C revision

Computing Lab

Indian Statistical Institute

Outline

1 Types

2 Memory layout: where are various things stored?

3 Allocating and deallocating memory

Types

- **ALL** data stored in memory as a sequence of 0s and 1s
- Variable's type determines how a sequence of 0s and 1s is interpreted

Example:

byte

0	1	0	0	0	0	0	1

- **Integer value:** 65
- **Character representation: 'A'**
- For arithmetic operations: interpreted as integer
  `x = x + 65` and `x = x + 'A'` mean the same thing
  `x = x - 48` and `x = x - '0'` mean the same thing
- For printing:
  - as integer (`printf("%d\n", x)`): `65` is printed
  - as character (`printf("%c\n", x)` or `putchar(x)`): `A` is printed

Built-in types: integer data types

Type	Size**	Minimum value	Maximum value
char	8	-2^7	$2^7 - 1$
short int	16	-2^{15}	$2^{15} - 1$
int	32	-2^{31}	$2^{31} - 1$
long int	32	-2^{31}	$2^{31} - 1$
long long int	64	-2^{63}	$2^{63} - 1$
unsigned char	8	0	$2^8 - 1$
unsigned short int	16	0	$2^{16} - 1$
unsigned int	32	0	$2^{32} - 1$
unsigned long int	32	0	$2^{32} - 1$
unsigned long long int	64	0	$2^{64} - 1$

** in bits (typical)

- Use `sizeof` if you need to know the actual size, e.g., `sizeof(a)`

Signed and unsigned

- **Unsigned types:** if a variable of unsigned type occupies k bits, its value can be between 0 and $2^k - 1$.
- **Signed types:**
  - Bit sequences stored are the same as for unsigned types (i.e., $B = b_{k-1}b_{k-2} \ldots b_1 b_0$).
  - **BUT** they are interpreted differently.
    - if $b_{k-1} = 0$, B is interpreted as for unsigned types;
    - if $b_{k-1} = 1$, B is interpreted as a *negative* number in **two's complement representation**.
  - Range of values: -2^{k-1} to $+(2^{k-1} - 1)$

Two's complement representation

x is a variable of integer type, stored in k bits.

- If $0 \leq x \leq 2^{k-1} - 1$, x is represented as usual in binary.
- If $x < 0$, it is represented in (k-bit) two's complement form by the number $\boxed{2^k - |x|}$ (in binary).

  Examples:
  - `char x = -1;` `x` is represented by $2^8 - 1 = 255 = 1111\ 1111$.
  - `char x = -128;` `x` is represented by $2^8 - 2^7 = 2^7 = 1000\ 0000$.
- Thumbrule to compute the k-bit two's complement representation of $x < 0$:
  1. Let B denote the k bit representation of $|x|$.
  2. Flip each bit of B to get B'.
  3. Add 1 to B'.

This is called the one's complement of B.

(Why does this work?)

Built-in types: “real” (floating point) numbers

Type	Size**
float	32
double	64
long double	128

- Use apostrophe `'` as thousand-separator, if required.
- See `float.h` for limits and other gory details.

  (use `locate` if required)
- At times behaviour may be counter-intuitive (see `counterintuitive-floats.c`).

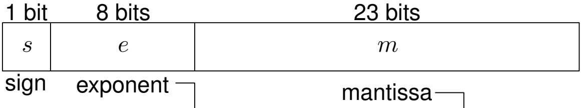
Examples:

Decimal notation	Exponential / scientific notation
`1.23456`	`3.45e67`
`1.`	`+3.45e67`
`.1`	`-3.45e-67`
`-0.12345`	`.00345e-32`
`+.4560`	`1e-15`

`e` means ‘10 to the power’

Built-in types: “real” (floating point) numbers

IEEE 754 representation



$$\equiv (-1)^s\ 1.m \times 2^{127+e}$$

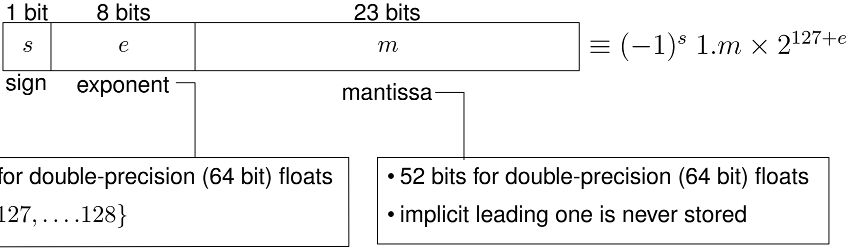
- 11 bits for double-precision (64 bit) floats
- $e \in \{-127, \ldots .128\}$

- 52 bits for double-precision (64 bit) floats
- implicit leading one is never stored

NOTE: The original exponent e plus a constant bias (127 for 32-bit rep.) is actually stored in the *exponent* field.

