Programming in C

3.9.2014

Tiina Niklander

Course structure

- Based on C programming course in Aalto
- On-line material from Aalto: http://src.aalto.fi/c
- Exercise submission on our own server: http://tmc.mooc.fi/hy
- No lectures, but paja guidance

- TMC deadline every week: Wednesday at 14.00

- Use a course book in addition to the on-line material

Course material

- Exercise submissions to our TMC: tmc.mooc.fi/hy
- On-line material from: src.aalto.fi./c

- Any good C course book is useful addition:
  - Kernighan & Richie: C programming language
  - Müldner : C for Java programmer
  - King: C Programming, A Modern Approach

C language

C assumes that programmer is intelligent enough to use all of its constructs wisely, and so few things are forbidden.

- Closely tight with UNIX – same original developers
- ’Low-level’ language
  - explicit memory allocation and deallocation!
  - allows operations on the memory addresses and bit-level
  - allows dynamic type changes of variables
  - important concept: POINTER (= memory address)

C can be a very useful and elegant tool. People often dismiss C, claiming that it is responsible for a ”bad coding style”. The bad coding style is **not the fault of the language**, **but** is controlled (and so caused) by **the programmer**.

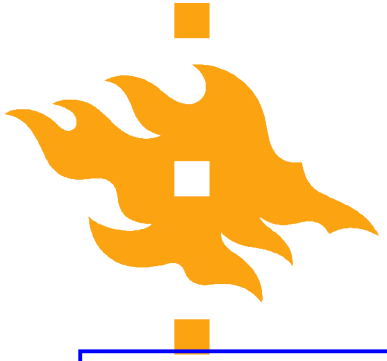
Compiling: gcc -ansi -pedantic -Wall

- Options –Wall and –pedantic show all possible warnings (used by TMC)
- Option –ansi makes sure that the compiler follows standard (some differences in gcc exist)

```
gcc -ansi -pedantic -Wall -o helloworld helloworld.c
/home/fs/niklande/C-luennot/esimerkit/helloworld.c: In function `main':
/home/fs/niklande/C-luennot/esimerkit/helloworld.c:3: warning:
implicit declaration of function `printf'
```

```
int main (void)
{
  printf(”Hello world \n”);
  return 0;
}
```

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



Multiple modules
(separate source files with headers)

```
/* main.c */
#include <stdio.h>
#include ”eka.h”
#include ”toka.h”
int main (void)
{
  eka(); toka ();
  return 0;
}
```

```
/* eka.c */
#include <stdio.h>
#include ”eka.h”
void eka (void)
{
  puts(” eka ”);
}
```

```
/* eka.h */

void eka (void);
```

```
/* toka.c */
#include <stdio.h>
#include ”toka.h”
void toka (void)
{
  puts(” toka ”);
}
```

```
/* toka.h */

void toka (void);
```

Could write everytime:

```
gcc –c main.c
gcc –c eka.c
gcc –c toka.c
gcc –o ohjelma main.o eka.o toka.o
```

→ Instead, use **makefile**

makefile

```
gcc –c main.c
gcc –c eka.c
gcc –c toka.c
gcc –o ohjelma main.o eka.o toka.o
```



```
make
```

Write makefile just once

Use several times with command make

```
# makefile
CC = gcc –ansi –pedantic –Wall
ohjelma: main.o eka.o toka.o
        $(CC) –o ohjelma main.o eka.o toka.o
eka.o: eka.c eka.h
        $(CC) –c eka.c
toka.o: toka.c toka.h
        $(CC) –c toka.c
main.o: main.c eka.h toka.h
        $(CC) –c main.c
```

Example program

- What does this program do?

