C Programming Language

Šimon Řeřucha

v2.1

Background

"Each language has its purpose, however humble.
Each language expresses the Yin and Yang of software.
Each language has its place within the Tao.
But do not program in COBOL if you can avoid it."
The Tao Of Programming, 1.3

Programming Paradigms

- Imperative
  - Hard-wired / assembly - Machine code, Assemblers
  - Procedural - early Basics
  - Structured - Pascal, C, Fortran
  - Object-oriented - C++, Java
  - Event-driven, Automata-based, Graphical etc.
- Declarative
  - Functional – Common Lisp, Scheme, Haskell
  - Logic – Prolog

C Programming Language

- general-purpose programming language
- imperative, structured
- originally for system software on Unix
- one of most widely used
  - system software (e.g. Linux Kernel)
  - application software (with GUI library)
  - embedded systems

Brief History

- 1969-71: K&R C developed by AT & T Bell Labs
- late 70s-80s: C gets popular
- ...: C standard library
- 1990: adopted first ANSI C standard
- 2004+: Embedded C recommendations
- 12/2011: C11 standard

C-related Standards

- K&R C – pre ANSI (1971)
- ANS X3.159-1989 (1989) – known as ANSI C, C89 includes Standard Library
- ISO/IEC 9899:1990 (1990) – known as ISO C, C90 adopted ANSI standard
- ISO/IEC 9899:1999 (1999) – C99
- ISO/IEC 9899:2011 (2011) – C11

Unix-related Standards

- IEEE 1003.1 (last revision 2004)
  POSIX 1 – Unix API
- Sinle Unix Specification, SUS (1994, 97, 2002)
  incorporates POSIX.1
- FIPS 151-1, 151-1 – Federal Information Processing Std
  more specific than POSIX.1
- System V Interface Description, SVID3 / SVID4
- BSD

Embedded Systems Related Standards

- none.
- "Embedded C" technical reports.
- usually (subset of) Ansi C standard library + platform related extensions

C Basics

"There are only two kinds of programming languages: those people always bitch about and those nobody uses."
Bjarne Stroustrup.

General

- small number of keywords

  auto, break, case, char, const, continue, default, do, double, ese, enum, extern, float, for, goto, if, int ,long,

  register, return, short, signed, sizeof, static,struct, switch,typedef,union,unsigned, void, volatile, while
- lot of arithmetical and logical operators
- free-format, case sensitive
- simple structure aimed at algorithmization
- relies on libraries for a meaningfull operation

Variables

- static typed, weakly-enforced (implicit type conversion possible)
- single namespace, no sigil (e.g. `$var`)
- first class: accesible as a unit
  integer, structure
  but not array
- compound types – array, enum, struct
- lexical scope of visibility
- allows user-defined types

Functions

- procedure (function)-based
- declaration syntax similar to usage syntax
- fn definition outside functions
- basic polymorphism

Other Features

- low-level access to computer memory
- basic modularity – separated compilation
- preprocessor for macros, source text inclusion, conditional compilation
- linker for pre-compiled objects inclusion
- complex function delegated to (standard) library routines

What is not included

- object-oriented approach – but basic concepts are usable
- garbage collection – proper disposal of unused memory is left to programmer
- GUI library, multithreading, data persistence, lambda calculus

Lexical Conventions

- case sensitive, keywords are lower case
- comments: `/* multiline */` `// singleline`

Source character ser

- Letters: `a - z, A - Z, _`
- Digits: `0 -- 9`
- Punctuation: `~ ! @ # \% ^ & * ( ) - + = : ; " ' < > , . ? | / \ { } [ ]`
- Whitespace: space, horizontal tab, vertical tab, form feed, newline

Others possible, hardly portable (ASCII 127+, UNICODE)

Integer Constants

decimal `0 - 9` (default type)
minus sign - for negative numbers
Ex.: `15, 0, 1, -2`

octal `0 - 7`
begin with zero
Ex.: `065, 015, 0, 01`

hexadecimal `0 - 9, A - F`
begin with `0x` (case insensitive)
Ex.: `0x12, 0xDeadBeef, 0Xcd`

binary `0,1` non-standard!
its support depends on compiler
typically begin with `0b` (case insensitive)
Ex.: `0b10110010`

`~L`, `~U` suffixes to explicitly declare long / unsigned constant,
e.g. `129u, 12345LU`

Real Constants

decimal `15., 56.8, .84, 3.14`

exponential `5e6, 7e23`

- `double` by default
- `~F`, `~L` suffixes to explicitly declare float / long double constant, e.g. `3.14f`, `12e3L`

Character Constants

straight `'a'`, `'x'`, `'#'`

octal backslash + 3 octal digits, e.g. `\012`, `\x007`

hexa `\x` + 2 hex digits, e.g. `\x0A`, `\x00`

Well known constants:

- `\n` newline
- `\r` carriage return
- `\t` tab
- `\b` backspace
- `\\` backslash
- `\'` apostrophe
- `\0` zero character (e.g. string termination)

String Constants aka Literals

- delimited by quotes: ""
  ex. "A string","much longer string"
- automatic concatenation (chaining), e.g.
  "All this" " pieces"
  "will turn into one string!"
- Escape sequences used for input of special characters:
  "Be aware of \"nothing\" from girls' mouth!"
  "Do not mismatch slash '/' and backslash '\'"

Operators I.

