Programming in C

Prof. Gustavo Alonso
Computer Science Department
ETH Zürich
alonso@inf.ethz.ch
http://www.inf.ethz.ch/department/IS/iks/

Programming in C

- A brief history of C
- C as a programming language
- C Programming
  - main function
  - constants, variables, data types
  - operators, control structures
  - functions
  - data structures
  - pointer arithmetic
  - structures
  - dynamic memory allocation

A brief history of C

- Programming languages are used to specify, design, and build software systems.
- Programming languages evolve with the systems they are used to construct. C is a good example of how this process takes place.
- UNIX was developed at around 1969. It first version was programmed in assembler and run on a DEC PDP-7.
- The second version was ported to a PDP-11 in 1971 and it was a great success:
  - 16 KB for the system
  - 8 KB for user programs
  - disk of 521 KB
  - limit of 64 KB per file
- While writing a FORTRAN compiler for UNIX, a new programming language was developed: B
- B was interpreted (like Java) and, therefore, slow. To solve the performance problems of B, a new language was created: C
  - allowed generation of machine code (compilation)
  - declaration of data types
  - definition of data structures
- In 1973 UNIX was rewritten in C something that was never done before
  - C is much easier to handle than assembler but
  - first C version of UNIX was 20 to 40 % larger and slower than assembler version

C as a programming language

- C has been standardized (ANSI C) and spawned new languages (C++, Stroustrup, 1986) that improve C
- The basic characteristics of C are:
  - small in size
  - loose typing (lots of freedom, error prone)
  - structured (extensive use of functions)
  - designed for systems programming (i.e., low level programming of the type needed to implement an operating system)
  - C is higher level than assembler but still close to the hardware and allows direct manipulation of many system aspects: pointers, memory allocation, bitwise manipulation ...
- As we will see when we study assembler, C is not very far from the assembler language but it provides higher level language constructs (functions, data structures) that facilitate programming without loosing too much performance
- Being a low level language, C gives a lot of freedom to the programmer:
  - it has the advantage that good programmers can implement very efficient programs in a compact manner
  - it has the disadvantage that most of us are not good programmers and the freedom C grants is usually translated in error prone, messy code

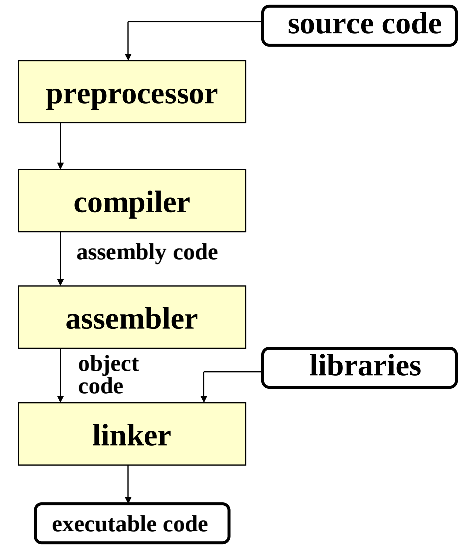
This is C

Winner of the international Obfuscated C Code Contest
http://reality.sgi.com/csp/iocc

```
#include <stdio.h>

  main(t,_,a)
  char *a;
  {return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
  main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a ):3,main ( -94, -27+t, a
  )&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(_,
  t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+\
  ,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/\
  +k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# ){n\
  l]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
  n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
  ;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
  #'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
  :t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
  +1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
  i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```


The C compilation model

- The Preprocessor accepts source code as input and
  - removes comments
  - extends the code according to the preprocessor directives included in the source code (lines starting with #)
- The Compiler takes the output of the preprocessor and produces assembly code
- The Assembler takes the assembly code and produces machine code (or object code)
- The Linker takes the object code, joins it with other pieces of object code and libraries and produces code that can be executed



Structure of a C program

- A C program contains the following elements:
  - Preprocessor Commands
  - Type definitions
  - Function prototypes -- declarations of function types and arguments.
  - Variables
  - Functions
- All programs must contain a single *main()* function. All function, including *main*, have the following format:

```
type function_name (parameters) {
     local variables
     C Statements
}
```