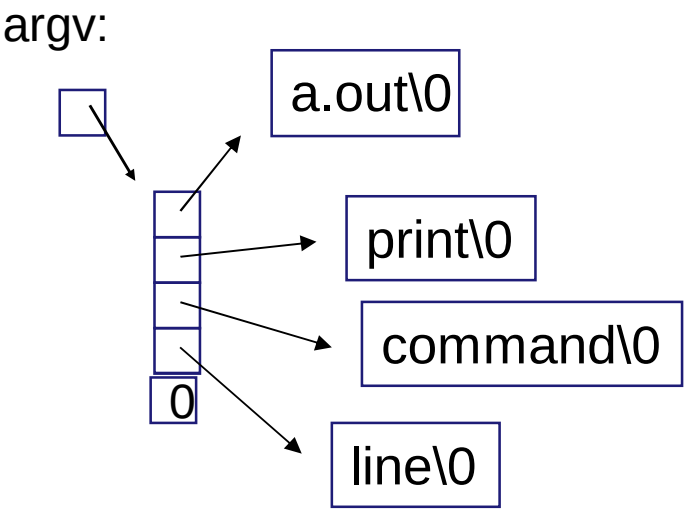
```
#include <stdio.h>
/*   Explaining comment removed */
int main(int argc, char** argv)
{
  int i;

  for (i=0; i < argc; i++)
     printf("%s%s", argv[i],
         (i <argc-1) ? " " : "");
   printf("\n");
return 0;
}
```

Example program: a.out print command line

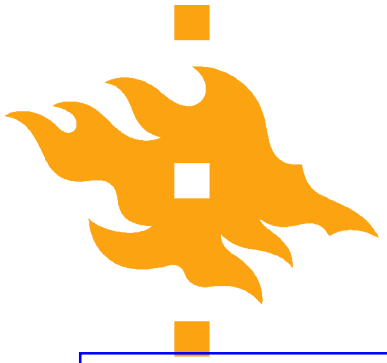
```
#include <stdio.h>
/* Echo the command line with params */
int main(int argc, char** argv)
{
  int i;

  for (i=0; i < argc; i++)
     printf("%s%s, argv[i],
          (i <argc-1) ? " " : "");
   printf("\n");
return 0;
}
```



NOTICE:
- Parameters
- Array indexing

Modification:
How would you avoid printing the name of the program?



Command line parameters: typical usage

```
static int process_parameters(int argc, char *argv[]) {
  int i, string_found =0;
  for(i=1; i<argc; i++){ /* process command switches. Note side effects! */
    if (argv[i][0] == '-') {  switch (argv[i][1]) {
      case 'c': count_lines = TRUE; break;
      case 'i': ignore_case = TRUE; break;
      case 'b': line_beginning = TRUE; break;
      default: printf("Unknown option %s - ignored \n", argv[i]);  break;
    }} else {
      if (!string_found) {
        copy(string, argv[i], STRINGSIZE); string_found =1;
      } else {
        printf("Only one search string! \n"); return FALSE;   } }}
  if (!string_found) {
    printf("The search string must be given!\n"); return FALSE; }
  return TRUE; }
```

options: -c, -i, ja -b

Search string cannot start with character -

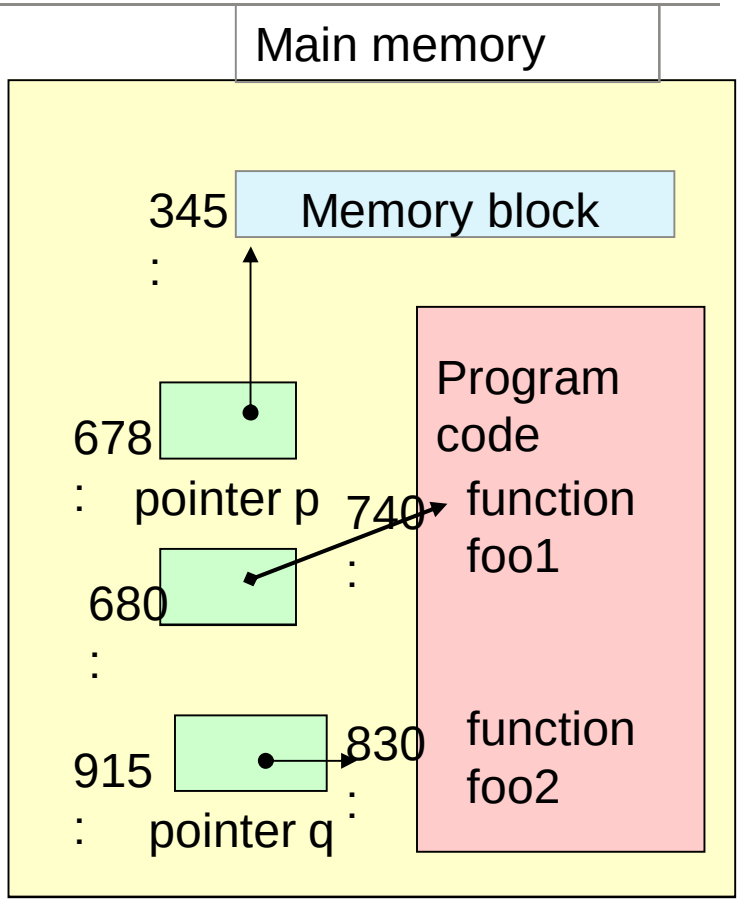
Functions always return value!