- basic assignment: `=`
- arithmetics: `+`, `-`, `*`, `/`, `%`,
- relations: `==`, `!=`, `<`, `>`, `<=`, `>=`
- logical: `!`, `&&`, `||`
- bitwise logic: `~`, `&`, `|`, `^`
- shift operators: `<<`, `>>`
- advanced assignment:
  `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `~=`, `<<=`, `>>=`
- increment/decrement: `++`, `--`
- subexpresion: `()`

Operators II.

- conditional: `?` `:`
- calling function: `()`
- (de)referencing: `*`, `&`, `[]`
- sequencing: `,`
- type conversion: `(type)`
- addressing: `.`, `->`
- misc: `sizeof()`

Assignment

Basic operator: `=`
Advanced ops: `+=`, `*=`, `-=`, `...`

```
a = 1;
b = -2;
f = 1.34; // real number
f *= -2; // f is -2.68
a = max (1,3); // saving return value
```

Comparison operators

- == ... equality
- != ... inequality
- <, > ... less/bigger than
- <=, >= ... less/bigger or equal to

Zero is treated as FALSE, non-zero as TRUE – newer compilers provide `bool` datatype, but any integer is feasible for logic evaluation.

Assignment vs. Comparison

Caution!

Mismatching = and == is eternal source of error and grey hair.

E.g.:

`i == x` is TRUE if `i` is equal to `x`.

`i = x` is TRUE if `x` is nonzero.

Now days are brighter, compiler often issues a warning.

Ex.: Comparison

```
int i = 1, j = 1;

j = j && (i = 2); // true
j = j && (i == 3); // false
j = j || (i / 2); // true
j = !j && (i = i + 1); // false
```

Lazy Expression Evaluation

```
if ( y != 0 && x / y < z) ...
```

Generally, for `y = 0`, evaluation would end up with error. Due to *lazy evaluation*, the result is known before second part is evaluated.

Operator Precedence

	Operator	Description	Assoc.
1	`++ --`	Suffix increment and decrement	→
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
2	`++ --`	Prefix increment and decrement	←
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	Type cast	
	`*`	Dereference	
	`&`	Address-of	
	`sizeof`	Size-of	
3	`* / %`	Multiplication, division, and modulus	→
4	`+ -`	Addition and subtraction	
5	`<< >>`	Bitwise left shift and right shift	

Operator Precedence II.

	Operator	Description	Assoc.
6	`< <=`	For relational operators < and ≤ respectively	→
	`> >=`	For relational operators > and ≥ respectively	
7	`== !`	For relational = and ≠ respectively	
8	`&`	Bitwise AND	
9	`^`	Bitwise XOR (exclusive or)	
10	`|`	Bitwise OR (inclusive or)	
11	`&&`	Logical AND	
12	`||`	Logical OR	
13	*`c ? t : f`*	Ternary conditional	←
14	`=`	Direct assignment	
	`+= -=`	Assignment by sum and difference	
	`*= /= %=`	Assignment by product, quotient, and remainder	
	`<<= >>=`	Assignment by bitwise left shift and right shift	
	`&= ^= |=`	Assignment by bitwise AND, XOR, and OR	
15	`,`	Comma	→

Variables

Variable is a memory-stored element of data, that features:

- name (unique identifier)
- data type (known at compile-time)
- scope (visibility)

Identifiers

- identifiers start with alphabetic or underscore character, may contain numbers
- identifier begginging with underscore are often system reserved, not recommended (e.g. `_demo`)
- identifier length used to be limited (e.g. 8 or 32 characes)

Smart naming conventions

Use meaningfull names: long, self-descriptive
Use short names just for loop indices.

Basic datatypes

int	integer, signed or unsigned, min 16-bit, typ. 16–32-bit
char	smallest unit, integer type, typ. 8-bit
long int	min 32-bit, abbreviated as `long`
float	`single precision floating-point, typ 32-bit (sign + 8exp + 23man)`
double	`double precision floating-point, typ 64-bit (1 + 11 + 53)`
long double	`extended precision, typ 64--80-bit`

C99: adds `long long`, `bool`

Basic datatypes II.

- all integer types either `signed` (implicit) or `unsigned`
- signed range $(-2^{(n-1)}, 2^{(n-1)} - 1)$
- unsigned range $(0, 2^n - 1)$
- floating point types stored in form $m \times 2^n$

Examples

```
bool flag; // true = 1; c99 only
char c; // signed !
int i; // same as "signed int i"
unsigned int ui; // "unsigned ui" also valid
long int a; // "long a" also valid
long long int u; // c99 only
float f; //
long double xxx; // extended precision
```

Fixed-width datatypes (C99)

intN_t	integer of exactly N-bits, e.g. `int16`
uintN_t	`unsigned` variant
int_leastN_t, uint_leastN_t	`(unsigned) integer at least N-bits wide`
int_fastN_t, uint_fastN_t	`fastest type of (unsigned) integer at least N-bits wide`
intmax_t	`maximum width integer type`

Ex: `uint16_t, uint_fast32_t`

Functions

Function is a block of statements with defined arguments and return value. Any function features:

- identifier
- arguments (typed)
- return value (typed)
- scope
- declaration, definition

Functions

Declaration:

- tells the compiler that the function is defined (somewhere)
- specifies the name and type of arguments and return value (also referred to as function prototype)
- function must be declared before it is used

Definition:

- defines the function as a sequence of statements

Ex.: Functions

```
// declaration
double pi_times (int num);

// definition
double circ (int r){
    return pi_times( 2 * r ) ;
}

double pi_times (int num){
    return num * 3.14;
}
```

Function Scope

A function might be called at a certain point in source text when:

- the function is previously defined within same module
- the function is previously declared within same module
  - i.e. the function is defined within same module, or
  - i.e. the function exists in another module

The prototypes are typically kept in header files.

