Introduction to the C programming language

(for the Operating Systems lecture)

Hans P. Reiser

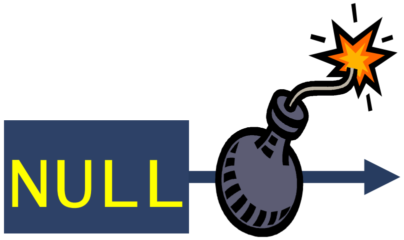
Introduction

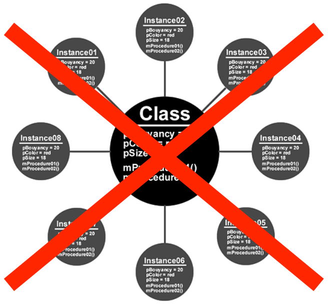
- C
  - A general-purpose language
  - Developed beginning of 1970s
  - Designed for implementing system software
  - Widely used programming language
- Notable properties
  - Procedural language
  - Not type-safe, memory access and addressing via pointers
  - Compound operators (`++, --, +=, >>=,` …)
  - Compact notation:

```
int c=0,b;
while((b=fgetc(f))!=EOF)c+=(b==10)?1:0;
fseek(f,0,SEEK_SET);
```

Why C ?

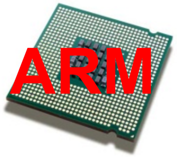
- C is
  - antique
  - not type-safe
  - not object-oriented
  - error-prone and tedious
  - … AJAX, Python, Java, PHP, C++, C#, ObjC sooo much better





Why C ?

- But C is also
  - powerful
  - efficient
  - close to the machine
  - standards-compatible, portable
  - widely used for
    - OSes, embedded systems
    - libraries
    - anywhere where (space/time) efficiency matters
  - foundation for many follow-on languages (C++, C#, Java)

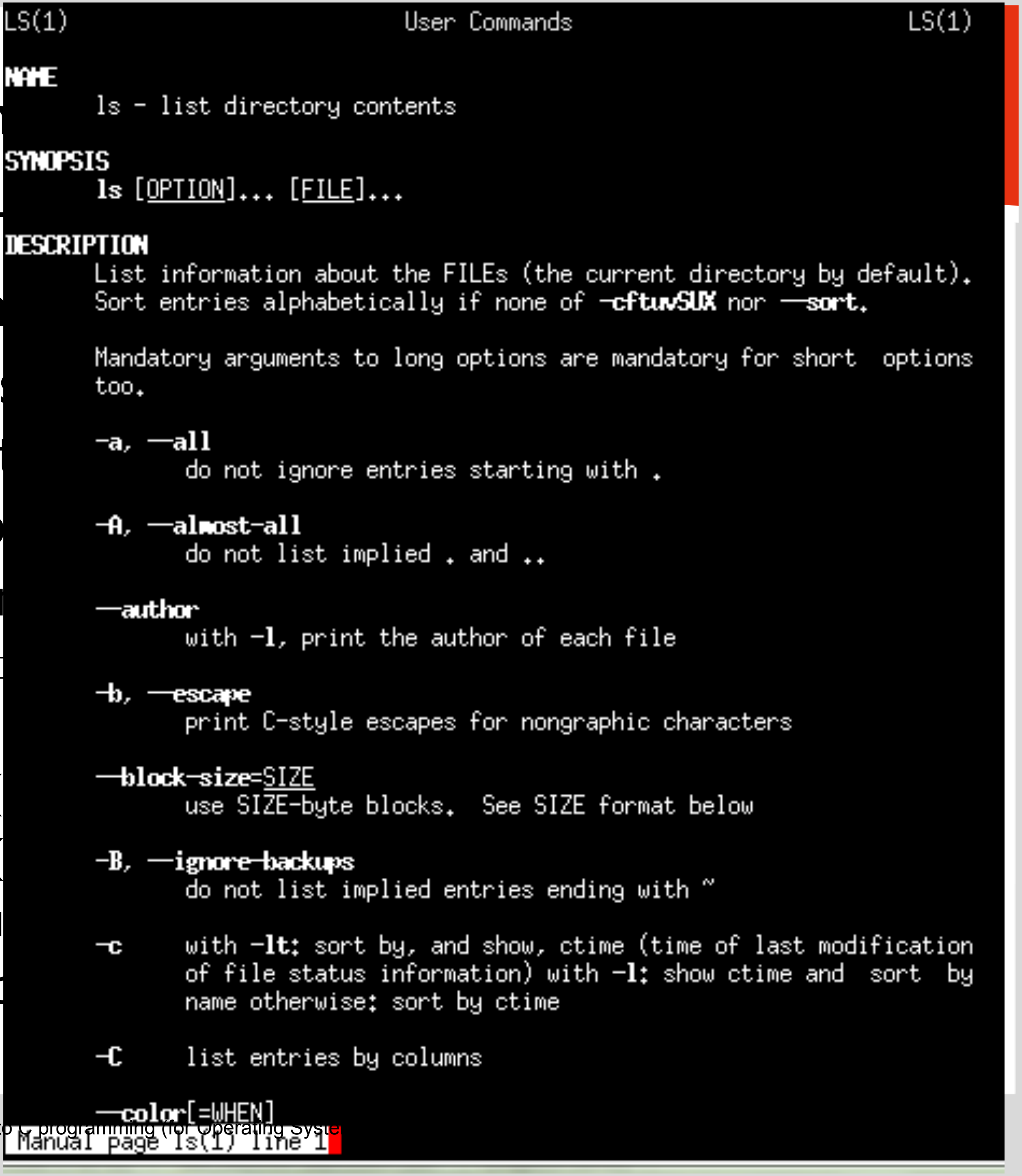


Introduction / Getting Help

- This lecture is NOT is not a complete reference to C.
- I assume you already know some Java or C.
- During homework assignments, get help as you need:
  - Library calls/system calls, parameters, return values
  - UNIX man(ual) page. Start with `man man`.
  - man page sections (`man 1 ls`):
    - 1 commands (`ls, gcc, gdb`)
    - 2 system calls (`read, gettimeofday`)
    - 3 library calls (`printf, scanf`)
    - 5 file formats (`passwd`)
    - 7 miscellaneous (`signal`)

Introduction / Gettin

- This lecture is NOT
- I assume you alrea
- During assignments
  - Library calls/ syst
  - UNIX man(ual) p
  - man page sectio
    - 1 commands (
    - 2 system calls
    - 3 library calls (
    - 5 file formats (
    - 7 miscellaneou
  - Search for man-p



```
LS(1)                         User Commands                         LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort.

       Mandatory arguments to long options are mandatory for short  options
       too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              use SIZE-byte blocks.  See SIZE format below

       -B, --ignore-backups
              do not list implied entries ending with ~

       -c     with -lt: sort by, and show, ctime (time of last modification
              of file status information) with -l: show ctime and  sort  by
              name otherwise: sort by ctime

       -C     list entries by columns

       --color[=WHEN]
 Manual page ls(1) line 1
```

Getting Help

- C syntax/semantics
  - “The C Programming Language” by Kernighan and Ritchie (“K&R”)
- Thorough guide to UNIX programming
  - “Advanced Programming in the UNIX Environment” by Stevens and Rago.

Hello World!

```
#include <stdio.h>
int main (void)
{
       printf ( ”Hello World!\n” );
       return 0;
}
```

- `#include` preprocessor (inserts contents of file).
- `stdio.h` contains the declaration of printf.
- `main` program starts here.
- `void` keyword for absence of arguments
- `{ }` basic blocks / scope delimiters.
- `printf` prints to the terminal.
- `‘\n’:` newline character.
- `return` leave function, give return value.