Briefly about pointers

```
char *p;  /* pointer to a character or string */
int *q;   /* pointer to interger (or structure) */
/*Memory allocated only for the pointer! */
```

```
char *p = ”This string is allocated”;
int numbers[] = {1, 2, 3, 4, 5};
double table[100];
```

Allocate memory for the structure and set the pointer to the structure.



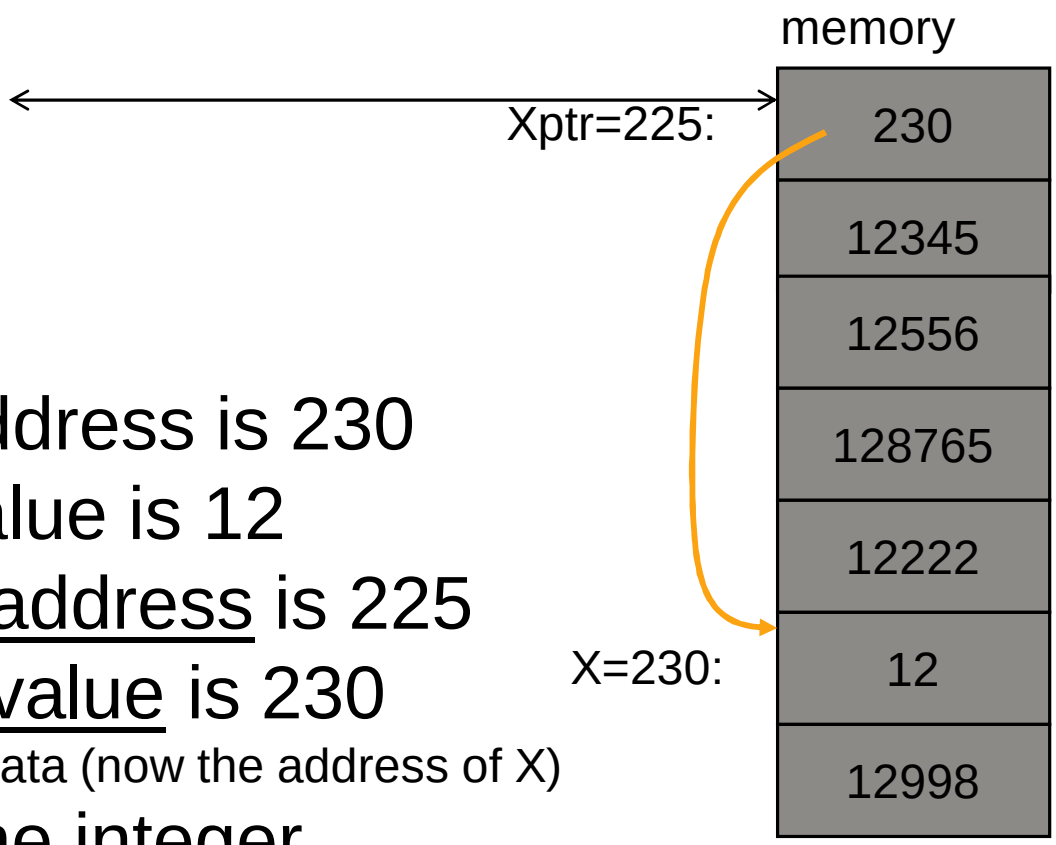
Addresses and values

```
Xptr  DC   0
X     DC   12
      LOAD  R1, =X
      STORE R1, Xptr
      LOAD  R2, X
      LOAD  R3, @Xptr
```

; R1 ← 230

; R2 ← 12

; R3 ← 12



TITO

- Variable X's address is 230
- Variable X's value is 12
- Pointer Xptr's address is 225
- Pointer Xptr's value is 230
  - Address of some data (now the address of X)
- The value of the integer that Xptr points to is 12

In C language as: Y = *Xptr;
/* value of the location Xptr points to */

Pointers (and arrays)