```
#include <stdio.h>
#define TIMES 10 /* upper bound */

double myfunction(float);
            /* Function prototype - Declaration */

main() {
  double wert;
  double pi = 3.14;

  printf("Multiply by 10\n");

  wert = myfunction(pi);

  printf("%d * %f  = %f\n",
                TIMES, pi, wert);
}

double myfunction(double zahl){
   int i;
   double count = 0;

   count = TIMES * zahl;

   return count;
}
```


Data types

- C has the following basic data types

C Type	Size (in bytes)	Lower bound	Upper bound	Use
char	1	-	-	characters
unsigned char	1	0	255	small numbers
short int	2	-32768	+32767	integers
unsigned short int	2	0	65536	positive int
(long) int	4	-2^{31}	$+2^{31}-1$	large int
float	4	$-3.2\cdot10^{\pm38}$	$+3.2\cdot10^{\pm38}$	real numbers
double	8	$-1.7\cdot10^{\pm308}$	$+1.7\cdot10^{\pm308}$	large reals
void	0	-	-	no return value

- The sizes of the data types are not standardized (depend on the implementation)
- The type *void* is used to indicate functions that return no value or null pointers
- With #define, one can introduce symbolic constants

```
#define LIMIT 100
```

Variables and constants

CONSTANTS

- A constant specifies a value that cannot be modified by the program
- Special constants for use with strings:
  - \n new line
  - \t tabulator
  - \r carriage return
  - \b backspace
  - \" escape double quote
  - \0 end string
- Symbols defined through the preprocessor:

```
#define ESC '\033' /* ASCII
                      escape */
```

VARIABLES

- A variable specifies an area of memory that contains a value of a given type that can be modified by the program

```
short x;
long y;
unsigned a,b;
long double lb;
unsigned short z;
```

- sizeof() is a function that returns the size of a given variable in bytes (the size depends on the type of the variable
- Variables should be initialized before they are used (e.g., in the declaration) otherwise the variables contain a random value

Type conversions (casts)

- In C, the type of a value can change during the run time of a program, this is known as type conversion or type cast
- The change can be explicit (the programmer does it) ...

```
var_new_type = (new_type) var_old_type

int a;
float x = (float) a;
```

- or implicit (the compiler takes care of it in order to perform operations among variables of different types):
  - char and short can be converted to int
  - float can be converted to double
  - in an expression, if an argument is double, all arguments are cast to double
  - in an expression, if an argument is long, all arguments are cast to long
  - in an expression, if an argument is unsigned, all arguments are cast to unsigned

```
int a = 3;
float b = 5.0;
float c = a + b;      /* a is transformed into double */
```


Scope

- The scope determines where within a program a particular entity is known and can be accessed
- The scope of a variable is the part of a program where the variable can be manipulated. The scope can be global or local
- Global variables are typically declared before the main function. They can be accessed from anywhere in the program. Try to avoid global variables (a matter of programming style)
- Local variables are valid and visible within a particular block (a block is a set of C statements enclosed within brackets { ...}). Once the control flow is outside the block, the variable no longer exists
- Local variables are created (space in memory is allocated for them) when the control flow in the program reaches the block where they are declared (e.g., a function) and are destroyed (deallocated from memory) when the control flow leaves the block
- The creation and destruction of variables can be controlled:
  - extern: the variable is defined in a different module
  - static: for local variables: makes them last the entire program, for global variables: restricts the scope to the current module
  - register: try to use a CPU register for the variable
  - auto: default for local variables

Scope (example 1)

```
int global_variable;

int main () {
        int local_variable;

        global_variable = 1;
        local_variable = 2;

                {
                int very_local_variable;
                very_local_variable = 3;
                }

}
```

Expressions and priority

- +, -, *, /, % are the basic arithmetic operators
- Addition

```
x = 3 + 4;
```

- Subtraction

```
x = 10 - 3;
```

- Multiplikation

```
x = 3 * 4;
```

- Division

```
x = 73 / 8;
/* x=9, if int x */
x = 73.0 / 8.0;
/*x=9.125,if float x */
```

- Modulo

```
x = 73 % 8;
/* x=1, the reminder of   the
   division */
```

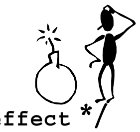
- Multiplication operators (*, /, %) have a higher priority than the additive operators (+, -). When evaluating an expression, operators with a higher priority are evaluated first:

```
x = 2 + 3 / 2 + 3;
          /* x = 2 + 1 + 3 */

x = (2 + 3) / (2 + 3);
          /* x = (5 / 5) = 1 */

    x = 4*(1/2);      /* x = 0 */
    x = 4*1/2;        /* x = 2 */
```

Short-hand operators

- C allows for a short hand notation that introduces side effects. This is done through the prefix- or postfix operators ++, --
- If ++ or -- are used as prefixes, the variable is modified before it is used in the evaluation of an expression:

```
a = 3;
b = ++a + 3;    /* b = 4 + 3 = 7 and a = 4 side effect */
```



- If ++ or -- are used as postfixes, the variable is first used to evaluate the expression and then modified.