Compiling and running Hello World!

```
$gcc helloworld.c -o helloworld
$./helloworld
Hello World!
```

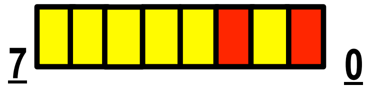
- Compilation:
  - Generating binary executable from source code
  - Comprises two main steps (besides preprocessor)
    - Generating binary object file for each source code file
    - Linking binary object files, resolving all addresses
- Execution
  - Operating system launches binary executable
  - Contains processor instructions (arch-specific, eg. x86)
  - May load libraries as needed

Basic Data Types

- Basic data types in C:

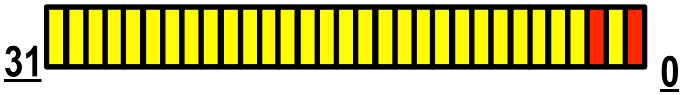
```
char c = 5; char c = ‘a’;
```

- char: one byte, usually for characters



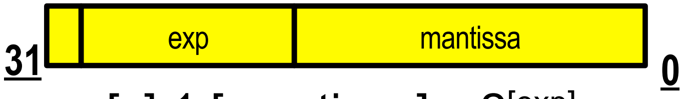
```
int i = 5; int i = 0xf; int i = ‘a’;
```

- int: usually 4 bytes, holds integers



```
float f = 5; float f = 5.5; double d = 5.98798;
```

- float: 4 bytes, floating point number
- double: 8 bytes, double precision floating point number



[s] 1.[mantissa] x $2^{[exp]}$

Basic Data Types

- Examples

```
int   i = 5/2;        // i = 2;
```

- integer logic, no decimal places, no rounding

```
float f = 5.0f/2;        // f = 2.5f
```

- decimal logic for float and double

```
char  a = ‘a’/2;      // a = 97 / 2 = 48
```

- remember, chars are one-byte numbers
- “character” meaning is interpreted by the console (ASCII table, ’a’ = 97)

signed vs. unsigned

- Can specify properties via keywords:

```
signed int i = -5;            // i=-5
unsigned int j = 100-200;     // j=4294967196
```

- **signed** or **unsigned** arithmetic (note the wrap)

```
short int i = 1024;           //-32768…32767
long  int j = 1024;          // -2147483648…2147483647
```

- **short** or **long** word size

	short int	int	long int	long long
32-bit architecture	16	32	32	64
64-bit architecture	16	32	64	128

- note: ranges and bitsizes vary with architecture

sizeof, inttypes.h, const, volatile

- Other properties

```
sizeof int; sizeof long int;  //4 and 4 on x86 32-bit
```

- Use **sizeof** to determine variable size

```
#include <inttypes.h>
int8_t i; uint32_t j;
```

- Use types from inttypes to be sure about sizes

```
const int i=5;
```

- variable is **const**ant, modification will raise compiler error

```
volatile int i=5;
```

- variable **volatile**, may be modified elsewhere
  - for example by different program in shared memory
  - important for CPU caches, registers and assumptions thereof

static

- Other properties:

```c
int myroutine(int j) {
        int i=5;
        i = i+j;
        return i;
}

k = myroutine(1); // k = 6;
k = myroutine(1); // k = 6:
```

```c
int myroutine(int j) {
        static int i=5;
        i = i+j;
        return i;
}

k = myroutine(1); //  k = 6;
k = myroutine(1); //  k = 7:
```

- basic block / function-local variables (eg. int i)
    - placed on stack or in registers
- **not so if** variable **static**
    - (if applied to local variables within function or basic block)
    - makes variable persistent across multiple invocations

Characters, strings, printf

- In C, characters are encoded as 1-byte “numbers” (char)

```
char c = ‘a’;
putc(c);
```

- Console driver translates those numbers into characters
- Uses ASCII table for that purpose

```
printf(“Hello”);
```

- Library call ‘printf’ from stdlib.h to print strings

```
int i=5; float f=2.5;
printf(“The numbers are i=%d f=%f”, i, f);
```

- Comprised of a format string and arguments
- Format string may contain format identifiers (%d)
- man 3 printf

Characters, strings, printf

- remember, characters are just “numbers”
- ASCII table translates those numbers (man ascii)

```
char c = ‘a’;
char c = ‘a’ + 1;  // c = ‘b’, since ‘b’ follows ‘a’ in ASCII
```

- Assign characters to variables via single quote ′
- Can calculate with characters

```
\n      newline                      \”     double quote
\t      tab                          \0     NULL, end of string
\’      single quote
```

- Special ASCII characters encoded via leading backslash

Characters, strings, printf

- remember, characters are just “numbers”
- ASCII table translates those numbers (man ascii)

```
char c = ‘a’;
char c = ‘a’ + 1;  // c = ‘b’, since ‘b’ follows ‘a’ in ASCII
```

- Assign characters to variables via single quote '
- Can calculate with characters

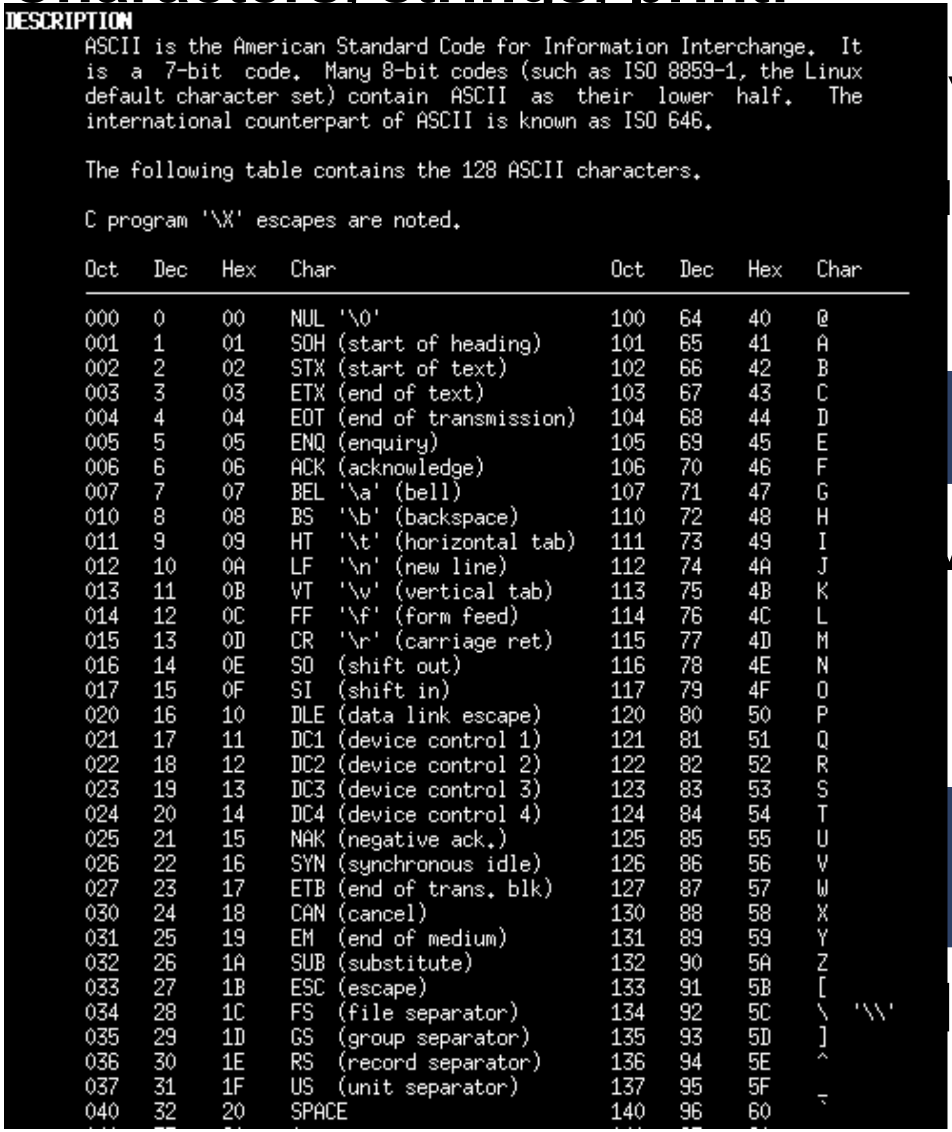
```
\n     newline                              \”     double quote
\t     tab                                  \0     NULL, end of string
\’     single quote
```

- Special ASCII characters encoded via leading backslash

Characters, strings, printf

“numbers”