- Pointers are variables whose values are addresses.
- Array name can be thought of as a constant pointer.
- Array is just a sequene of values with a joint name.
  int a[15] is sequence of 15 integers.
- Array name is a pointer whose value is the address of the first element in the sequence.
  pa = &a[0]
  pa = a
- pointer arithmetic allows operations on array elements
  *(pa +3) is the same as a[3]
  pa+3 is the same as &a[3]

Pointer arithmetics and operations

Remember:
NULL

p = &c address of c
c = *p value of the address pointed by p
c = **r -"- (two 'jumps')
p = q allowed when p and q of same type

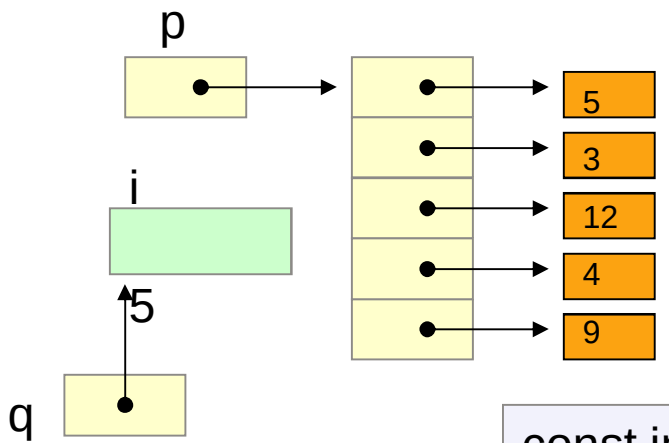
p+i, p-i p is array, i has to be interger with suitable value
p-q, p an q pointers of the same array and q<p
p < q, p == q

*ip++ increments the address by 'one'

(*ip)++ increments the value in the address by one

Pointer arithmetics

```
int  **p;
    int    *q,r;
    int     i;
   i= **p
```




```
q = &i; /* q's new value is i's address
 i = *q+1;     i = ++*q;   /* i=6*/
 i = *q++;   /* ???? */
 r = q; *r = 3; /* i=3 */
```

```
const int  *p;
int  const *p;
const  int const  *p;
```

```
char msg [] = "It is time";    msg: It is time\0
char  *pv ="It is time";       pv -> It is time\0
```

```
void *p;   i= *(int*) p;
```

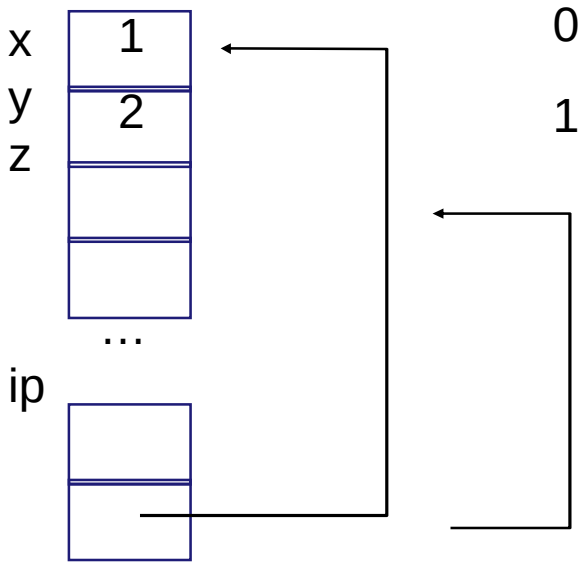
Example code

```
int main(int argc, char**
argv)
{
 int x=1, y=2, z[10];
 int *ip;
int *p, q;
int *r, *s;

ip = &x;
y = *ip;  /* y = x =1 */
*ip = 0;   /* x = 0 */
ip = &z[0];
}

double atof(char * string);
```

p is a pointer variable and
q is integer variable



Pointers as arguments for functions are very common. (Always used with arrays and needed for call by reference)

Memory allocation

- Explicit memory allocations!
- **malloc** – static data structures
- calloc – dynamic array
- realloc – change the size of already allocated object
- **free** – deallocate the memory

```
/* ALWAYS CHECK THE RETURN VALUE!!! */
if (k=malloc(sizeof(double)))
     error; /* allocation failed, do something else or terminate program */

/* memory allocation succeeded and k is the pointer to the new structure */
```

Functions: Call by value, call by reference

C uses always call by value => function cannot change the value it receives as argument.