Variable Scope

A variable might be referenced at a certain point in source text when:

- it is previously defined within same function or globally
- it is previously declared within same function or globally
  - that indicates the variable is defined within same module, or
  - that indicates the variable exists in another module and is made accessible to other modules.

The local variables are defined within functions (even in blocks by newer standards), global variables are typically defined within modules' source text and declared in header files.

Void Type

- useful e.g. for explicit declaration of empty parameters or empty return value
- and more, as seen later

```
void print(int a);      // fn that returns nothing
int rand(void);         // fn w/ no parameters
```

Two-way Branching: IF

Syntax: `if(condition) then expr1;` or
Syntax: `if(condition) then expr1; else expr2;`

```
if (a == 1) b = 2;

if (a == 1){
    b = 2;
    c = 1;
}else{
    a = 1;
}
```

Multi-way Branching: SWITCH

```
switch(c){
  case 1: ...
          break;
  case 2: ...
          break;
  default: ...
          break;
}
```

Multi-way Branching: SWITCH II.

```
switch(c){
  case 'f': ... //
  case 'F': ...
        break;
  default: ...
        break;
}
```

- when break is ommited, evaluation continues in next branch

While loop

Syntax: `while(condition) expr1;`

- loop while `condition` holds
- `condition` evaluated before evaluating `expr`

```
a = 123;

while (a > 0){
    b = cool_fn(a);
}
```

Do .. While loop

Syntax: `do expr1 while(condition);`

- loop while `condition` holds
- `condition` evaluated after evaluating `expr`

```
a = 32;

do{
    b = cool_fn(a);
}while (a > 0);
```

FOR loop

Syntax: `for(e_start; e_end; e_iter) expr;`

- predefined number of cycles
- `e_start` evaluated once at the beginning
- `e_end` evaluated every beginning of iteration
- `e_iter` evaluated every end of iteration

```
for (i = 0; i < 10; i++){
        b = cool_fn(i);
}
```

FOR loop II.

- any of `e_start`, `e_end`, `e_iter` might be ommited

```
// same function as previous loop
i = 0;
for (; i < 10; ){
    b = cool_fn(i);
    i++;
}

// endless loop
for(;;){ ... }
```

Ex. More Loops

```
// empty loop
while (i < 1) ;

// nested loops
for(i = 0; i < 10; i++){
    for(j = 0; j < 10; j++){
        s = compute(i, j);
    }
}
```

Loop Control

- `break` – immediately finishes innermost looping
- `continue` – skips rest of current iteration

```
// skip trendy_fn() if cool_fn() returns value below ze
for (i = 0; i < 10; i++){
    b = cool_fn(i);
    if(b < 0) continue;
    c = trendy_fn(i,b);
}

// stop looping when cool_fn() returns zero
for(;;){
    b = cool_fn(++i);
    if(b = 0) break;
}
```

GOTO command

Syntax: `label: ... goto label;`

- be AWARE!

```
// trivial example -- to be AVOIDED
puentezero:
   ...
goto puentezero;
```

Ex.: Meaningfull use of GOTO

```
// escaping nested loops
for(i = 0; i < 10; i++){
    for(j = 0; j < 10; j++){
        for(k = 0; k < 10; k++){
           if(a[j] == 0) goto error;
           n = b[k] + a[i] / a[j];
        }
    }
}
goto allright;

// handle zero divisor
error:
 ...
allright:
 ...
```

RETURN statement

Syntax: `return expr;` or `return(expr);`

```
return;              // return from void fn
return 1;            // return constant, same as "return(1)"
return (a);          // return value of a variable
return cool_fn(a);   // return a return value of other fn
```

- finishes current function
- define the return value of function
- in `main()` finishes entire program;

Conditional operator

Syntax: `condition ? expr1 : expr2`
corresponds to: `if(condition) expr1; else expr2;`

```
int i, k, j = 2;

i = (j == 2) ? 1 : 3;
k = (i > j) ? i : j;
k = ((i > j) ? do_it() : do_sth_else());

(i == 1) ? i++ ; j++;
```

Comma operator

Syntax: `expr1, expr2`

- first expression is evaluated,
- then second expression is evaluated,
- its value is returned.

```
// equal to i = b;
i = (a, b);

// ... use in for loop
for (i = 0; i < 10; i++, j--)
        ...
```

Type Casting

- implicit vs. explicit
- declare that object of defined type should be treated as object of another type

Implicit:

- `int -> unsigned -> long -> float -> double`
-

```
int a;
unsigned b;
long l;
float f;

l = a + b; // cast to unsigned, then to long
f = l / a; // cast to long, then to float
```

Type Casting II.

```
void print_int(int a);

float f = 3.14195;

// f is implicitly casted to int
print_int(f);

// the same, explicitly
print_int((int) f);
```

Use explicit typecasting for better readability!

