int main(int argc, char **argv)
```

- `argc` C-style strings are being passed to the main function from `argv[0]` through `argv[argc-1]`. The name of the executable is `argv[0]`, the first command line argument is `argv[1]` and so on. Thus `argv` is an array of pointers to `char`.
- **In-class Exercise**. Draw the memory layout of the `argv` array.

Variable Argument Lists in C (1)

- C allows a function call to have a variable number of arguments with the variable argument list mechanism.
- Use *ellipsis* `...` to denote a variable number of arguments to the compiler. the ellipsis can only occur at the end of an argument list.
- Here are some standard function calls that use variable argument lists.

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
int execlp(const char *file, const char *arg,
    ...);
```

- See `man stdarg` for documentation on using variable argument lists. In particular, the header file contains a set of macros that define how to step through the argument list.
- See Section 7.3 in the K&R C book.

Variable Argument Lists in C (2)

Useful macros from `stdarg` header file.

- `va_list argptr;` is used to declare a variable that will refer to each argument in turn.
- `void va_start(va_list argptr, last);` must be called once before `argptr` can be used. `last` is the name of the last variable before the variable argument list.
- `type va_arg(va_list ap, type);` Each call of `va_arg` returns one argument and steps ap to the next; `va_arg` uses a type name to determine what type to return and how big a step to take.
- `void va_end(va_list ap);` Must be called before program returns. Does whatever cleanup is necessary.
- It is possible to walk through the variable arguments more than once by calling `va_start` after `va_end`.

Variable Argument Lists Example


```c
/* C-examples/varargs/test-varargs.c */
#include <stdio.h>
#include <stdarg.h>

void strlist(int n, ...)
{
  va_list ap;
  char *s;

  va_start(ap, n);
  while (1) {
    s = va_arg(ap, char *);
    printf("%s\n",s);
    n--;
    if (n==0) break;
  }
  va_end(ap);
}
int main()
{
  strlist(3, "string1", "string2", "string3");
  strlist(2, "string1", "string3");
}
```