Call by reference done with pointers

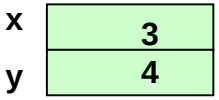
```
void swap(int x, int y) {
  int apu;
  apu=x;
  x=y;
  y= apu;
}
```

copies

Addresses of x and y

```
void swap(int *x, int *y) {
    int apu;
    apu=*x;
    *x=*y;
    *y= apu;
}
```

Call: swap (&x, &y);




```
double product (const double block, int size);
```

Make sure that function does not change the variable (ANSI standard!)

Example code: copy a string - Passing array to a function

```
#include <stdio.h>

void kopioi( char *s, char *t)
{
   int i =0;
  while (  (s[i] = t[i]) != '\0' )
    i++;
}

int main (void)
{
  char here [] ="This string is copied.",
        there[50];
  kopioi ( here, there);    printf("%s\n", there);
  kopioi ( there, here);    printf("%s\n", there);
  return 0;
}
```

Strings (character arrays) as arguments. C is always passing only the address of the first element of any array.

Processing one character of each array

Example code: copy a string – Now with pointers

Version 1:

```
void kopioi( char *s, char *t)
{
  while (  (*s = *t) != '\0' )
    s++; t++;
}
```

Version 2:

```
void kopioi( char *s, char *t)
{
  while (  (*s++ = *t++) != '\0' )
    ;
}
```

NOTE: The Fuction prototype is identical with the previous slide

Version 3:

```
void kopioi( char *s, char *t)
{
  while  ( *s++  =  *t++ )   ;
}
```

Minimalistic!

Structures (or aggregates or records)

Structures are collections of related variable (of different types) under one name.

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

```
typedef struct LogItemType {
        TRID trid;
        int oper;
        int len;
        char data[256];
} LogItemType;
```

Variables are defined using the struct name:

```
struct point p1, p2, p3;
```

A member (or field) of a structure is referenced with dot operator '.'

```
p1.x = 2 * p2.y;
```

Pointers to structs

- Struct pointers are like any other pointer

```
struct point *pp;
…
pp = &p4;
printf(”orig (%d, %d)\n”, (*pp).x, (*pp).y);
…
```

- Pointers are so common that Osoittimet ovat niin yleisiä, että tuo muoto (*pointer).member is shortened by arrow operator pointer->member.

```
printf(”orig (%d, %d)\n”, pp->x, pp->y);
```

- Use parethesis, if uncertain about operation order.

sizeof and union

- sizeof-function returns the numerical size of allocation

```
sizeof(struct point);
sizeof(p1);
sizeof(&p1);
sizeof(double);
```

- Members of a union share the same memory area (and are thus alternatives)

```
union u_tag {
  int ival;
  float fval;
  char * sval; };
```

More about pointers and some good practices

- Generic pointer (void *p) can be used with type cast to handle a variable of that type.

```
*(double *)p
```

- Memory allocation for n integers

```
int *p;
if ((p=malloc(n*sizeof(int))) == NULL)
   error;
```

- Memory deallocation: remember to free(p); p=NULL;
- i's element of array

```
p[i]      (preferred over *(p+i) )
```

- Handling an array p

```
for (pi = p;  pi < p+SIZE; pi++)
   remember to use pointer pi in the loop
```