The Big Picture

"A program should be light and agile, its subroutines connected like a string of pearls. The spirit and intent of the program should be retained throughout. There should be neither too little or too much, neither needless loops nor useless variables, neither lack of structure nor overwhelming rigidity. "

The Tao Of Programming, 4.2

Modular structure

Function:

- isolated sequence of statements
- reusable block of code
- only one `main()` function, program entry point

Module:

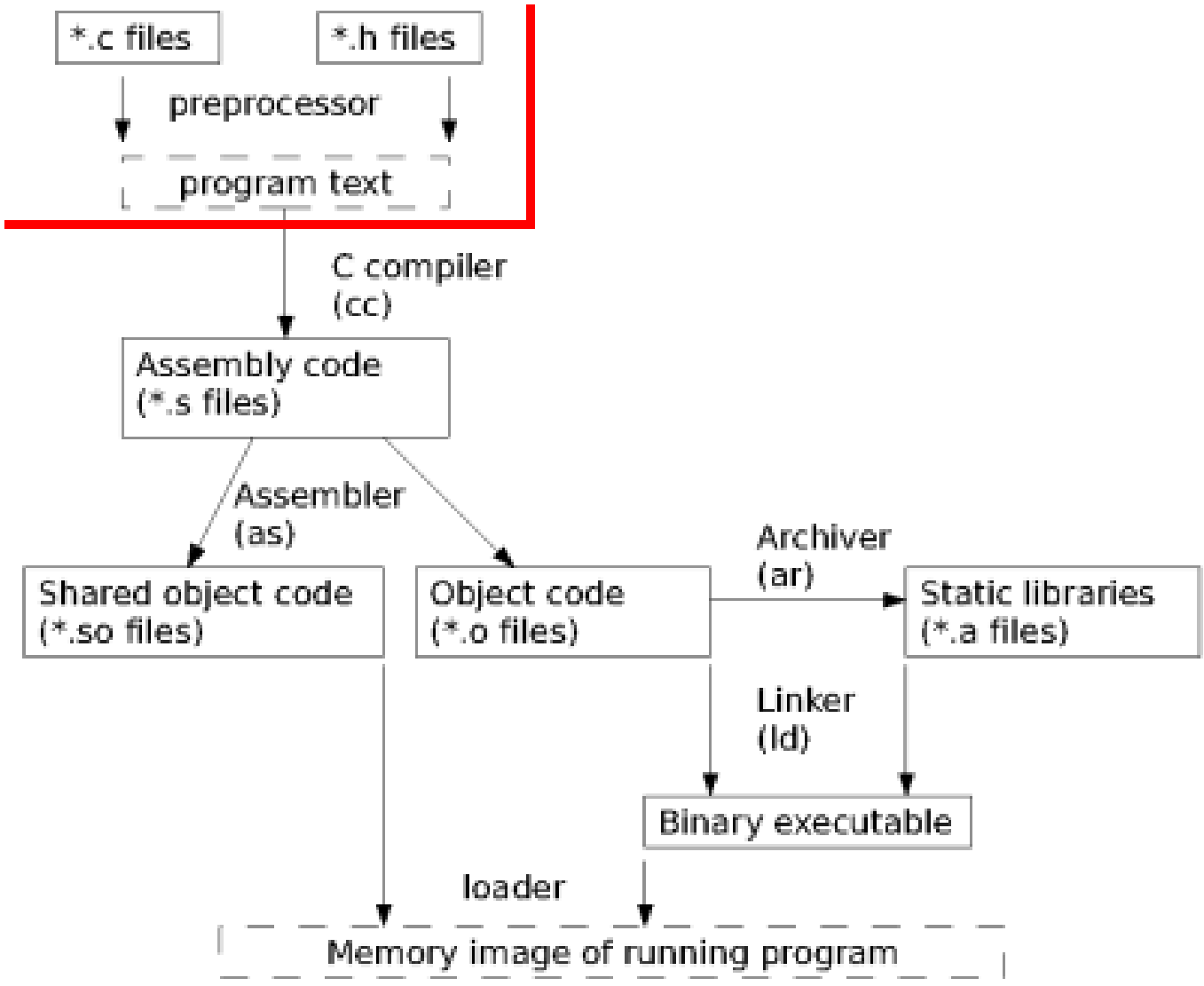
- in C module is equal to *source file* (*.c)
- groups function for specific task
- function and shared variables usualy declared in respective *header files* (*.h)
- multiple modules might be converted into single program image (e.g. executable, hex file) in process of *compilation*
- only method to keep order in large projects

Compilation

Process that converts your program text into a memory image of the program.