```
a = 3;
b = a++ + 3;    /* b = 3 + 3 = 6 and a = 4 */
```

- Almost all operators can be combined with =

```
a += b;   /* a = a + b */
a  *= b;  /* a = a * b */
```


Bit-Operators

- The Bit-Operators & (AND), ^ (Exclusive-OR), and | (Inclusive-OR) manipulate bits according to standard two valued logic

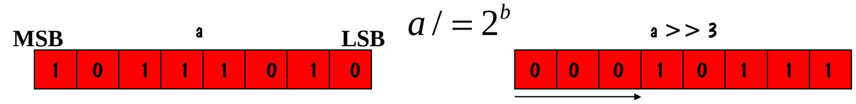
Bit1	Bit2	Bit1 & Bit2	Bit1 ^ Bit2	Bit1 | Bit2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

- With & one can set bits to 0.
- With ^ one can reverse the valu of bits (0 becomes 1 and 1 becomes 0)
- With | one can set bits to 1.

Shift Operators

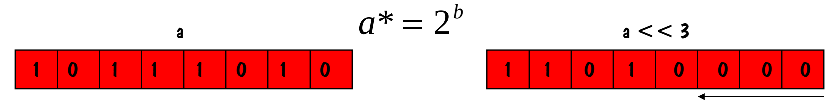
- << and >> are used to manipulate the position of the bits in a byte or a word
- With a >> b the bits in the variable a are displaced b positions to the right (the new bits are filled with 0).

$$a /= 2^b$$



- With a << b the bits in the variable a are displaced b positions to the left (the new bits are filled with 0).

$$a *= 2^b$$



Comparison and logical operators

- The comparison operators return a 1 or a 0 depending on the result of the comparison. The comparison operators in C are
  - < (smaller than)
  - > (greater than)
  - <= (smaller or equal than)
  - >= (greater or equal than)
  - == (equal than)
  - != (not equal than)
- && and || are the logical AND and OR operators

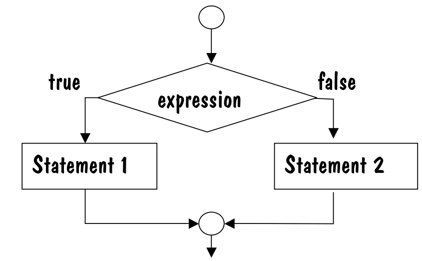
```
( a != b) && (c > d)
(a < b) || (c > d)
```


if statement

- The if – then – else statement can be used with or without the else. The two forms are:

```
if (expression)
    statement1

if (expression)
    statement1
else
    statement2
```

- In both cases, when the expression is true, then statement1 is executed. If the expression is false, then, in the first case, statement1 is skipped (not executed), and, in the second case, statement2 after the else is executed.




```
if (a >= 3) a = a - 3;
if (a == 3) a = a * 3;
else a = a * 5;
```

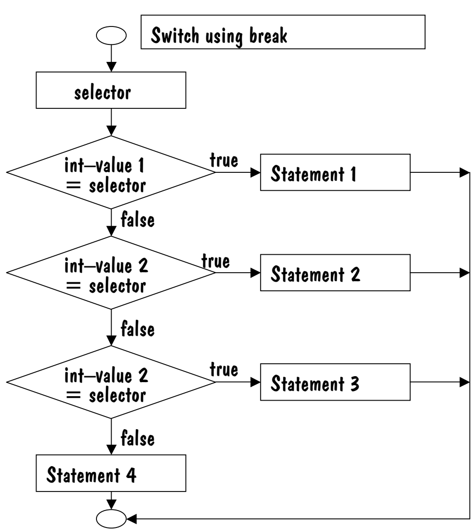
```
if (a >= 3) { a = a - 3;
if ( a == 3) a = a * 3;}
else a = a * 5;
```


switch statement (1)

- The switch statement is used to conditionally perform statements based on an integer expression (selector)

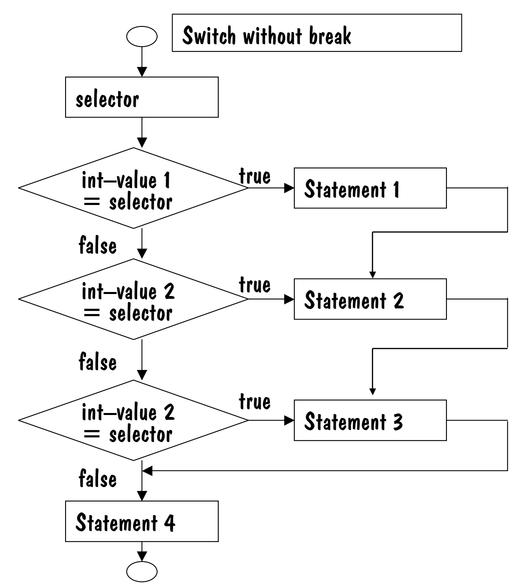
```
switch (selector) {
case int-value1 : statement1; break;
case int-value2 : statement2; break;
case int-value3 : statement3; break;
default: statement4;
}
```

- The exact behavior of the switch statement is controlled by the break and default commands
  - break continues execution after the switch statement
  - default is executed if no other match is found



Switch statement (2)