mbers

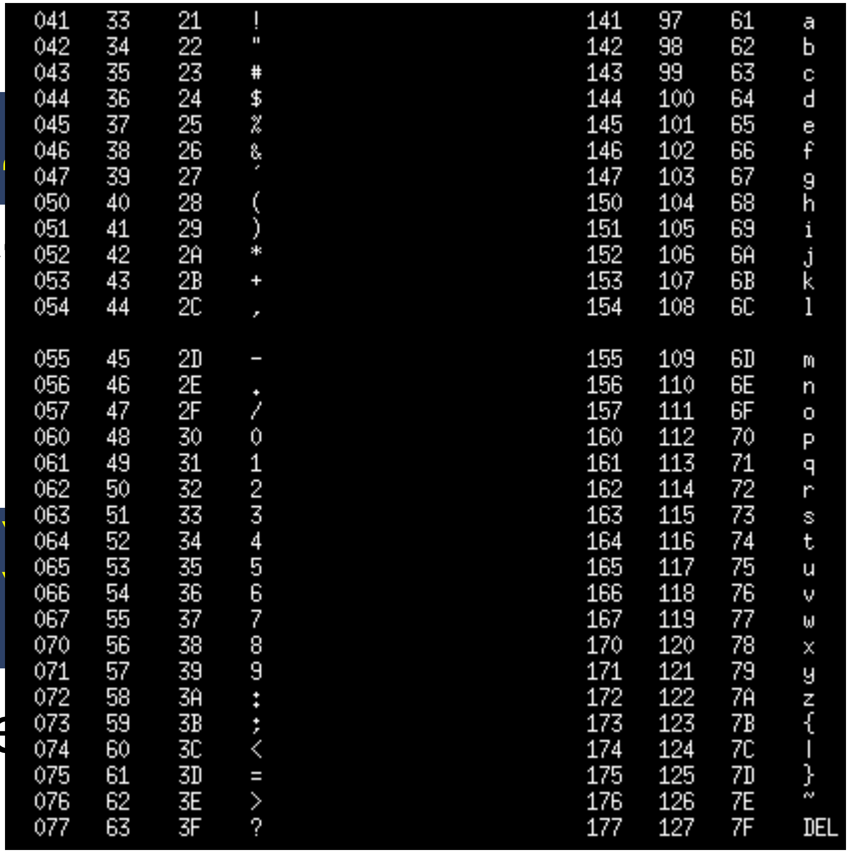


```
DESCRIPTION
       ASCII is the American Standard Code for Information Interchange.  It
       is  a  7-bit  code.  Many 8-bit codes (such as ISO 8859-1, the Linux
       default character set) contain  ASCII  as  their  lower  half.   The
       international counterpart of ASCII is known as ISO 646.

       The following table contains the 128 ASCII characters.

       C program '\X' escapes are noted.

       Oct   Dec   Hex   Char                        Oct   Dec   Hex   Char
       ------------------------------------------------------------------------
       000   0     00    NUL '\0'                    100   64    40    @
       001   1     01    SOH (start of heading)      101   65    41    A
       002   2     02    STX (start of text)         102   66    42    B
       003   3     03    ETX (end of text)           103   67    43    C
       004   4     04    EOT (end of transmission)   104   68    44    D
       005   5     05    ENQ (enquiry)               105   69    45    E
       006   6     06    ACK (acknowledge)           106   70    46    F
       007   7     07    BEL '\a' (bell)             107   71    47    G
       010   8     08    BS  '\b' (backspace)        110   72    48    H
       011   9     09    HT  '\t' (horizontal tab)   111   73    49    I
       012   10    0A    LF  '\n' (new line)         112   74    4A    J
       013   11    0B    VT  '\v' (vertical tab)     113   75    4B    K
       014   12    0C    FF  '\f' (form feed)        114   76    4C    L
       015   13    0D    CR  '\r' (carriage ret)     115   77    4D    M
       016   14    0E    SO  (shift out)             116   78    4E    N
       017   15    0F    SI  (shift in)              117   79    4F    O
       020   16    10    DLE (data link escape)      120   80    50    P
       021   17    11    DC1 (device control 1)      121   81    51    Q
       022   18    12    DC2 (device control 2)      122   82    52    R
       023   19    13    DC3 (device control 3)      123   83    53    S
       024   20    14    DC4 (device control 4)      124   84    54    T
       025   21    15    NAK (negative ack.)         125   85    55    U
       026   22    16    SYN (synchronous idle)      126   86    56    V
       027   23    17    ETB (end of trans. blk)     127   87    57    W
       030   24    18    CAN (cancel)                130   88    58    X
       031   25    19    EM  (end of medium)         131   89    59    Y
       032   26    1A    SUB (substitute)            132   90    5A    Z
       033   27    1B    ESC (escape)                133   91    5B    [
       034   28    1C    FS  (file separator)        134   92    5C    \  '\\'
       035   29    1D    GS  (group separator)       135   93    5D    ]
       036   30    1E    RS  (record separator)      136   94    5E    ^
       037   31    1F    US  (unit separator)        137   95    5F    _
       040   32    20    SPACE                       140   96    60    `
```

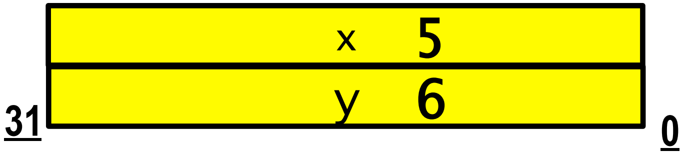


```
 041    33    21    !                                   141    97    61    a
 042    34    22    "                                   142    98    62    b
 043    35    23    #                                   143    99    63    c
 044    36    24    $                                   144    100   64    d
 045    37    25    %                                   145    101   65    e
 046    38    26    &                                   146    102   66    f
 047    39    27    '                                   147    103   67    g
 050    40    28    (                                   150    104   68    h
 051    41    29    )                                   151    105   69    i
 052    42    2A    *                                   152    106   6A    j
 053    43    2B    +                                   153    107   6B    k
 054    44    2C    ,                                   154    108   6C    l

 055    45    2D    -                                   155    109   6D    m
 056    46    2E    .                                   156    110   6E    n
 057    47    2F    /                                   157    111   6F    o
 060    48    30    0                                   160    112   70    p
 061    49    31    1                                   161    113   71    q
 062    50    32    2                                   162    114   72    r
 063    51    33    3                                   163    115   73    s
 064    52    34    4                                   164    116   74    t
 065    53    35    5                                   165    117   75    u
 066    54    36    6                                   166    118   76    v
 067    55    37    7                                   167    119   77    w
 070    56    38    8                                   170    120   78    x
 071    57    39    9                                   171    121   79    y
 072    58    3A    :                                   172    122   7A    z
 073    59    3B    ;                                   173    123   7B    {
 074    60    3C    <                                   174    124   7C    |
 075    61    3D    =                                   175    125   7D    }
 076    62    3E    >                                   176    126   7E    ~
 077    63    3F    ?                                   177    127   7F    DEL
```

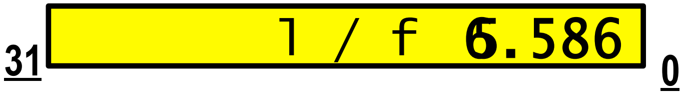
Compound data types

```
struct coordinate {
       int x;
       int y;
}
```



- **struct**ure: Collection of named variables of different types

```
union longorfloat {
       long l;
       float f;
}
```



- **union**: *single* variable that can have multiple types
- Note the difference between struct and union!

  sizeof c = 2*sizeof int vs. sizeof lf = max(sizeof float, sizeof long)

```
struct coordinate c;
c.x = 5;
c.y = 6;
```

```
union longorfloat lf;
lf.l = 5;
lf.f = 6.586;
```

- Members are accessed by name

Functions

```
unsigned int sum(unsigned int a, unsigned int b) {
        return a+b;
}
```

- Functions encapsulate functionality (reuse)
- Functions structure code (reduced complexity)
- Functions must be **declared** and **defined**

```
unsigned int sum(unsigned int a, unsigned int b);
```

  - Declaration states the signature (return type, name, params)

    <return type> function name ([<arg1> [, <arg2>[. . .]]]);

```
unsigned int sum(unsigned int a, unsigned int b) {
        return a+b;
}
```

  - Definition states the implementation
  - Definition implicitly declares the function

Declaration vs. definition

- Example: declaration of function other file

```
int sum(int a, int b)
{
        return a+b;
}
```

sum.c

```
#include <stdio.h>

int sum(int a, int b);

int main(void)
{
   printf ( ”%d\n”, sum(1,2));
   return 0;
}
```

main.c

Declaration vs. definition

- Use header file for frequently used declarations

```
int sum(int a, int b);
```

mymath.h

```
#include “mymath.h”


int sum(int a, b)
{
       return a+b;
}
```

sum.c

```
#include <stdio.h>
#include “mymath.h”

int main(void)
{
   printf ( ”%d\n”, sum(1,2));

   return 0;
}
```

main.c

Declaration vs. definition

- Use **extern** to declare global variables defined elsewhere

```
int sum(int a, int b);
extern float pi;
```

mymath.h

```

#include “mymath.h”

float pi=3.1415927;
int sum(int a, b)
{
        return a+b;
}
```

sum.c

```
#include <stdio.h>
#include “mymath.h”

int main(void)
{
   printf ( ”%d\n”, sum(1,2));
   printf ( ”%f\n”, pi);
   return 0;
}
```

main.c

Static declaration

- Use **static** to limit scope to current file
(when applied to global variables and functions)

```
int sum(int a, int b);
extern float pi;
```

mymath.h

```

#include “mymath.h”

static float pi=3.1415927;
int sum(int a, b)
{
       return a+b;
}
```

sum.c

```
#include <stdio.h>
#include “mymath.h”

int main(void)
{
   printf ( ”%d\n”, sum(1,2));
   printf ( ”%f\n”, pi); X
   return 0;
}
```

main.c

Static declaration

- Use **static** to limit scope to current file
  (when applied to global variables and functions)

```
int sum(int a, int b);
```

mymath.h

```

#include “mymath.h”

static float pi=3.1415927;
int sum(int a, b)
{
       return a+b;
}
```

sum.c

```
#include <stdio.h>
#include “mymath.h”
static float pi=3.1415927;
int main(void)
{
   printf ( ”%d\n”, sum(1,2));
   printf ( ”%f\n”, pi); √
   return 0;
}
```