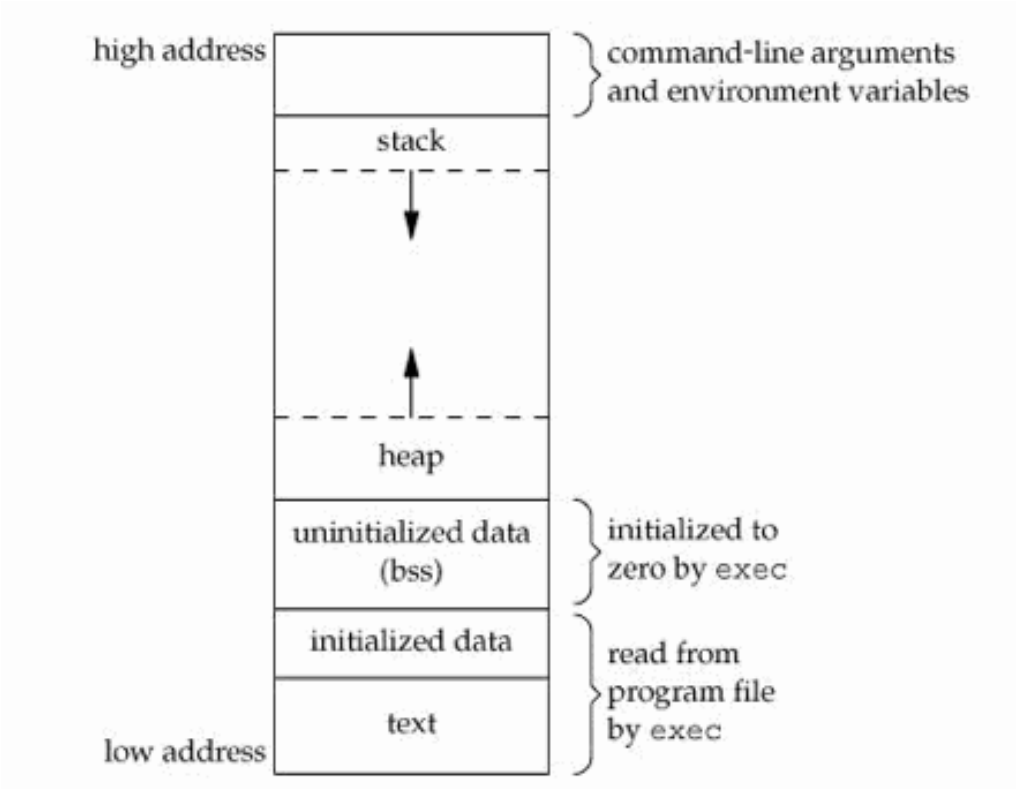
- Preprocessor – prepares source text
  - conditional compilation, macro expansion, file inclusion
- Compiler – transform text into machine-code (*.s)
  - modules compiled separately
  - syntax checking
- Assembler – creates object files (*.o)
- Linker – merges object files into single program image
  - merges assembled modules together with precompiled libraries
  - resolves function and variable names resolution
  - assign final adresses to funcs and variables

Compilation process



Memory Allocation

Every function and variable need to have allocated memory space.



Memory Allocation II.

Static memory allocation:

- memory allocated by linker / program loader
- within the program code (program text)

Dynamic memory allocation:

- stack – for `auto` variables and function passing
- heap – available for programmer (`malloc()` etc.)

Variable Scope

- local var – visible within function / block
- global var – visible outside function (within module), may be exported to other modules
- static local var – local var that retains value between calls
- static global var – visible only in module where is defined

Variable Modifiers – memory classes

auto	default for local variables (S)
static	local variables that retains value between fn calls (T) OR global variable, visible only within parent module
register	variable put in working register for faster access
extern	refers to variable from other module (no memory space allocated)

Variable Modifiers – type classes

const – constant value, cannot be changed
often used to declare that function argument will not be changed:

```
int seek(const char *str,char what);
```

volatile – value changed by external means (e.g. interrupt)
prevents compiler optimizations

C Preprocessor

- process the source text before compilation
- macro processing
- conditional compilation
- discard comments
- does not check syntax!
- lines for preprocessor start with #

Directives

Macro (un)definition:

- `#define name any expansion text`
- `#undef name`

File inclusion:

- `#include <filename>`
- `#include "filename"`

Error messages:

- `#error Error message`

Compilation parameters:

- `#pragma`

Directives II.

Conditional compilation:

- `#if constant_condition`
- `#elif #else #endif`

... if macro `name` is defined:

- `#ifdef name`
- `#elif #else #endif`

... if macro `name` is NOT defined:

- `#ifndef name`
- `#elif #else #endif`

Simple Macros – Symbolic Constants

```
#define  MAX          1000
#define  PI           3.14159
#define  TWO_PI       (2 * PI)
#define  AND          &&
```

- no ; after definition (unless wanted)
- no = between name and expansion
- new constant may use previously defined macro
- constant might appear anywhere in text, except in "quoted string"
- UPPERCASE as a strongly-reccomended convention

Simple Macros II.

```
#define  FILENAME    "letter.txt"
#define  LONGSTRING  This is a long string constant\
                     that won't fit single line.
#define  KHZ         *1024
#undef   FILENAME
#define  FILENAME    "notes.txt"
```

- long constants may use \ as delimiter, followed by newline
- constant is valid from definition (not before)
- until EOF of `#undef`

Macros w/ Parameters

`#define` *macro_name(Arg1, Arg2, ... ArgN) expansion*

```
#define  is_upper(c)  ((c) >= 'A' && (c) <= 'Z')
```

- no space between `macro_name` and opening `(`
- preprocessor does no type checking!
- recursion not possible!