```
switch (selector) {
case int-value1 : statement1;
case int-value2 : statement2;
case int-value3 : statement3;
default: statement4;
}
/* fall through */
```



Switch (example)

```
char a = 'A';
switch (a) {
 case 'A': x *= x; break;
 case 'B': x /= x; break;
 default: x += 5;
 }
```

- Once the case 'A' is found, x *= x is executed and then we continue after the switch statement. in all cases, only one case will be executed (either 'A' or 'B' or 'default')
- Note that a is of type char. It does not matter, char is treated as an integer using type conversion

```
a = 'A';
switch (a) {
  case 'A': x *= x;
  case 'B': x /= x;
  default: x += 5;
  }
```

- Once case 'A' is found, x *= x is executed. However, there is no break. This means we continue executing statements while ignoring the cases (the check is not performed anymore). Thus, the following statements will also be executed

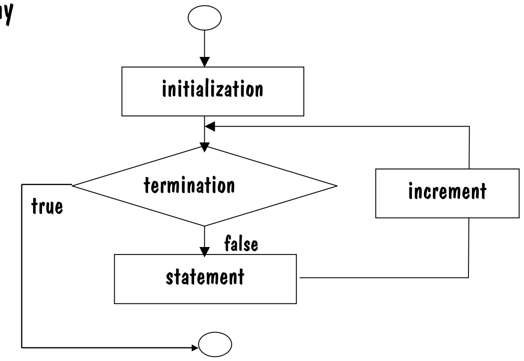
```
x /= x;
x += 5;
```


for statement

- The for statement provides a compact way to iterate over a range of values.

```
for (initialization; termination; increment) {
   statement
  }
```

- All elements in the for loop are optional:
  `for (; ; );`
- break can be used to interrupt the loop without waiting for the termination condition to evaluate to true
- continue can be used to skip execution of the body of the loop and re-evaluate the termination condition



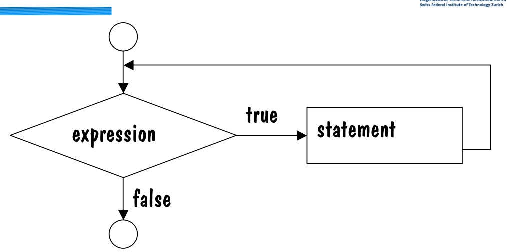

```
for (int x =0; x < 100; x = x + 3){
             if (x == 27) continue;
             else printf("%d",x);
             }
```


while statement

- The while statement is used to continually execute a block of statements while a condition remains true.

```
while (expression) {
       statement
    }
```

- As before, break and continue can be used to terminate the loop or to finish an iteration and go back to the evaluation of the expression
- for and while are equivalent (can you do it? write a for loop using the while statement and vice versa)



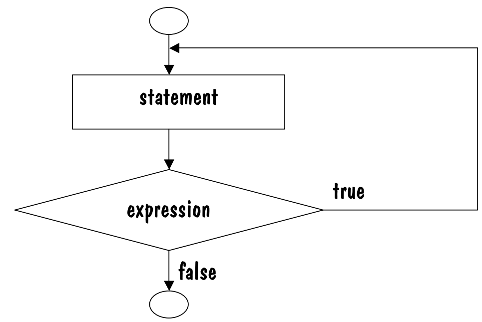

```
main() {
 char t;
 while((t = getchar()) != '!') {
  if (t >= 'A' && t<= 'Z')
   printf("%c\n", (char)(t+'a'-'A'));
  else
   printf( "%c\n", (char)t);
 }
}
```


Do-while statement

- The do-while is similar to the while statement except that the loop is always executed once and the condition is checked at the end of each iteration.

```
do {
   statement
 } while (expression)
```

- break and continue have the same effect as before



Odd ends

exit ()

- exit() terminates the execution of the program passing an integer error code. e.g. 0 -> no error, 1 -> not found, 99 -> crash
- exit() is a very primitive way to terminate a program and one that leaves only very limited chance to deal with failures. More modern programming languages use exceptions and exception propagation mechanisms to indicate the occurrence of an error without having to terminate the program

```
if (!(buf = AllocMem (BufSize);)) {
  printf("kein Speicher vorhanden");
  exit(NO_MEM);
}
```

Goto

- C was written as a language that is one step above assembly language. This can be seen, for instance, on the existence of a goto statement
- Do not use goto when programming. There are very good reasons to avoid it:
  - Style: avoid spaghetti code
  - Use of stack frames

Arrays

- An array is a finite set of variables of the same basic type
- Instead of giving each variable a name, we use enumeration and group all of them in an array
- The enumeration of elements within an array always starts with 0. If the array has N elements, the last element is in position N-1
- The C compiler does not check the array boundaries
  - this is a very typical error that it is very difficult to find (usually happens inside loops that traverse the array)
  - always check the array boundaries before accessing a variable in an array