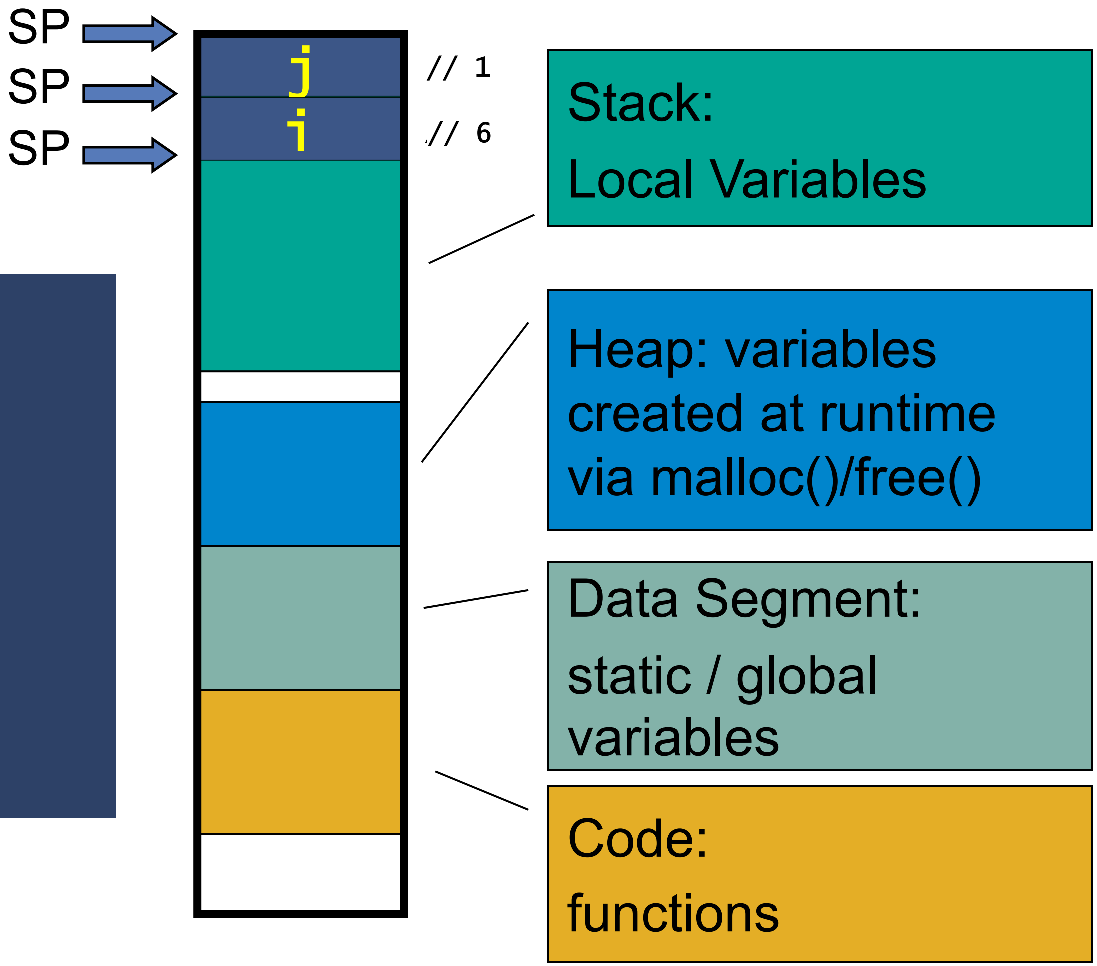
main.c

Stack/Heap/Data Segments and Variables

- A running program

```
int myroutine(int j) {
        int i=5;
        i = i+j;
        return i;
}

k = myroutine(1);
k = myroutine(1);
```



Stack:
Local Variables

Heap: variables created at runtime via malloc()/free()

Data Segment:
static / global variables

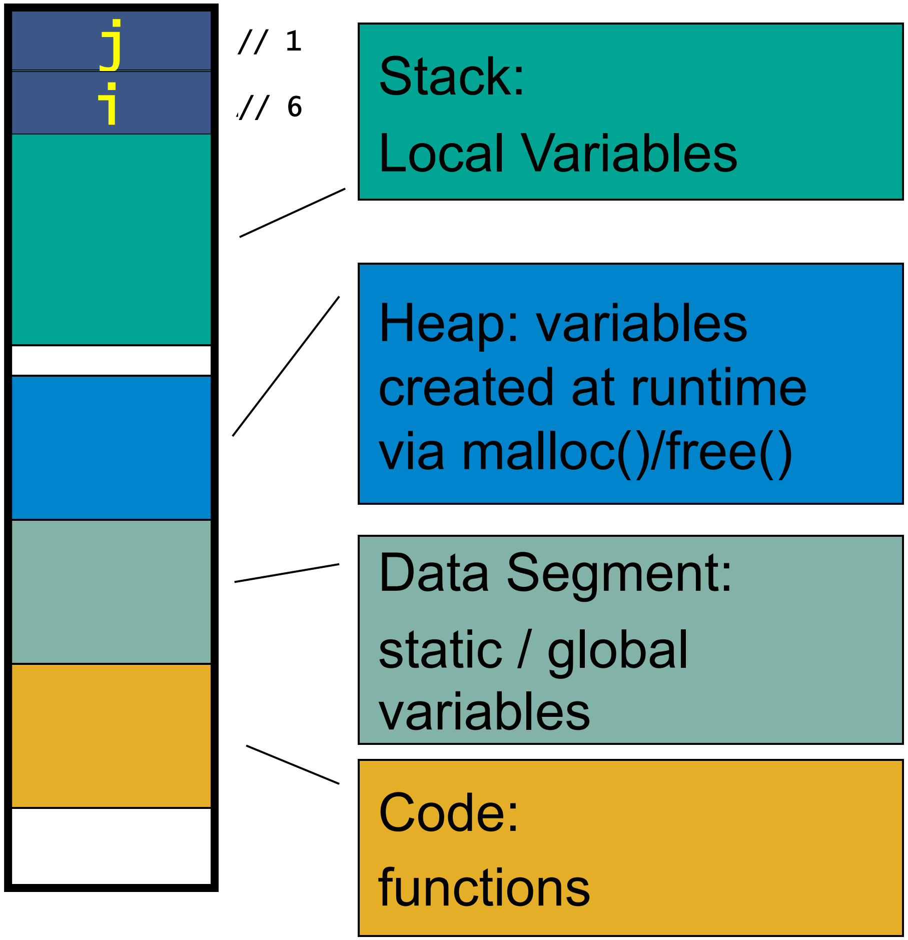
Code:
functions

Stack/Heap/Data Segments and Variables

- A running program

```
int myroutine(int j) {
        int i=5;
        i = i+j;
        return i;
}

k = myroutine(1);
k = myroutine(1);
```



Stack:
Local Variables

Heap: variables created at runtime via malloc()/free()