File Inclusion

```
// include system-defined file
#include <stdio.h>

// include user-defined file
#include "my_constants.h"
```

- included file is pasted into calling file
- directories with system/user files are system-dependent

Conditional compilation

```
#if 0
 part of code to be commented out
#endif

#if DEBUGLEVEL < 1

print("There's a big problem, something goes terribly wrong:\n");
print("It's raining, I'm hungry and there's no memory left!\n");
#error Out of memory!

#elif DEBUGLEVEL = 0

print("I'm running out of memory!\n");

#else

print("Everything is OK.\n");

#endif
```

Conditional compilation II.

```
#ifdef _HAVE_DSP_UNIT

#define PRECISION_LIMIT 500

int calc_meaning_of_life(void){
return sqrt(2^6 + ((666 % 11) + 1) * 1e3);
}

#else

#define PRECISION_LIMIT 50

int calc_meaning_of_life(void){
return 42;
}
#endif
```

Preventing repeated inclusion

```
/*
*  my_constants.h -- shared global constants
*/

#ifndef MY_CONSTANTS_H
#define MY_CONSTANTS_H

#define RANGE_MAX 1000
#define RANGE_MIN 10

#endif
```

Separate Compilation

Example: module for advanced math

- module `amath.c` + header file `amath.c`
- module `main.c`

main.c

```
/*
* main.c -- top level module of AMATH project
*/

// system headers
#include <stdio.h>

// user-defined headers
#include "amath.h"

// main ()
void main(void){
  int a = 1; b = 3;
  float res;

  printf("Input vector: %d %d\n",a,b);
  printf("Maximum: %d \n",am_max(a,b));
  printf("Average: %f \n",am_avg(a,b));

  res = am_ratio(a,b);
  if(amath_error == DIV_BY_ZERO){
    printf("Ratio not defined!");
  }else {
    printf("Ratio: %f \n",am_ratio(a,b));
  }

  return 0;
}
```

amath.h

```
/*
* amath.h -- AMATH project, amath module header
*/
//
#define DIV_BY_ZERO -1;

// extern variables
extern int amath_error;

// function prototypes
int am_max(int a, int b);
float am_avg(int a, int b);
float am_ratio(int a, int b);
```

amath.c

```
/*
* amath.c -- AMATH project, amath module
*/

// global variables
int amath_error;

// local prototypes
int am_sum(int a, int b);

// global functions
int am_max(int a, int b){
  return (a > b ? a : b);
}

float am_avg(int a, int b){
  return am_sum(a,b) / 2;
}

float am_ratio(int a, int b){
  if (b == 0){
    amath_error = DIV_BY_ZERO;
    return 0;
  }
  return (a / b);
}

// local fn definition
int am_sum(int a, int b){
  return a + b;
}
```

C Standard Library

- standard functions, contained $\pm$ in every C distribution
- i/o system, string ops, math funs, time functions etc.

reference: http://www.cplusplus.com/reference/clibrary/

Real Power of C

"A C program is like a fast dance on a newly waxed dance floor by people carrying razors."
Waldi Ravens.

Advanced Data Types

- custom datatypes – `typedef` command
- pointer – adressing arbitrary object in memory
- structure – multiple items in single data type
- array – ordered set of elements of same type
- enumerators, unions

Operator SIZEOF

- determines size of an object [Bytes]
- evaluated at compile time

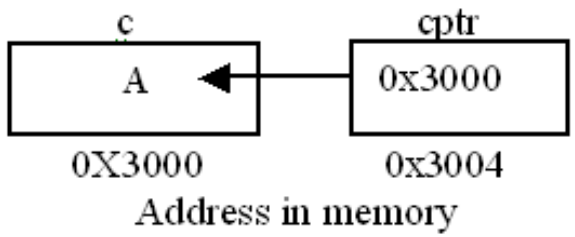
```
a = sizeof(int);

typedef struct { int name;  char *notes; float vek; } p
a = sizeof(person);
```

Pointers

- variable that store address of another variable, structure, function, ...
- typed: "pointer to the type XXX"
- but may be type-casted
- size is machine-dependent

```
char c = 'A';
char *cptr;
cptr=&c;
```



Pointers II.

- reference op `&`: `&var` denotes address of variable `var`
- dereference op `*`: `*ptr` denotes value of variable placed on address `ptr`

```
int i = 1, j;   // integer
int *pi;        // pointer to integer
p_i = &i;       // store address of i to p_i
*p_i = 5;       // same as i = 5;
```

Note: asterisk in pointer definition – do not mix with dereferene.

Pointers and Functions

- both arguments of function or return value can be a pointer
- useful for *call by reference*
- function may modify the argument

```
int *max(int *a, int *b){
   //returns pointer, not the value
   return (*a > *b ? a : b);
}
```

Pointers TO Functions

- address of memory where function is located
- no parameter checking!

```
// functions
int print1(int);
int print2(int);

// declaration
int (*pif)();
int a;

// assignment
pif = print1;
(*pif)(1);
pif(a);

pif = print2;
pif(a);
pif(a,1,2); // extra arguments ommited
pif(); // undefined value in parametes
```

Arrays

- homogenous datatype, set of variables of same kind
- ordered, indexed from zero
- array bounds not checked by compiler
- array elements stored linearly in memory

```
int a[10];          // define array of 10 elements
a[1] = 6;           // assign value to second element
  // (first is a[0])
a[10] = -1          // bad bad bad bad bad
```

Note: array overrun is usual cause of errors ... and freezes ... and resets ... and security compromises ... etc.

Arrays II.