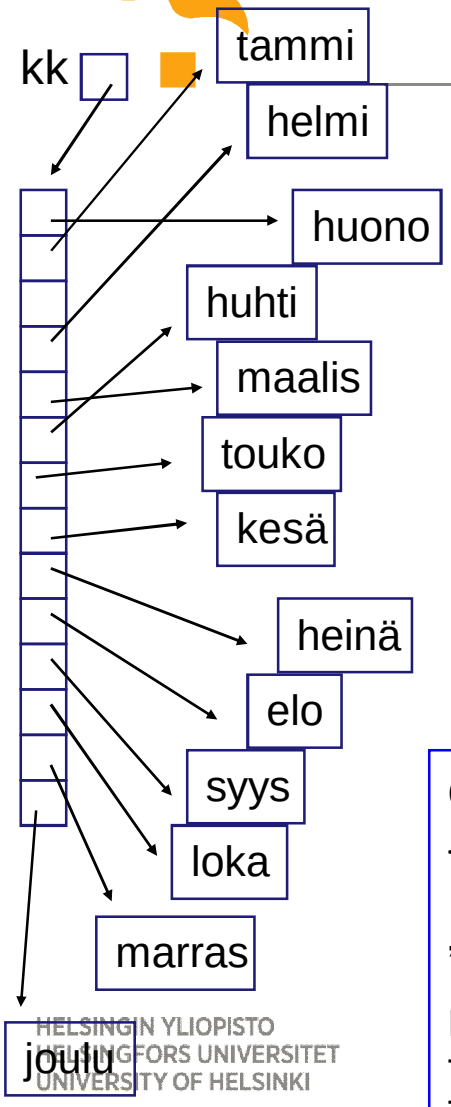
Still more

- Call by reference
  1. Prototype's argument – a pointer
     void f(int *pp)
  2. In the function use the pointed value.
     *pp
  3. In the function call: address of the variable
     f(&mtja);
  4. In the function call: pointer
     f(osmtja);
- Array of struct

  for (p = block; p < block + n*elSize; p+= elSize)
- i. element of struct array

  p = block + i*elSize

Pointer arrays



Pointer has a value and it can be also pointed to by pointer to a pointer:

```
int **a;
```

Similarly we can have array of pointers:

```
int *t[10];
```

Remember the command line arguments.

```
char *kk_nimi(int k)
{
  static char *kk[] = {”huono”, ”tammi”, ”helmi”, maalis”, ”huhti”,
”touko”, ”kesä”, ”heinä”, ”elo”, ”syys”, ”loka”, ”marras”, ”joulu”};
return ( (k<1 || k > 12)  ? kk[0] : kk[k] );
}
```

Function pointers

- Functions also have an address and we can use a that address as a value of a function pointer.

```
int (*lfptr) (char[], int);
lfptr = getline;  /* when int getline(char s[], int len); */
```

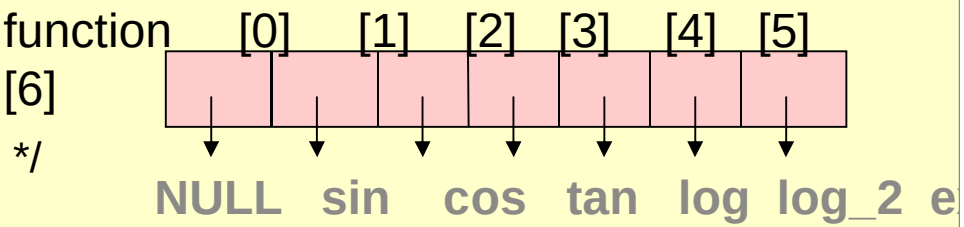
- Function pointers can be
  - passed to other functions, returned from functions
  - stored in arrays,
  - assigned to other function pointers
- stdlib.h has function qsort, whose one argument is the sorting function

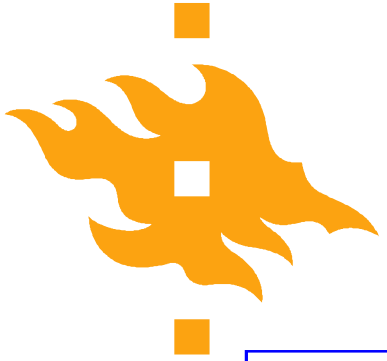
Function array from include/linux/quota.h

```
 /* Operations which must be implemented by each quota format */
struct quota_format_ops {
        int (*check_quota_file)(struct super_block *sb, int type);
                /* Detect whether file is in our format */
        int (*read_file_info)(struct super_block *sb, int type);
                /* Read main info about file - called on quotaon() */
        int (*write_file_info)(struct super_block *sb, int type);
                /* Write main info about file */
        int (*free_file_info)(struct super_block *sb, int type);
                /* Called on quotaoff() */
        int (*read_dqblk)(struct dquot *dquot);
                /* Read structure for one user */
        int (*commit_dqblk)(struct dquot *dquot);
                /* Write structure for one user */
        int (*release_dqblk)(struct dquot *dquot);
                /* Called when last reference to dquot is being dropped */
};
```

```
void main (void) {
int choice;  double x, fx;
funcptr fp;
……..
funcprt function[7] = {NULL, sin, cos, tan, log , log_2, exp}; /*määriteltyjä funktioita*/
/*  funktiomenun tulostus: käyttäjää valitsee   haluamansa vaihtoehdon  */
 ….
scanf (”%i”, &choice);
/* lisäksi tarkistetaan, että valinta on sallittu arvo */
…
if (choice ==0) break;
printf(”Enter x: ”); scanf/(”%lg”, &x);
fp = function[choice];
fx = fp(x);
printf(”\n  (%g) = %g\n”, x, fx);
}
}
```

```
typedef  double (*funcptr) (double );
```