```
#include <stdio.h>
float data[5];  /* data to average and total */
float total;    /* the total of the data items */
float average;  /* average of the items */

main() {
   data[0] = 34.0;
   data[1] = 27.0;
   data[2] = 45.0;
   data[3] = 82.0;
   data[4] = 22.0;

   total = data[0] + data[1] + data[2] + data[3] +
     data[4];
   average = total / 5.0;
   printf("Total %f Average %f\n", total, average);
   return (0);
}
```

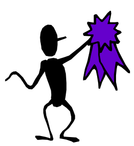

Multi-dimensional arrays

```
int a[3][3]
```

MEMORY

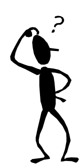
1	2	3	4	5	6	7	8	9
a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	a[2][1]	a[2][2]

```
int a = 1;
for (i=0; i < 3; i++)
        for (j=0; j < 3; j++)
                matrix[i][j] = a++;
```



1	2	3	4	5	6	7	8	9

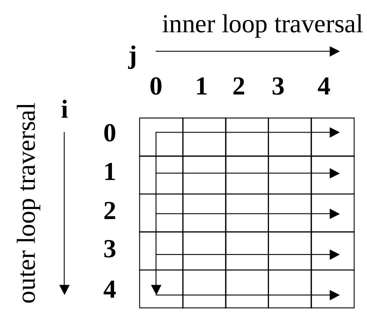
```
int a = 1;
for (i=0; i < 3; i++)
        for (j=0; j < 3; j++)
                matrix[j][i] = a++;
```



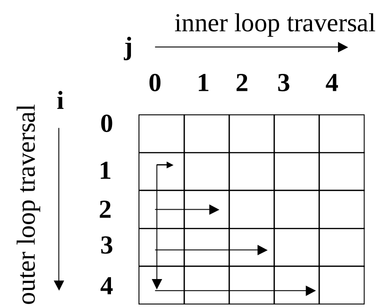
1	4	7	2	5	8	3	6	9

Array traversals (examples)

```
int array[5][5];
for (int i=0; i < 5; i++)
   for (int j=0; j < 5; j++)
     array[i][j] = 1;
```




```
int array[5][5];
for (int i=0; i < 5; i++)
        for (int j=0; j < i; j++)
           array[i][j] = 1;
```



More on arrays

- Arrays can be initialized when they are defined:

```
/* a[0] = 3, a[1] = 7, a[2] = 9 */
int a[3] = {3, 7, 9};

/* liste[0]=0.0, …, liste[99]=0.0 */
float list[100] = {};

int a[3][3] = {
    { 1,  2, 3}
    { 4, 5, 6}
    { 7, 8, 9}
};
```

1	2	3	4	5	6	7	8	9

- Strings are arrays of characters terminated with the null character \0

```
char str[6] = {'h','a','l','l','o','\0'}

char str[6] = "hello";
```

- For string manipulation, however, use the string.h library
- In C, arrays are just a syntactic convenience to make programming easier. Arrays are, for the compiler, the same as pointers (the array name is a pointer to the beginning of the array)

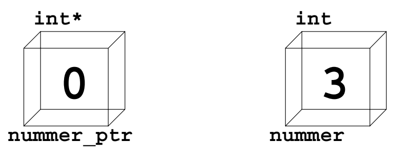
Pointers

- A variable has a name, an address, a type, and a value:
  - the name identifies the variable to the programmer
  - the address specifies where in main memory the variable is located (i.e., the beginning of the memory region reserved for this variable)
  - the type specifies how to interpret the data stored in main memory and how long the variable is
  - the value is the actual data stored in the variable after if has been interpreted according to a given type
- Pointers are language constructs that allow programmers to directly manipulate the address of variables
- Pointers are used with the * and & operators:

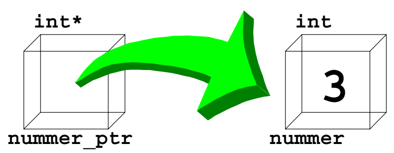
```
int* px;  /*px = pointer to an integer*/

int x;    /*x is an integer */

px = &x; /* px gets the address of x */
         /* or px points to x */

x = *px; /* x gets the contents of */
         /* whatever x points to  */
```


Pointers (I)

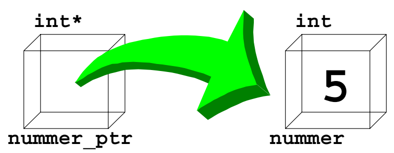
```
int nummer = 3;
int* nummer_ptr = NULL;
```