Initialization:

```
int prime[3] = {5, 7, 11};//
int prime[3] = {5, 7}; // prime[2] is initialized to zero
int prime[3] = {5, 7, 11, 13}; // error -- more elements
char letters[5] = {'a','h','o','j', '!'};
```

Note: array is NOT a primary datatype, following assignment is not correct:

```
int happy[3]  =  {7, 13, 25};
int tmp[10];
tmp = happy;
```

Arrays and Pointers

- arrays and pointers are closely related
- for `int arr[10];`, `arr` is constant pointer to beginning of memory segment

```
int i, happy[3]  =  {7, 13, 25};
int *ptr;

ptr = happy;

for (i = 0; i < 3; i++){
cool_fn(ptr[i]);
cool_fn(happy[i]);
}
```

Arrays as Function Arguments

- always passed by reference

```
// equivalent prototypes
int max (int n, const int *arr);
int max (int n, const int arr[]);

// definition
int max(int n, int ...){
    int lmax = 0, i;

    for (i = 0; i < n; i++)
        if(arr[i] > lmax) lmax = arr[i];
    }
    return lmax;
}

// programmer is responsible for boundary checking!!!
int pole[3] = {1, 3, 4};

// following statement lead to undefined result
printf("Max: %d",max(5,pole));
```

Pointer arithmetics

Pointers are de-facto numbers, following operation may have sense:

- comparison (==,!=,<,>): `if(p1 == p2){ ...`
- sum of pointer and integer: `*(p + n)`
- increments (++, --)
- difference (`p1 - p2`)

Comparing Pointers

- comparison of memory addresses
- have sense only for pointers to same memory area – e.g. single array
- only pointers of same type, except NULL

Pointer arithmetics II.

Assume that:

```
int *pc, data[10] = ...
pc = data;
```

Then it holds:

```
*pc = pc[0] = data[0]
*(pc + 1) = data[1]
*(pc + n) = data[n]
```

- *(pc + n) points to n-th element (of type `pc`) after `pc`

Pointer arithmetics – addressing

Assume, that `sizeof(char) == 1`, `sizeof(int) == 2`, `sizeof(float) == 4`

```
char *pc = 10;    int *pi = 10;    float *pf = 10;
```

Then it holds:

```
p_c + 1 == 11
p_i + 1 == 12
p_f + 1 == 14
(char *) p_i + 1 == 11
(char *) p_f + 1 == 11
```

Ex. Pointers, Array, Type-casts

```
#define ARRAY_SIZE 3
int i;
unsigned int arr1[ARRAY_SIZE]  =  {0xfeed, 0xdead, 0xbeef};
unsigned int *ptr_i;
unsigned char *ptr_c;

ptr_i =  arr1; // ... = &(arr[0]) also possible
ptr_i =  (char *)arr1;

// print integer values
for (i = 0; i < ARRAY_SIZE; i++) printf("0x%x ",ptr_i[i]);

// or
for (i = 0; i < ARRAY_SIZE; i++,ptr_i++) printf("0x%x ",*ptr_i);

// print individual characters
for (i = 0; i < ARRAY_SIZE * sizeof(int); i++) printf("0x%x ",ptr_c[i]);
```

Strings

- basically array of `char`
- terminated with null character (`'\0'`)
- literals enclosed in double-quotes: `"string"`;

```
char s1[10] = "Hello!"; //
char s2[] = "Howdy!";   //
char *ptrs;

s1 = "Bye!";                // not possible!
ptrs = s2;                  // as w/ any other array
```

Ex. Strings, Pointers and Functions

```
#include<ctype.h>

void str_to_upper(char *string){
    int i;
    char *ptr = string;

    // loop until '\0' character occurs
    while(*(ptr++)) // or while(*ptr != 0)
      *ptr = toupper (*ptr);
}

char retezec[] = "PokusnyRetezec c. #1";
str_to_upper(retezec);

printf("Converted string  %s\n",retezec);
```

Dynamic Allocation

```
void *malloc (size_t size);
void free(void *ptr);

#include <stdlib.h>

// create array with powers for n = 1..10
unsigned char *p_c;
p_c = (unsigned char *) malloc (sizeof(char) * 10);

for (i = 0; i < 10; i++)  p_c[i] = i + 1 * i + 1;
...
free ((void *) p_c);
```

Dynamic Allocation II.

```
#include <stdlib.h>

// create array of structures
fellow_t *my_friends;
p_c = (fellow_t *) malloc (sizeof(fellow_t) * 10);
```

Custom datatypes

- keyword `typedef`
- syntax: `typedef` *type definition type name*

```
// type definition
typedef unsigned long int ulong;

// variables of new type
ulong temp, temp1;
```

Structures

- heterogenous datatype (first class)
- used to incorporate attributes of single entity (the very fundaments of OOP:)

```
struct struct_name {
   ...items...
} var_name;
```

Declaring Structure

```
// define variables ann, mary and claire
struct {
    int height;
    float weight;
    int age;
} ann, bob, claire;

// declare named structure and than define variables
struct props{
    int height;
    float weight;
    int age;
};
struct props ann, bob, claire;
```

Declaring Structure II.

```
// define new datatype
typedef struct props{    // structure name may be ommited
    int height;
    float weight;
    int age;
    char *name;
} t_fellow;

// define variables
t_fellow ann, bob, claire;

// define pointers
t_fellow *best_friend;

// init
t_fellow peter = {185, 75.8, 22, "Peter"};
t_fellow tania = {.name = "Tania", .height = 168, .weight = 55.2, .age = 20 };
```