Bitwise operations:

& bitwise and
| bitwise or
^ bitwise xor (exclusive or)
<< left shift
>> right shift
~ one's complement

```
#include <stdio.h>
/* Bittipeliä*/
int main(void)
{
  enum {LL = 011 };
  int i, j;

  i = 0;
  j = i | LL;
  printf("i: %d, LL (okt):%o, i|LL: %d, oktaalina %o\n",
              i, LL, j, j);
  printf("1 & 6: %d, 1 && 6: %d\n",
              1 & 6, 1 && 6);
  printf("1<<3: %d, 8>>3: %d\n",
              1<<3, 8>>3);
return 0;
}
```

NOTE: && logical AND

Prints:
i: 0, LL (okt):11, i|LL: 9, oktaalina 11
1 & 6: 0, 1 && 6: 1
1<<3: 8, 8>>3: 1

Example: function pointer as argument

Java: ”overriding”

Function that can change the sort algorithms during the execution based on number of elements

```
int (*fp) (void);
```

Function pointer

```
int *fp()
```

Function returns pointer to int!

```
int fname(); /* function must have same prototype */
fp = fname;  /*  fp() means now same function as fname()
```

```
void qsort(*line[], int left, int right, int (*comp)(void *, void*));
```

Function search

```
/* Search a block of double values */
int search( const double *block , size_t size,
           double value) {
  double *p;

  if(block == NULL)
    return 0;

  for(p = block; p < block+size; p++)
    if(*p == value)
      return 1;

  return 0;
}
```

Pointer as call by value

Go through the structure

Generic function search

C has no polymorfism, but we can emulate it with generic pointers (of type void*).

Function prototype can have all arguments and return value of generic (undefined) type void

```
int searchGen(const void *block,
              size_t size, void *value);
  /* Generic type is not enough */
  /* Must have more information and arguments */

int searchGen(const void *block,
  size_t size, void *value, size_t elSize
  int (*compare)(const void *, const void *));
```

Prototype:

Number of elements

structure

Size of element

Comparison function

Call back function

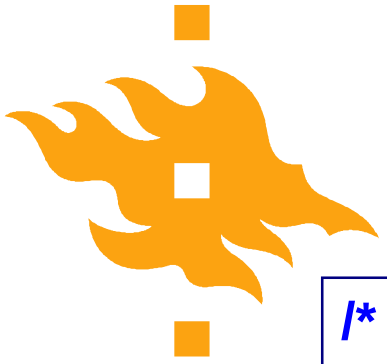
Calling routine must define a **Call back** function

Using typed arguments in the call back function prototype

```
int comp(const double *x, const double *y) {
   return *x == *y;
}
```

With undefined arguments the prototype must also use undefined arguments

```
int comp(const void *x, const void *y) {
   return *(double*)x == *(double*)y;
}
```



Generic search - calling routine

```
/* Application of a generic search */
#define SIZE 10
double *b;    double v = 123.6;     int i;
int main (void) {
 if(MALLOC(b, double, SIZE))
   exit(EXIT_FAILURE);
 for(i = 0; i < SIZE; i++) /* initialize */
   if(scanf("%lf", &b[i]) != 1) {
        free(b);
        exit(EXIT_FAILURE);
   }
 printf("%f was %s one of the values\n",
   v, searchGen(b, SIZE, &v, sizeof(double), comp)
             == 1 ? "" : "not");
return 0;  /* tai exit(EXIT_SUCCESS); */
}
```

Generic search function

```
int searchGen(const void *block,
   size_t size, void *value, size_t elSize,
   int (*compare)(const void *, const void *)) {
   void *p;
   if(block == NULL)
      return 0;
   for(p = (void*)block; p< block+size*elsize;
            p = p+elsize)
      if(compare(p, value))
         return 1;
   return 0;
}
```

NOTE: Pointer operations must use the size of the element!

