CSCB09

Software Tools and Systems Programming

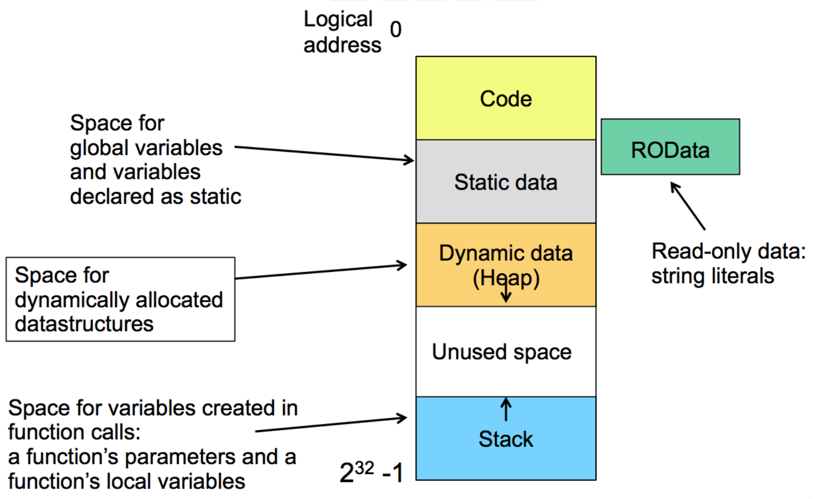
导师： VC

UTSC Week 02 Class | 2022/1/17

S P E E D U P E D U C A T I O N

Disclaimer

Memory


Logical address
0
Code
Space for global variables and variables declared as static
Static data
ROData
Dynamic data (Heap)
Read-only data: string literals
Space for dynamically allocated datastructures
Unused space
Space for variables created in function calls: a function's parameters and a function's local variables
Stack
2^{32} -1

Type	Size	Value Range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615
pointer	8 bytes	…

File I/O in C

File functions work with `FILE *` all the time:
FILE = Type representing stream state. Actual definition varies by compiler and OS, but most likely a **struct**, so we pass around **pointers**.

(Poor naming because it isn't about files, it's about contents and streams.
'STREAM' would be better. I should also say "stream functions" instead of "file functions".)

Header to `#include: <stdio.h>`

Open

`FILE *fopen(const char *filename, const char *mode)`

Mode	**meaning**	**if file exists**	**if not**
"r"	read	cool	error
"w"	write	truncate	create
"a"	append	cool	create
"r+"	read+write	cool	error
"w+"	write+read	truncate	create
"a+"	append+read	cool	create

"read" and "write" start at the beginning of the file.
"append" start at the end of the file.
"truncate" = erase original content.

Returns NULL if error. (This one you should always check.)

Close

`int fclose(FILE *stream)`

Returns 0 if success, EOF if error.

Why need to close:

1. Limit on how many streams are open per process.
2. Writing may be delayed (buffered) until closing. (More on buffering later.)
3. Windows: No two processes can open the same file.

Formatted I/O

```
int fprintf(FILE *stream, const char *format, ...)
int fscanf(FILE *stream, const char *format, ...)
```

In fact:

`printf(format, args)` = `fprintf(stdout, format, args)`
`scanf(format, args)` = `fscanf(stdin, format, args)`

`stdin` and `stdout` are pre-opened streams for standard-in and standard-out.

There is one more: `stderr`, so you can output error messages and leave 'stdout' for normal data/output.

(No need to manually close 'stdin', 'stdout', 'stderr'.)

* Usually just use 'perror' to print error message to stderr:
`void perror(const char *prefix)`

Character I/O

One single character:

```
int putchar(int c) /* stdout */
int putc(int c, FILE *stream)
int getchar(void) /* stdin */
int getc(FILE *stream)
```

Returns the character written/read if success, `EOF` if error or end of stream.

Important:

EOF fits in 'int' but not 'char'. Correct usage of 'getchar' and 'getc' involves:

1. Store return value in 'int' variable, not 'char'.
2. Test for EOF.
3. If not EOF, safe to down-convert to 'char'.

String I/O

`int fputs(const char *string, FILE *stream)`

Returns `EOF` if error. Also difference from ‘puts’: ‘fputs’ does not add newline at the end, ‘puts’ does.

`char *fgets(char *dest, int n, FILE *stream)`

Reads at most n − 1 characters or until (and including) newline.

Returns NULL if error or end of stream, dest if success.

Exercise 1: Why does it need you to provide n?

Exercise 2: Why n − 1 characters read, not n?

Arbitrary Data I/O

```
size_t fread(void *dest, size_t s, size_t n, FILE *stream)
size_t fwrite(const void *data, size_t s, size_t n, FILE *stream)
```

Read/Write n items, each item s bytes (ask `sizeof`); in-memory raw bytes used.

Returns how many items read/written.

Some use cases:

A whole array.
Record (struct) (single or array of).
Raw bytes (array of ‘unsigned char’ usually).

Cross-platform watchout:

The same raw bytes can mean different values on different platforms. Check compatibility!

Seeking

`int fseek(FILE *stream, long int i, int origin)`

Set the file pointer to the location `origin + i`

origin	**go to i bytes from:**
SEEK_SET	beginning
SEEK_END	end
SEEK_CUR	current position

Returns non-zero on error.

Error vs End-of-Stream Disambiguation

`getc`	returns EOF if end-of-stream or error.
`fgets`	returns NULL.
`fread`	returns < n.

etc.

How to tell end-of-stream from error:

`int feof(FILE *stream)` Returns true if end-of-stream status.

Important:

Does **not** predict for next read; only **remembers** for previous read. You must read before asking.

`int ferror(FILE *stream)` Returns true if error status.

Buffering

C file I/O delays writing:

accumulates data in buffer until large chunk, then requests kernel to write that chunk.

And hastens reading:

requests kernel to read a large chunk into buffer, then serves your puny ‘getc’ etc. from buffer.

Why:

Huge overhead per kernel request (system call, “syscall”) regardless of data size.
1 syscall for 1000 bytes beats 1000 syscalls for 1 byte each.

But can be disabled/reconfigured if your application needs.