Built-in types: “real” (floating point) numbers

IEEE 754 representation



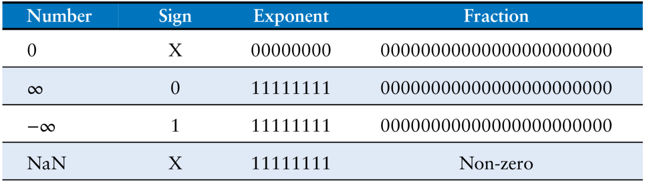
$\equiv (-1)^s \; 1.m \times 2^{127+e}$

- 11 bits for double-precision (64 bit) floats
- $e \in \{-127, \ldots .128\}$

- 52 bits for double-precision (64 bit) floats
- implicit leading one is never stored

NOTE: The original exponent e plus a constant bias (127 for 32-bit rep.) is actually stored in the *exponent* field.

Special numbers (SRC: HARRIS AND HARRIS, 2ND ED.)



Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

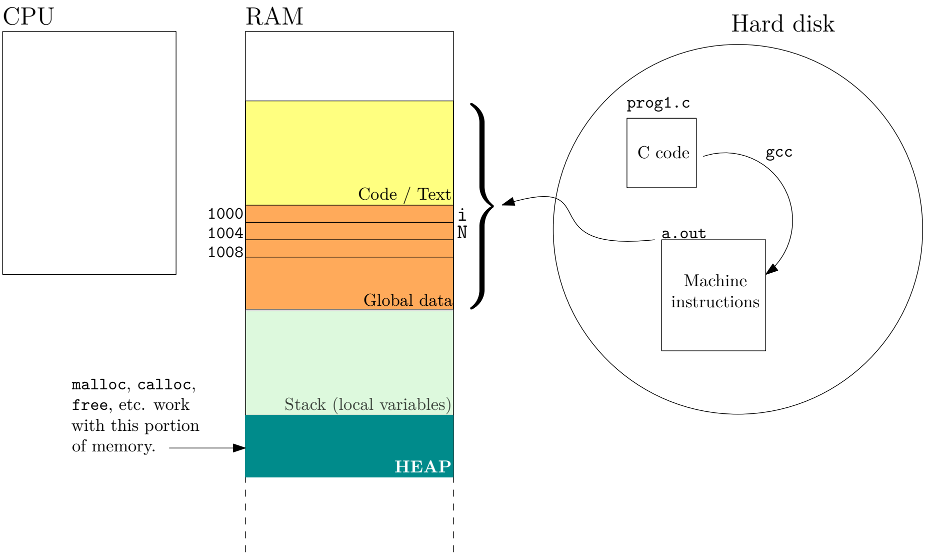
Outline

1 Types

2 Memory layout: where are various things stored?

3 Allocating and deallocating memory

Simplified view of a program’s memory



Activation records / stack frames

Where are the activation records (AR) stored?

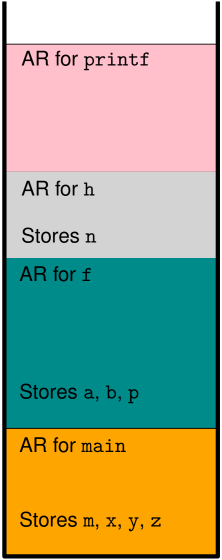
- Simple solution: AR == one fixed block of memory per function
  LATER: does not work for recursive functions
- Better solution: one block of memory per *function call*
  - AR allocated / deallocated when function is called / returns
  - variables created when function is called; destroyed when function returns
  - need to keep track of *nested* calls
  - function calls behave in *last in first out* manner
    ⇒ use *stack* to keep track of ARs

Activation stack

- Activation records / stack frames stored in a chunk of memory called *activation stack* or *call stack*
- When a function is called, its activation record is added to the end of the activation stack.
- When function returns, its activation record is removed.
- LATER: works for recursive functions