Modular programming in C

C does not support objects or modular programing, but it has features that can be used to mimic them

- Functions and fuction prototypes
- Header files

Using these features you can separate the interface and implementation

'hidden' implementation

public interface

Structuring a small c program: order of elements in the file

#include directive to use (standard) libraries

Defining constants ad types

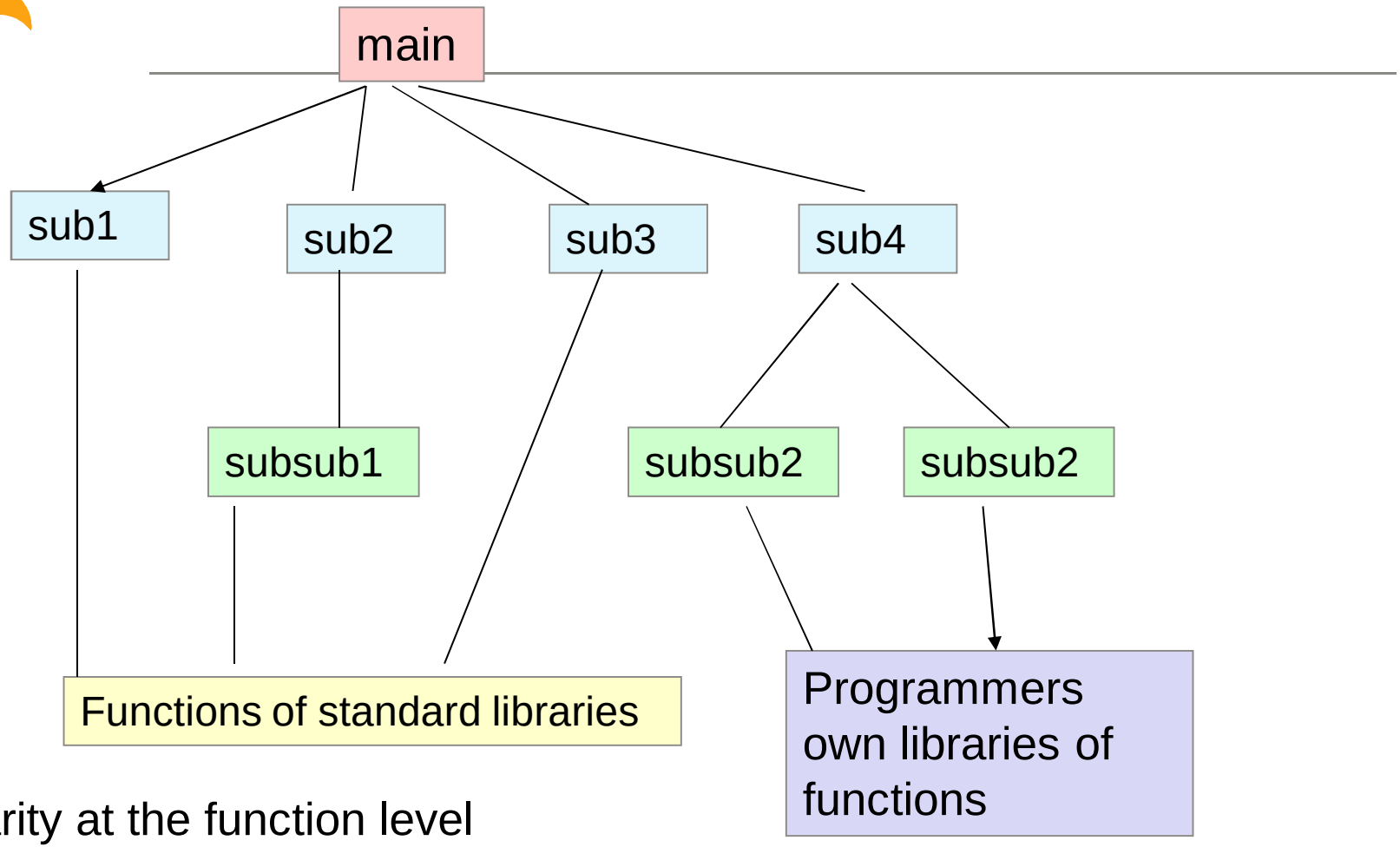
- Easy to locate, modify and control

Function prototypes (before the functions in used)

Function main (every program has one)

Function definitions (=code of the function)

What if a larger project with several programmes? - Split to multiple files and libraries



Modularity at the function level