```
nummer_ptr = &nummer;
```

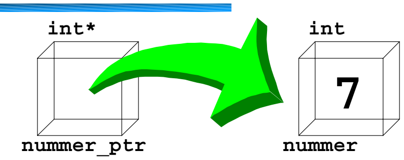



```
nummer = 5;
```



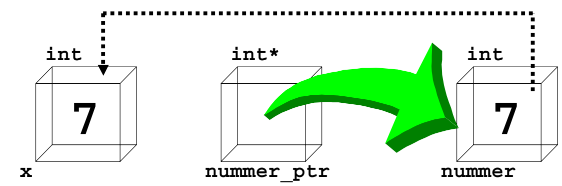
Pointers (II)

```
*nummer_ptr = 7;
```



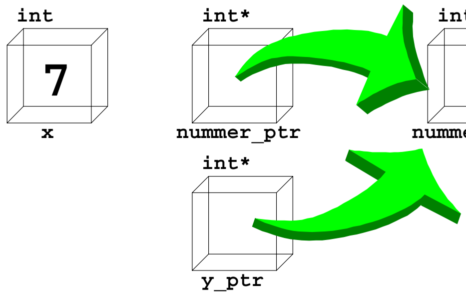

```
*nummer = 8;    /* CAREFUL, 8 is not a pointer but an integer */
```

```
int x = *nummer_ptr;
```

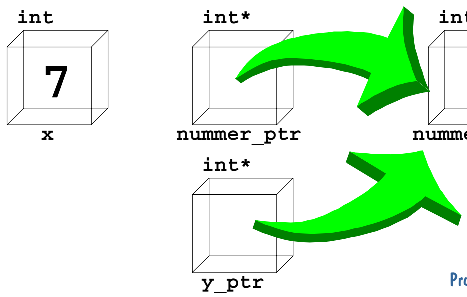


Pointers (III)

```
int* y_ptr = nummer_ptr;
```

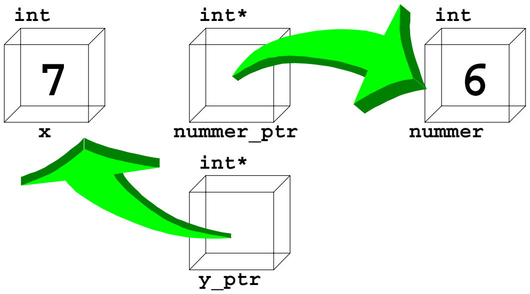



```
*y_ptr = 6;
```



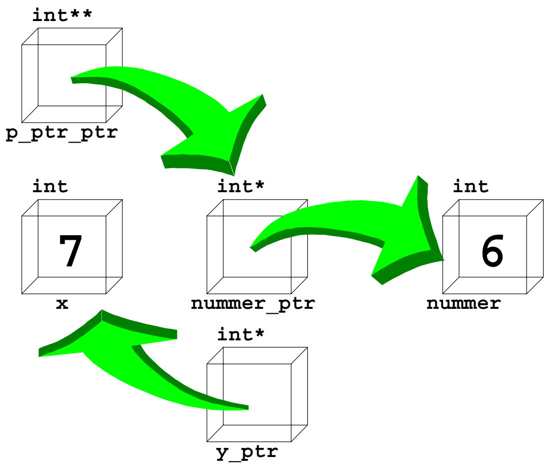
Pointers (IV)

```
y_ptr = &x;
```



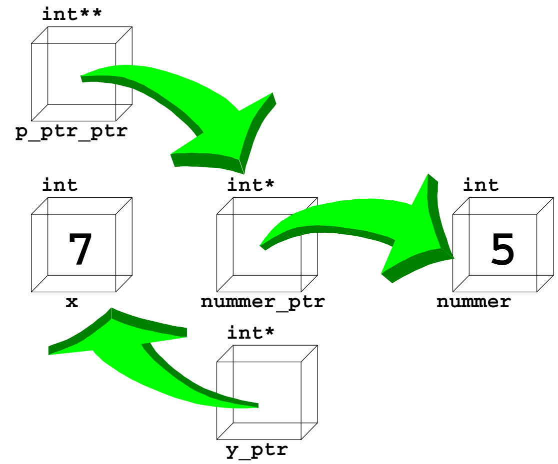
Pointers (V)

```
int* *p_ptr_ptr;
p_ptr_ptr = &nummer_ptr;
```



Pointers (VI)

```
*(*p_ptr_ptr) = 5;
```



Pointers, arrays and strings

- An array is in reality a pointer:

```
int a[10], y;
int* px;
px = a;      /* px points to a[0] */
px++;    /* px points to a[1]   */
px=&a[4]; /*px points to a[4] */
y = *(px+3) /*y gets the value*/
                   /* in a[3]        */
```

- The pointer arithmetic in C guarantees that if a pointer is incremented or decremented, the pointer will vary according to its type. For instance, if px points to an array, px++ will always yield the next element independently of what is the type stored in the array
- Strings can be manipulated through pointers:

```
char* message;
message = "This is a string";
```

- message is a pointer that now points to the first character in the string "This is a string"
- Again, use the string.h for string manipulation rather than doing it directly (you will avoid many errors)

Troubles with pointers

What is printed by the following code?