Activation stack: example

```
void main(void)
{  ...
   m = f(x, y*z);
   ...
}

int f(int a, int b)
{  ...
   if (a > 0)
      p = g(b);
   else
      p = h(b / 2);
   return p;
}

int g(int m)
{ ... }

int h(int n)
{ ... printf(...); ...}
```



AR for printf

AR for h

Stores n

AR for f

Stores a, b, p

AR for main

Stores m, x, y, z

Outline

1 Types

2 Memory layout: where are various things stored?

3 Allocating and deallocating memory

Topics to be covered

1. How do you allocate space for an array if you do not know (a reasonable upper bound on) the size when writing your program?
2. What to do if an array is full, and you need to store more elements?
3. Multi-dimensional arrays
4. Difference between `int a[M][N]` and `int **a;` ⟵ LATER

Variable length arrays (VLAs)

OK

```c
int num_elts;

scanf("%d", &num_elts);

int array[num_elts];
```

WRONG

```c
int num_elts;           // not initialised
int array[num_elts];    // num_elts == ???
```

Caution: (more detailed explanation later)

- Local variables allocated on stack
- Maximum stack size limited (often 8 MiB)
- Large local VLAs may not work

  Example: compile and run `large-vlas.c`; experiment with the array sizes in the program.

Alternative: use global / static / dynamic allocation

VLAs (contd.)

Reference: `https://en.cppreference.com/w/c/language/array`

- Expression evaluated + array allocated each time flow of control passes over the declaration
- Expression's value must be positive
- Array should not be accessed after declaration goes out of scope

  Exercise: is it actually deallocated?
- Cannot be members of structs / unions

Allocating memory

Syntax:

```
#include <stdlib.h>
(type *) malloc(n * sizeof(type))
(type *) calloc(n, sizeof(type))
(type *) realloc(ptr, n * sizeof(type))

free(ptr)
```

`malloc`, `calloc`, `realloc` return `void` pointers

Convenient macros: (see `common.h`)

```
#define Malloc(n,type) (type *) malloc( (unsigned) ((n)*sizeof(type)))
#define Realloc(loc,n,type) (type *) realloc( (char *)(loc), \
                            (unsigned) ((n)*sizeof(type)))
```

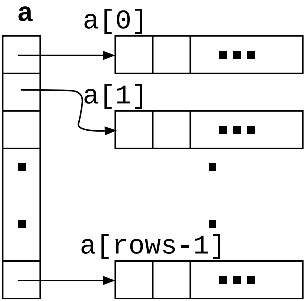
Extending an array using realloc

```
int *array, capacity = 100, num_elts = 0;

/* Initial allocation */
if (NULL == (array = Malloc(capacity, int))) {
    perror("out of memory");
    exit(1);  // instead of exit(0)
}

...

/* "Grow" the array when required */
if (num_elts == capacity) {
    capacity *= 2;
    if (NULL == (array = Realloc(array, capacity, int)) {
        perror("out of memory");
        exit(1);
    }
}
```

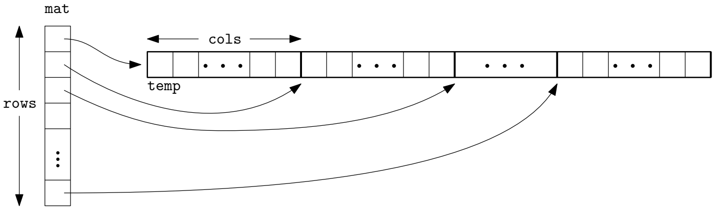
Multi-dimensional arrays

Multi-dimensional array = array of arrays = pointer to pointer

```
int **a, i;
a = (int **) malloc(rows * sizeof(int *));
for (i = 0; i < rows; i++)
    a[i] = (int *) malloc(cols * sizeof(int));
```



Multi-dimensional arrays: row-major storage



```
int ii;
int *temp;
if (NULL == (temp = (int *) malloc(rows*cols*sizeof(int))) ||
    NULL == (mat = (int **) malloc(rows * sizeof(int *))))
    ERR_MESG("Out of memory");
for (ii = 0; ii < rows; temp += cols, ii++)
    mat[ii] = temp;
```

Exercises

1. Work out the review questions on slide 4 (of 6) in `C-arrays.pdf`.
2. See slide 5 (of 6) in `C-arrays.pdf`. Print the address / location of "Style 1" and "Style 2" strings.
3. Print addresses of local and global variables, as well as variables allocated on the heap.
4. Experimentally determine / demonstrate the direction of stack growth for your environment.
5. Read up about stack smashing, if interested.