Data Segment:
static / global variables

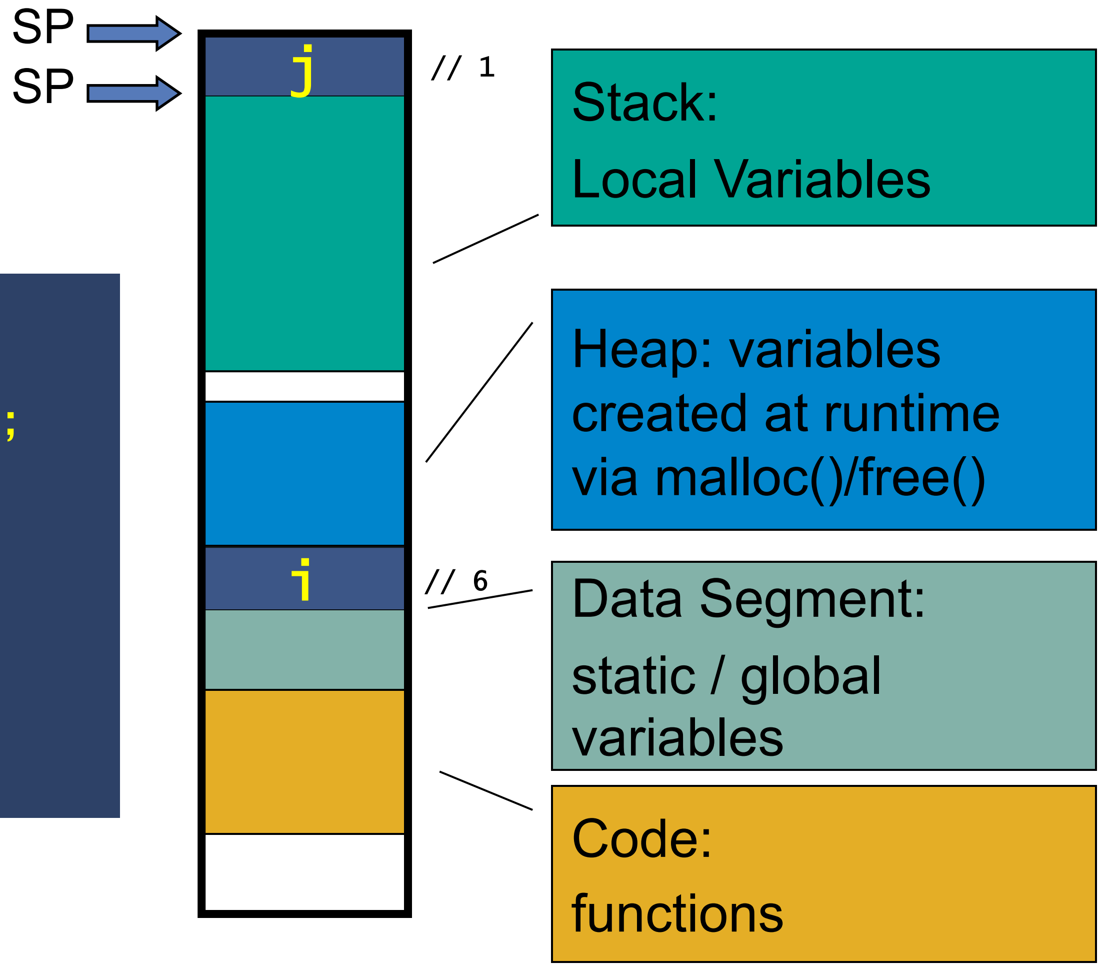
Code:
functions

Stack/Heap/Data Segments and Variables

- A running program

```
int myroutine(int j) {
       static int i = 5;
       i = i+j;
       return i;
}

k = myroutine(1);
k = myroutine(1);
```



Stack:
Local Variables

Heap: variables created at runtime via malloc()/free()

Data Segment:
static / global variables

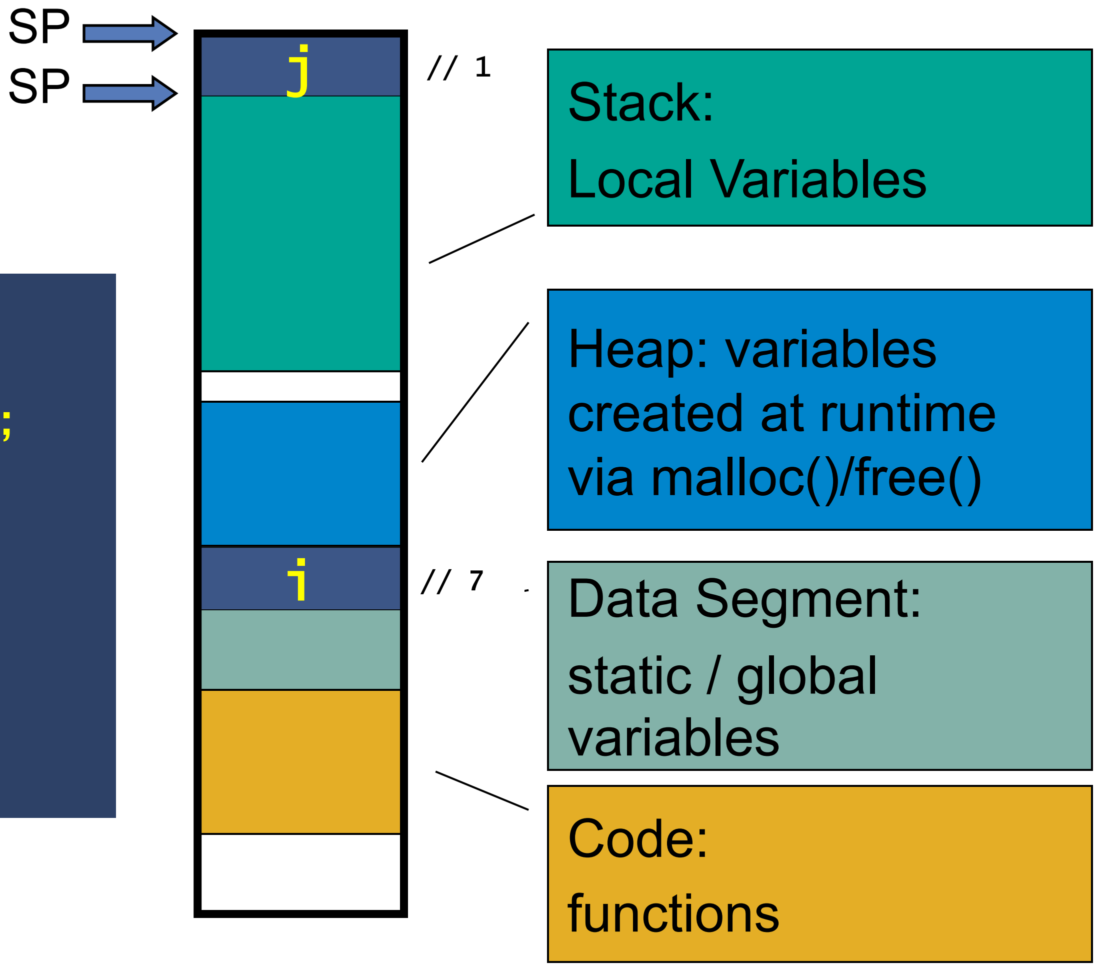
Code:
functions

Stack/Heap/Data Segments and Variables

- A running program

```
int myroutine(int j) {
        static int i = 5;
        i = i+j;
        return i;
}

k = myroutine(1);
k = myroutine(1);
```



Stack:

Local Variables

Heap: variables created at runtime via malloc()/free()

Data Segment:

static / global variables

Code:

functions

Function overloading

```
int sum(int a, int b) {
        return a+b;
}

int sum(int a, int b, int c) {
        return a+b+c;              X
}
```

- NO function overloading in C

```
sum.c:8:5: error: conflicting types for ‘sum’
sum.c:4:5: note: previous definition of ‘sum’ was here
```

```
int sum(int *summands, int size) {
        int sum = 0;
        int s = 0;
        for (s=0; s < size; s++)
                sum += *(summands+s);
        return sum;
}
```

- Use arrays or pointers ☺

Pointer

- Pointer: data type pointing to a value

```
int *p;
```

- pointer to an integer variable
- holds a memory address to a variable of type int

```
int a = 5;
int *q = &a;
```

- can be assigned to the address of an existing variable

```
int *p;
struct coordinate *c;
void *r;
```

- typically has a type, void denotes absence of type

```
int i  = *p;     // c = dereference(p) => 5
int x  = (*c).x; // x = dereference(c), member x
int x2 = c->x;   // short form of (*c).x
```

- can be dereferenced

Pointer

- Pointer: data type pointing to a value

```
int a=5;
```

```
int *p = &a;
```

```
int *q = 32;
```

```
int b = a+1;
```

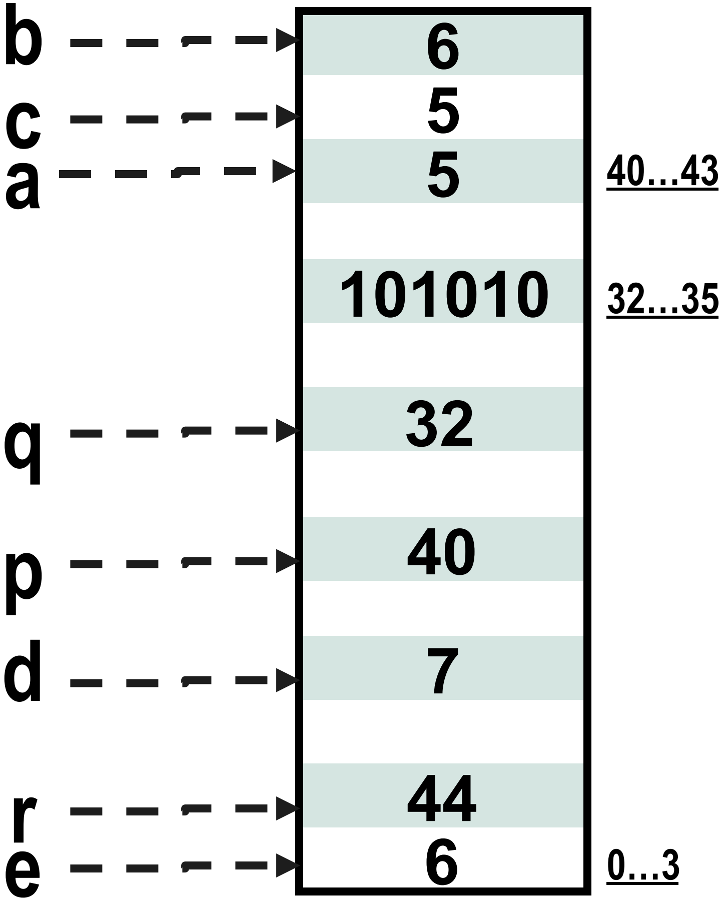
```
int c = *p;
```

```
int d = (*p)+2;
```

```
int *r = p+1;
```

```
int e = *(p+2);
```