Accessing Structure Elements

```
// static
ann.age = 21;
bob.height = 170;

// via pointer
best_friend = &claire;
claire->weight = 55;
(*claire).age = 22;
```

Structures as Function Parameters

```
// call by value
void print_age (t_fellow fellow){
printf("Fellow age is %d\d",fellow.age);
}
print_age(bob);

// call by reference
void print_age (t_fellow *fellow){
printf("Fellow age is %d\d",fellow->age);

}
print_age(&claire);
```

Static array of structures

```
// array of structures
t_fellow foreigners[15];

// that small girl from Ukraine
foreigners[12].height = 145;

t_fellow *best_friend;
best_friend = &foreigners[10];
best_friend->age = 14;
```

Dynamic array of structures

```
#define MAX_FRIENDS 13

// array of pointers
t_fellow *friends[MAX_FRIENDS];
int i;

for (i = 0; i < MAX_FRIENDS; i++){
friends[i] = (t_fellow *)malloc (sizeof(t_fellow));
}
```

Bit Fields

```
// time structure
typedef struct {
    unsigned int hour: 5;    // 0-32
    unsigned int minutes: 6; // 0-64
    unsigned int seconds: 5; // 0-32
} mytime;

sizeof(mytime) = 2;

typedef struct {
    unsigned int flag: 1; /* on-off flag*/
    unsigned int num: 4;
    : 3; /* 3-bits padding */
} mydate;
```

Enumerated Type

- "integer with nice names"
- names are assigned integer values, explicitly or
- automatic numbering starts from 0 and increments

```
// enum definition
enum traflites { red, green, blue};

// variable t1
enum traflites t1;

// assignment
t1 = red;
```

Enumerated Type II.

Enums are only weakly typed:

```
enum direction {North, South, West, East};
enum color {red, green, blue};

int paint(enum color c);

enum direction d = South;
paint(d);          // undetected error
```

Enumerated Type III.

Enums often used for bitfields

```
typedef enum {
perm_none = 0,
perm_read = 1,
perm_write = 2,
perm_exec = 4
} t_perm;

t_perm p1;
p1 = perm_read;
p1 = perm_none;
p1 = perm_read | perm_write ;

// but:
int demo = perm_read | perm_exec;
```

Unions

- similar to structures, but stores ONE of elements at time
- `sizeof(my_union)` is equal to size of largest member

```
union time {
    long simpleDate;
    double perciseDate;
} mytime ;
```

Ex. Structures in Unions

```
union cool_byte{
unsigned char c;
struct{
unsigned int low: 4;
unsigned int hi :4;
};
struct{
unsigned int b1 : 1;
unsigned int b2 : 1;
unsigned int : 6;
};
};

// b1 = 0x55
union cool\_byte b1 = { .c = 0x55 } ;
b1.low = 0x3;  // b1 = 0x53
b1.hi = 0x7;   // b1 = 0x73
b1.b1 = 0;     // b1 = 0x72
return 0;
}
```

2D Arrays

Static definition:

```
int arrA[3][4];
```

- allocated in continuous memory block
- equivalent to `*(*(arrA + i) + j)`
- fixed number of columns and row

Dynamic – pointer to array (rarely used):

```
int (*arrB)[4]; // pointer to array
```

- dynamically alocated in single block

```
arrB[i] = (int *[4])malloc ( 4 * 2 * sizeof(int));
```

- fixed number of columns, arbitrary number of rows

2D Arrays II.

Dynamic – array of pointers:

```
int *arrC[3]; // pointer array
```

- dynamically alocated per line

```
for (i = 0; i < 3; i++) arrC[i] = (int *)malloc ( 4 * sizeof(int));
```

- fixed number of rows of varying size

Dynamic – pointer to pointer:

```
int **arrD; // pointer to poiner
```

- dynamically alocated array for pointer to lines:

```
arrD = (int **)malloc ( 4 * sizeof(int));
```

- then allocation per line:

```
arrD[i] = (int *) malloc ( 3 * sizeof(int));
```

Best Practice

Thus spake the master programmer: "A well-written program is its own heaven; a poorly-written program is its own hell."
The Tao Of Programming, 4.1

Blocks & Indentation

Functions:

```
// ansi C
int max(int a, int b)
{
    statement;
    return;
}

// K&R C
int max(int a, int b){
    statement;
    return;
}
```

Blocks & Indentation II

Loops:

```
while(true){
    statements;
}

if(a < 1){
    statement;
}else{
    statement;
}
```

Spacing

- after comma/semicolon: `int a, b; float c;`
- around operator: `a = b / (b + 1);`
- except `++`,`--`,: `i++; --a;`
- in logical expression: `for(i = 0; i < 3; i++)`

Source file structure

- opening comment: filename, author, version, brief description
- #includes: only `.h` files, no `.c`; first system headers, then user-defined
- global variables (outside functions) to be exported
- local `#defines`
- local `typedefs`
- global variables (outside functions) for local use
- function prototypes of local functions
- `main()` function
- global functions definition
- local functions definition

Header file structure

- opening comment: filename, author, version, brief description
- global `#defines`: constants, macro functions
- global `typedefs`: structures etc.
- global variables declaration: from respective `.c` modules
- global functions prototypes: from respective `.c` modules

No definitions! Avoid other `.h` inclusion if possible.