Main memory



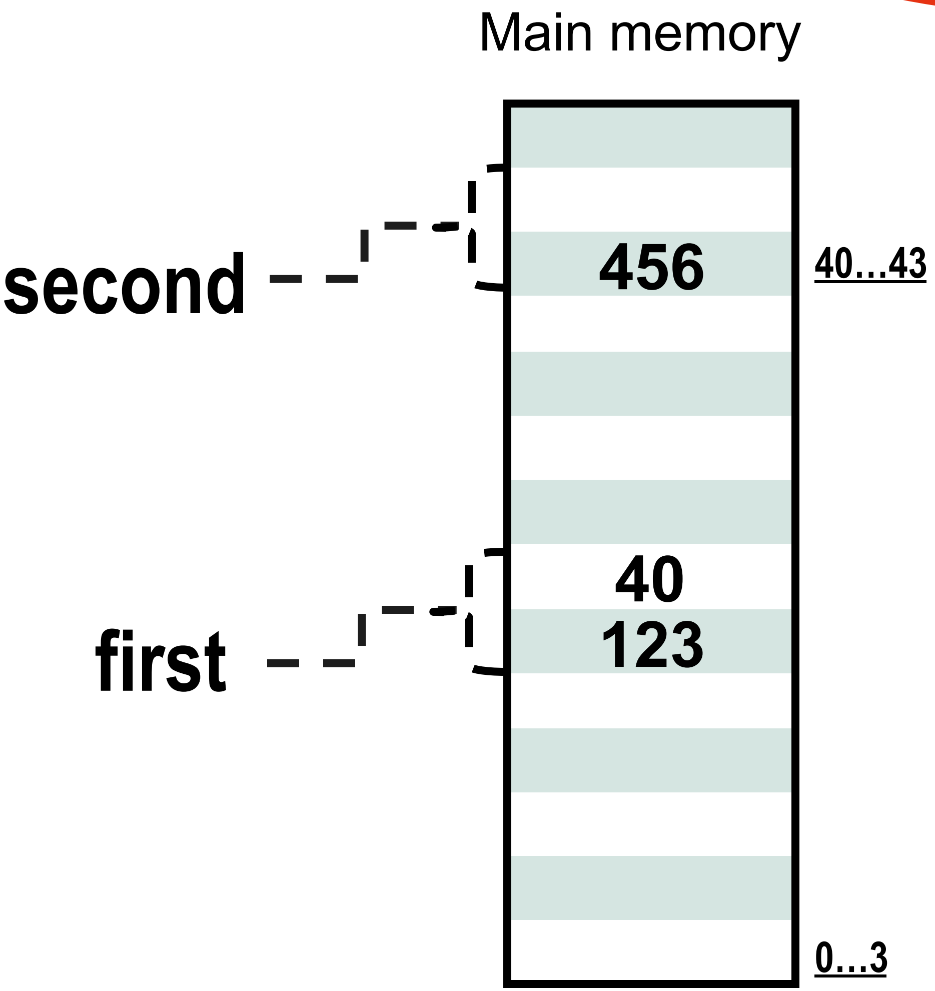
Example: linked list

- Linked list via next-pointer

```
struct ll {
  int item;
  struct ll *next;
};

struct ll first;
first.item = 123;

struct ll second;
second.item = 456;
first.next = &second;
```



Arrays

- Array: fixed number of variables *continuously laid out in memory*

```
int A[5];
```

- declare an array (and reserve space in memory)

```
A[4] = 25; A[3] = 24;
```

- assign 25 to last, 24 to first element

```
char c[] = { ‘a’, 5, 6, 7, ‘B’};
```

- initialize array, implicitly stating length

```
c[654] = ‘Z’;
```

- NO bounds checking at compile or run time
  (but may raise protection fault)

```
char *p = c;
*(p+1) = ‘Z’; p[3] = ‘B’;
```

- declare pointer to array; address elements via pointer

Array vs. pointer

- Pointer: data type pointing to a value

```
int A[3] = { 4, 5 , 6 };
```

```
int *p = A;
```

```
A[2] = 7;
```

```
p[2] = 8;
```

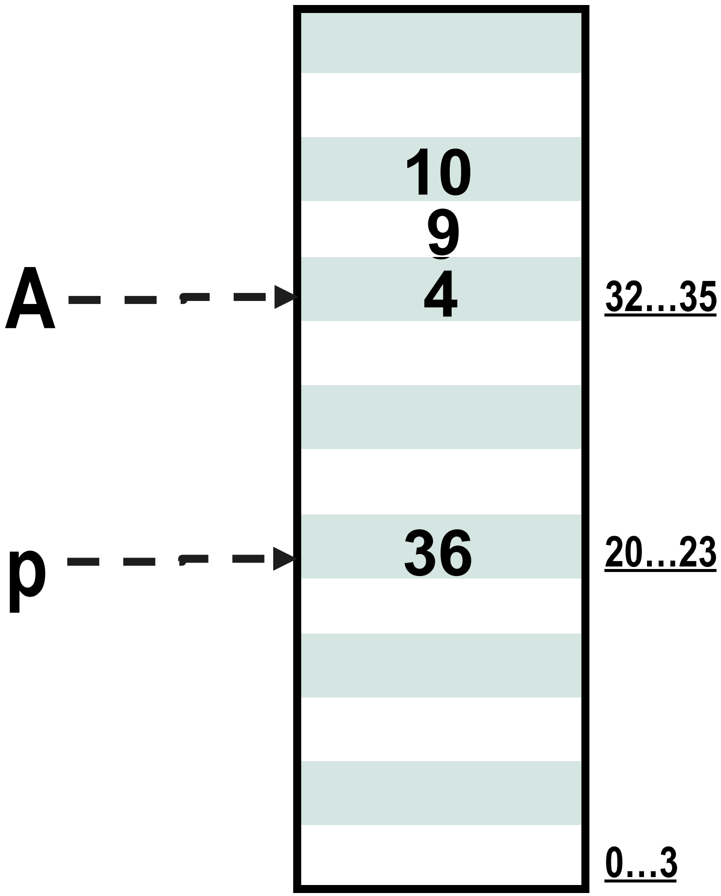
```
A = A + 1;
```
✗

```
p = p + 1;
```
✓

```
*p = 9;
```

```
p[1] = 10;
```

Main memory



Strings

- String: array of characters terminated by NULL (0)

```
char A[] = { 'J', 'a', 'n', '\0' };
char A[] = "Jan";
```

- declare and initialize string

```
const char *p= "Jan";
```

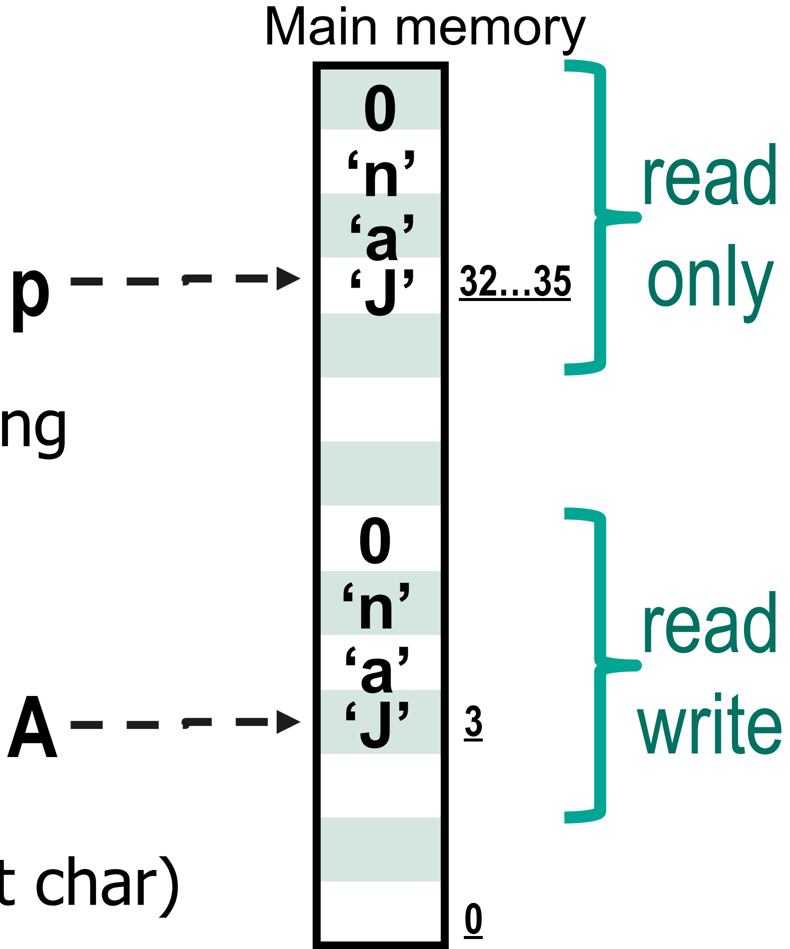
- declare const char pointer to string

```
A[2] = 'b';    ✓
```

- valid assignment

```
p[2] = 'b';
*(p+2) = 'b';  ✗
```

- both fail at compile time (p const char)
- Remember: pointer data type pointing to a value



Common string functions

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

- are defined in the header file string.h

```
size_t strnlen(const char *s, size_t maxlen)
```

- length of a string (up to n)

```
int strncmp(const char *s1, const char *s2, size_t n);
```

- compare two strings (up to n), return >0,0,<0

```
int strncpy(char *dest, const char *src, size_t n);
```

- copy a string (up to n)

```
 char *strtok(char *str, const char *delim);
```

- tokenize a string (eg. split line into words)

My first C routine

```
char* strncpy(char *dest, const char *src, size_t n){

  size_t i;

  for (i = 0 ; i < n && src[i] != '\0' ; i++)
    dest[i] = src[i];
  for ( ; i < n ; i++)
    dest[i] = '\0';
  return dest;
}
```

- Copies string src to dest up to n
- Uses a “for”-loop that
  - ends when n has been reached or src ends (whichever first)
  - copies, character-wise, src into dest
- Uses a second “for”-loop that zeroes out the rest of dest

Arithmetic and bitwise operators

```
a + b    a - b    a * b    a / b    a % b
```

```
a++;     a--;     a+=5;    a*=3;    a %=1;
```

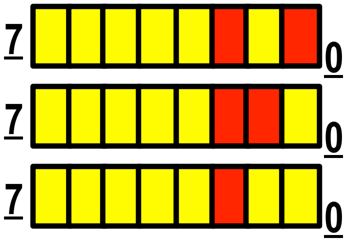
- arithmetic operators and their short forms

```
a=5;
if (a++ == 5) printf("Yes");
```

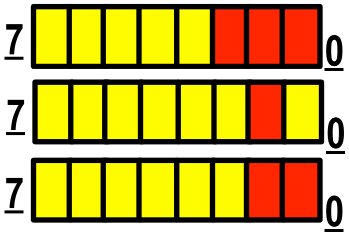
```
a=5;
if (++a == 5) printf("Yes");
```

- note the difference between pre- and post-increment

```
a & b    a | b    a >> b   a << b   a ^ b    ~a
```



```
a // 5
b // 6
a & b // 4
```



```
a | b  // 7
a >> 1 // 2
a ^ b  // 3
```

- logical operators often used for bit, address calculations

C routine using bit logic

```
uint8_t  bit_function(uint8_t val) {

       uint8_t mask = ~(1<<5);
       return val & mask;

}
```

7 🟨🟨🟨🟨🟨🟨🟨🟥 0	`1`
7 🟨🟨🟥🟨🟨🟨🟨🟨 0	`1<<5`
7 🟥🟥🟨🟥🟥🟥🟥🟥 0	`~(1<<5)`
7 🟨🟨🟥🟥🟨🟨🟨🟥 0	`val          // 49`
7 🟨🟨🟨🟥🟨🟨🟨🟥 0	`val & mask   // 17`

- mask out bit number 5

Loops, if-then-else

```
if ( a == b )
 printf(“Equal”);
else
 printf(“Different”);
```

```
if ( a == b )
 printf(“Equal”);
else {
 printf(“Different”); return 0;
}
```

- {} only needed for multiple statements

```
int i;
for (i=10; i>=10; i--)
 printf(“%d”, i+1);
```

```
int i=10;
while (i--)
  printf(“foo”);
```

```
int i=0;
do
  printf(“bar”);
While(i++ != 0);
```

- do-while-statement executed at least once

```
for (;;) {
  i = read();
  if (i>0)
    break;
  if (i==0)
    continue;
  do_something();
}
```

- with for-loops, can leave out any of initializer/expression/modifier
- use break and continue to exit/ skip

Expressions

```
if (<expression>)
while (<expression>)
for (<initializer>; <expression>; <modifier>)
```

- Operators and operands build expressions

```
if (n = 1)
```

```
while (n--)
```

```
for (n=10;n>0;n-=c)
```

- Assignments are expressions

```
if (n > 0)
```

```
while (n++ < 0)
```

```
while (n != 0)
```

- Comparisons are expressions
  - (n++ < 0) extends to 1 if n < 0 and to 0 otherwise, then increments n

```
if (n == 0)
```

```
if (n = 0)
```

```
if ((n = read()) < 0)
```

  - Note the difference between == and = !
  - Expressions can be nested (last example)

Logical operators

```
if ( a == 0 || b == 0)
```

```
if ( a > 0 && b < 0 )
```

```
if (!(a == 0))
```

- || logical OR
- && logical AND
- ! logical NOT

```
a = 0; b = 1;
if ( a == 0 || b == 0)
```

```
a = 0; b = 1;
if ( a != 0 && (b == read()) )
```

- Note: operators are evaluated in non-strict manner
  - First example: b == 0 never evaluated
  - Second example: b == read() never evaluated

All C operators (in order of precedence)

()	[]	->	.							
!	++	--	+y	-y	*z	&=	(type)		sizeof	
*	/	%								
+	-									
<<	>>									
<	<=	>	>=							
==	!=									
&										
^										
|										
&&										
||										
?	:									
=	+=	-=	*=	/=	%=	&=	~=	|=	<<=	>>=
,										

Switch/case

```
char a = read();

switch (a) {
  case ‘1’:
    handle_1();
    break;
  case ‘2’:
    handle_2();
    //break
  default:
    handle_other();
    break;
}
```

- Use **switch/case** to differentiate multiple cases.
- Note: need break statement to exit switch-loop
- If not given, code will fall through
- Example: with a == ‘2’, code will execute both handle_2() and handle_other()

Type casting

```
int i = 5;
float f = (float) i;
```

```
int i;
char c = (char) i;
```

- Explicit type casting (possibly losing precision)

```
char c = 5;
int  i = c;
```

```
float f = 0.555f;
double d = f;
```

- Some types are casted implicitly (if no precision loss)

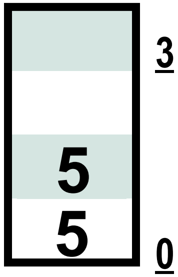
```
int i = 5;
float f = (float) (i / 2);
```

```
int i = 5;
float f = ((float) i) / 2;
```

- Watch out for precedence!

```

int i = 5;
char *p = (char *) &i;
*(p+1) = 5;
```



- Casting pointers changes address calculation!

Example Program Using File System Calls

```
/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);       /* open the source file */
if (in_fd < 0) exit(2);                /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
if (out_fd < 0) exit(3);               /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
    rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
if (rd_count <= 0) break;              /* if end of file or error, exit loop */
    wt_count = write(out_fd, buffer, rd_count); /* write data */
    if (wt_count <= 0) exit(4);        /* wt_count <= 0 is an error */
}

/* Close the files */
close(in_fd);
close(out_fd);
if (rd_count == 0)                     /* no error on last read */
    exit(0);
else
    exit(5);                           /* error on last read */
```

C preprocessor

- C preprocessor modifies *source* code
  - modified before compilation
  - based on preprocessor directives (usually start with #)

```
#include <stdio.h>
#include “mystdio.h”
```

- copies (literally!) contents of file to current file

Preprocessor search paths

```
#include <file>
```

- System include; search for file in:
  /usr/local/include
  libdir/gcc/*target*/*version*/include
  /usr/*target*/include
  /usr/include
  target: arch-specific path (i686-linux-gnu, x86_64-linux-gnu)
  version: gcc version (4.2.4, 4.6.1)
- Can add own paths with `–I<`*`dir`*`>`

```
#include “file”
```

- Local include; search in directory containing the *current file*
- Then in the paths specified by `–i<dir>`
- Then in system include paths described above

C preprocessor

```
#define PI 31415926535897
#define TRUE (1)
#define max(a,b) ((a > b) ? (a) : (b))
#define panic(str) do { printf(str); for (;;) } while(0);
```

- defines introduce replacements strings
  - Can have arguments (a,b, str)
  - Note: all based on string replacement!

```
#ifdef __unix__
# include <unistd.h>
#elif defined _WIN32
# include <windows.h>
#endif
```

```
#define DEBUG
#ifdef DEBUG
#define TRACE(x) printf(x)
#else
#define TRACE(x)
#endif
```

- defines can help structuring the code
  - quickly switch on/off include based on architecture or config
  - often leads to source code cluttering

Some notes on generated code

```
#include <stdio.h>

int val = 5;
int main(void) {
    val += 5;
    printf("%d\n", val);
    return val;
}
```

- A program marginally more complex than Hello World

```
$gcc -o myvar myvar.c
$./myvar
10
```

- Unsurprising result if compiled an run

```
$objdump -dhxs myvar
```

- Let’s (briefly) look at the generated code
- Objdump decodes and disassembles UNIX binaries

Some notes on generated code

```
myvar:     file format elf32-i386
Myvar
...
SYMBOL TABLE:
080483c4 g     F .text  00000034              main
0804a014 g     O .data  00000004              val
...
main():
/home/stoess/tmp/myvar.c:4
#include <stdio.h>
int  val = 5;
main() {
 80483c4:       55                      push   %ebp
 80483c5:       89 e5                   mov    %esp,%ebp
 80483c7:       83 e4 f0                and    $0xfffffff0,%esp
 80483ca:       83 ec 10                sub    $0x10,%esp
/home/stoess/tmp/myvar.c:5
    val += 5;
 80483cd:       a1 14 a0 04 08          mov    0x804a014,%eax
 80483d2:       83 c0 05                add    $0x5,%eax
 80483d5:       a3 14 a0 04 08          mov    %eax,0x804a014
/home/stoess/tmp/myvar.c:6
    printf("%d\n", val);
 80483da:       8b 15 14 a0 04 08       mov    0x804a014,%edx
 80483e0:       b8 c0 84 04 08          mov    $0x80484c0,%eax
 80483e5:       89 54 24 04             mov    %edx,0x4(%esp)
 80483e9:       89 04 24                mov    %eax,(%esp)
 80483ec:       e8 03 ff ff ff          call   80482f4 <printf@plt>
/home/stoess/tmp/myvar.c:7
    return val;
 80483f1:       a1 14 a0 04 08          mov    0x804a014,%eax
/home/stoess/tmp/myvar.c:8
}
```

- Function and variable names
- Translate to addresses

- Read, modify, write val

- Function call

Compiling and linking

```c
#include <stdio.h>

int  val = 5;
int main(void) {
    val += 5;
    printf("%d\n", val);
    return val;
}
```

myvar.c

```c
#include <stdio.h>

extern int  val;
int run_myvar2() {
    val += 10;
    printf("%d\n", val);
    return val;
}
```

myvar2.c

```
$gcc -o myvar myvar.c myvar2.c
```

- Compiles and links two source files

```
$gcc -c myvar.c myvar2.c
$ls *.o
myvar.o myvar2.o
```

- gcc –c compiles but doesn’t link
- generates two independent object files

Compiling and linking

```
myvar2.o:     file format elf32-i386
...
SYMBOL TABLE:
00000000         *UND*  00000000 val
... 00000000 <run_myvar2>:
   0:
   55                      push   %ebp
   1:   89 e5                       mov    %esp,%ebp
   3:   83 ec 18                    sub    $0x18,%esp
   6:   a1 00 00 00 00              mov    0x0,%eax
                        7: R_386_32         val
   b:   83 c0 05                    add    $0x5,%eax
   e:   a3 00 00 00 00              mov    %eax,0x0
                        f: R_386_32         val
  13:   8b 15 00 00 00 00           mov    0x0,%edx
                        15: R_386_32        val
  19:   b8 00 00 00 00              mov    $0x0,%eax
                        1a: R_386_32        .rodata
  1e:   89 54 24 04                 mov    %edx,0x4(%esp)
  22:   89 04 24                    mov    %eax,(%esp)
  25:   e8 fc ff ff ff              call   26 <run_myvar2+0x26>
                        26: R_386_PC32      printf
  2a:   a1 00 00 00 00              mov    0x0,%eax
                        2b: R_386_32        val
  2f:   c9                          leave
  30:   c3                          ret
```

- Object file contains code, space requirements
- External symbols unresolved
- Final addresses unresolved

Linking

```
$ld ... myvar.o myvar2.o –o myvar
```

- Linker (ld) “glues together” object files

```
$gcc myvar.o myvar2.o –o myvar
```

- Needs arch-/OS-specific params, invoke via gcc

```
--build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu -dynamic-linker /lib64/ld-linux-
x86-64.so.2 -z relro -o myvar /usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/
4.5.2/../../../crt1.o /usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/4.5.2/../../../
crti.o /usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/4.5.2/crtbegin.o -L/usr/lib/x86_64-
linux-gnu/gcc/x86_64-linux-gnu/4.5.2 -L/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/
4.5.2/../../.. -L/usr/lib/x86_64-linux-gnu myvar2.o myvar.o -lgcc --as-needed -lgcc_s --
no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/x86_64-linux-gnu/gcc/
x86_64-linux-gnu/4.5.2/crtend.o /usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/
4.5.2/../../../crtn.o
```

- This is (sort of) how gcc invokes ld

Libraries

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

int main(void) {
    float f = 0.555f;
    printf("%f", sqrt(f*4));
    return 0;
}
```

- Math header file contains declarations
- But not necessarily all definitions!

```
$gcc math.c -o math
/tmp/ccsGM8Gi.o: In function `main':
math.c:(.text+0x34): undefined reference to `sqrt'
collect2: ld returned 1 exit status
```

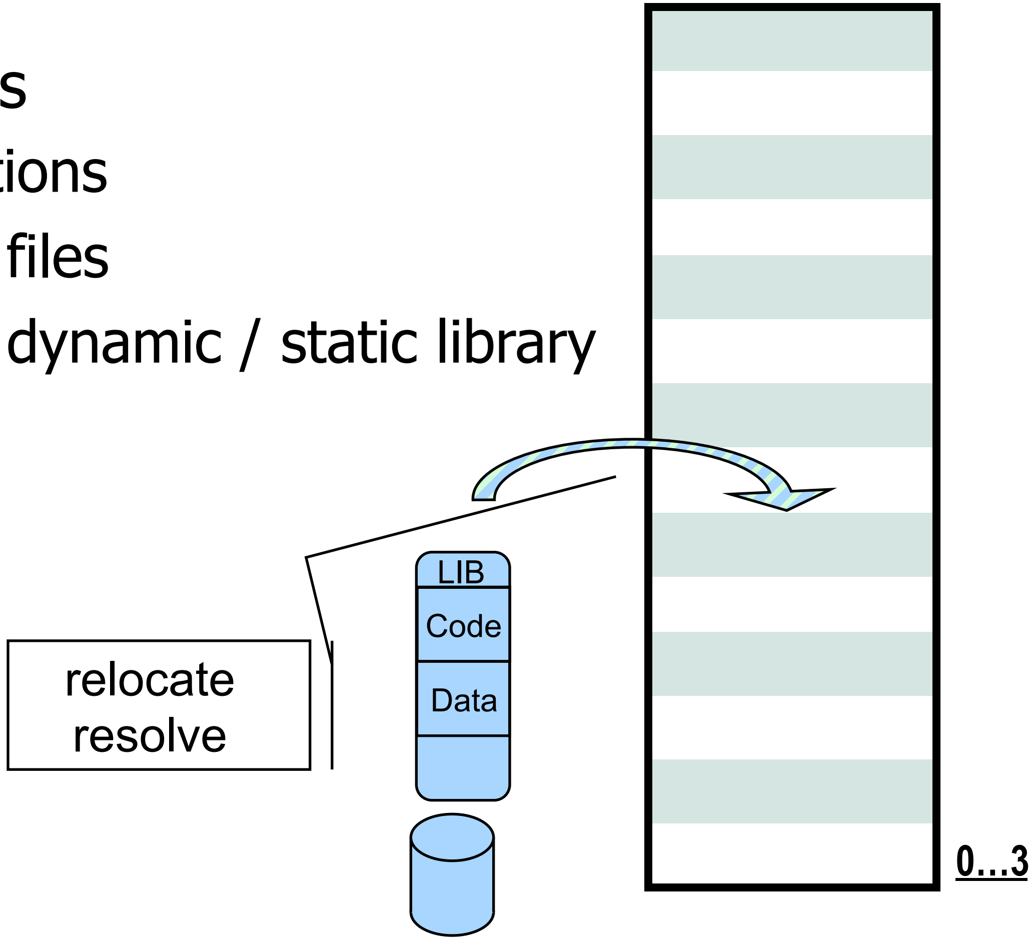
- Need to link math library

```
$gcc -lm math.c -o math
```

Libraries

```
$gcc –lm math.c
```

- Technically, a library is
  - a collection of functions
  - contained in object files
  - glued together in a dynamic / static library


relocate
resolve
LIB
Code
Data
0…3