```
#include <stdio.h>
void f(int *aa, int *bb) {
    *bb = 8;
    aa[1] = bb[2];
    aa = bb;
}

main() {
    int a[5] = { 1, 2, 3, 4, 5 }, *b;
    b = a + 2;
    f(a,b);
    printf("%d %d %d %d %d\n",
            a[0], a[1], a[2], a[3], a[4]);
}
```

What is printed by the following code?

```
#include <stdio.h>
void g(int *aa, int *bb) {
    bb[2] = aa[-2];
    *aa++ = 17;
    *++aa = 10;
}

main() {
    int blap[7] = { 1, 2, 3, 4, 5, 6, 7 };
    int *c = blap + 3;
    g(c,blap);
    printf("%d %d %d %d %d %d %d\n",
    blap[0], blap[1], blap[2], blap[3],
    blap[4], blap[5], blap[6]);
    }
```


Structures

- Structures allow programmers to define complex data types. A structure is a new (user defined) data type:

```
struct Id_card {
    char name[100];  /* Name */
      char adresse[100]; /*Address */
      short int geburtsjahr; /*Geburtsjahr*/
      int telefon;   /* Telefonnummer */
      short int semester; /* Semester */
} ethz, uniz;

struct Id_card erasmus;
```

- Structures of the same type can be copied with the operator = but they should not be compared with ==
- Access to the elements of a structure is as follows:

```
ethz.name = "Gustavo";
ethz.telefon = 1234567;
```

- Pointers can also refer to structures, in which case elements are accessed through the ->, or * operators:

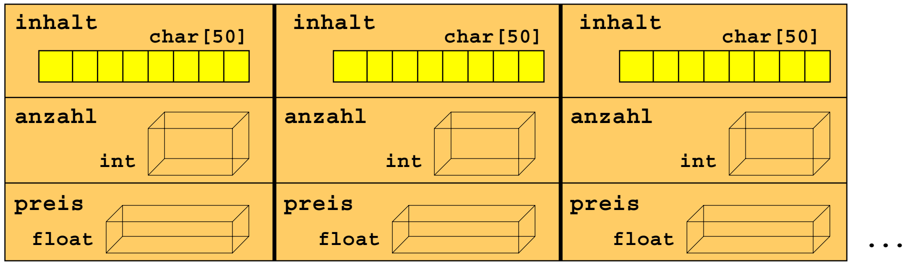
```
struct Id_card *pid;
pid = &ethz_student;
pid->name = "Gustavo";
pid->telefon = 1234567;
(*pid).name = "Gustavo";
(*pid).telefon = 1234567;
```

- In ANSI C, structures can be passed as arguments (by value or by reference) and can also be the return type of a function (this is not true in earlier versions of C)

Example structures

```
int main () {
       struct Typ_kiste {
              char inhalt[50];    /* was ist in der Kiste */
              int anzahl;         /* wieviel davon */
              float preis;        /* was kostet eine Einheit */
       };

       float wert;
       const int MAX_KISTEN = 10;
       struct Typ_kiste liste_kisten[MAX_KISTEN];

       /* Initialisierung … */

       /* Gesammter Wert */
       for (int i = 0; i < MAX_KISTEN; i++)
         wert += liste_kisten[i].anzahl * liste_kisten[i].preis;
       …
       }
```

struct



liste_kisten[0]	liste_kisten[1]	liste_kisten[2]
liste_kisten[0].inhalt	liste_kisten[1].inhalt	liste_kisten[2].inhalt
liste_kisten[0].anzahl	liste_kisten[1].anzahl	liste_kisten[2].anzahl
liste_kisten[0].preis	liste_kisten[1].preis	liste_kisten[2].preis

Functions

- C is a modular language where the main unit of composition is the function
- A function has the following elements:
  - a return type: specifies the type of the value returned by the function when it terminates
  - a function name: identifies the function for the programmer
  - arguments of defined types: parameters to pass to the function, which can be
    - by value: the function gets the a copy of the value of the parameters but cannot modify the actual parameters
    - by reference: the function gets the address (reference) of the parameters and can modify them
- General syntax:

```
returntype function_name(def of parameters) {
        localvariables
        functioncode
    }
```

- An example:

```
float findaverage(float a, float b) {
        float average;
        average=(a+b)/2;
        return(average);
    }
```

- In ANSI C functions must be declared as prototypes before they are defined:

```
float findaverage(float a, float b)
```


Examples

```
/*  SWAP.C      exchange values */
#include        <stdio.h>
void swap(float *x, float *y); /* prototype */
main() {
        float   x, y;
        printf("Please input 1st value: ");
        scanf("%f", &x);
        printf("Please input 2nd value: ");
        scanf("%f", &y);
        printf("Values BEFORE 'swap' %f, %f\n", x, y);
        swap(&x, &y);   /*      address of x, y */
        printf("Values AFTER  'swap' %f, %f\n", x, y);
        return 0;
}

/*      exchange values within function */
void swap(float *x, float *y) {
        float   t;
        t = *x; /*      *x is value pointed to by x     */
        *x = *y;
        *y = t;
       printf("Values WITHIN 'swap' %f, %f\n", *x, *y);
}
```

```
\* FACTORIAL *\
\* fact(n) = n*(n-1)*....2*1 *\

#include <stdio.h>

fact(n) {
   int n;
   if (n == 0)  return(1);
   return(n * fact(n-1));
}

main() {
   int n, m;

   printf("Enter a number: ");
   scanf("%d", &n);
   m = fact(n);
   printf("Factorial of %d is %d.\n", n, m);
   exit(0);
}
```


main() is also a function

```
/* program to print arguments from command line */
#include <stdio.h>

main(int argc, char **argv) {
    int i;

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

- argc stands for argument count and it contains how many arguments have been passed in the command line when the program is invoked
- argv is the argument vector (array) and it contains all the arguments passed through the command line
- argc is always at least 1 since argv[0] is the name of the program

```
\* append one file to the another */
#include <stdio.h>
main(int argc, char **argv) {
   int c;
   FILE *from, *to;
   if (argc != 3) {   /* Check the arguments. */
      fprintf(stderr, "Usage: %s from-file to-file\n", *argv);
      exit(1);
   }
   if ((from = fopen(argv[1], "r")) == NULL) {
      perror(argv[1]);          /* Open the from-file */
      exit(1);
   }
   if ((to = fopen(argv[2], "a")) == NULL) {
      perror(argv[2]);     /* Open the to-file */
      exit(1);
   }
/* Read one file and append to the other until EOF */
   while ((c = getc(from)) != EOF)
      putc(c, to);
   /*close the files */
   fclose(from);
   fclose(to);
   exit(0);
}
```

Dynamic memory allocation

- The definition of types and variables help the compiler to understand the program we have written
- The declaration of variables leads to the allocation of memory for those variables. In general, this happens automatically and without intervention of the programmer
- C allows the programmer to allocate and deallocate memory dynamically
- The functions used for memory allocation are in stdlib.h
- Typical function calls are
  - malloc
  - free

```
typedef struct node {
  int x,z;
  struct node *next;
} NODE;

NODE *nptr;
if ((nptr =((NODE)*) malloc(sizeof(NODE)))
     == NULL) {
  printf("No memory - bye bye"); exit(99);
}
```

- malloc returns a pointer to the allocated memory. The pointer is generic (void *) and it is a good practice to cast the pointer to the appropriate pointer type to avoid errors.
- Allocated memory must be returned to the system:

```
free(nptr);
```


Example dynamic array

```
/* This program simply reads integers into a dynamic
   array until eof. The array is expanded as needed */

#include <stdio.h>
#include <stdlib.h>
#define INIT_SIZE      8      /* Initial array size. */
main() {
      int num;        /* Number of integers */
      int *arr;       /* Array of integers. */
      int arrsize;    /* The size of the array of integers. */
      int m;          /* Index. */
      int in;         /* Input number. */

      /* Allocate the initial space. */
      arrsize = INIT_SIZE;
      arr = (int*) malloc(arrsize*sizeof(int));

/* Read in the numbers. */
num = 0;
while(scanf("%d", &in) == 1) {
      /* See if there's room. */
      if(num >= arrsize) {
            /* There's not.  Get more. */
            arrsize *= 2;
            arr = (int*) realloc(arr, arrsize*sizeof(int));
            if(arr == NULL){
               fprintf(stderr, "Allocation failed %d.\n");
               exit(18);
            }
      }

      /* Store the number. */
      arr[num++] = in;
}

/* Print out the numbers. */
for(m = 0; m < num; ++m)
      printf("%d\n", arr[m]);
}
```


Example string library

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

void main() {
char name1[12], name2[12], mixed[25];
char title[20];

  strcpy(name1, "Rosalinda");
  strcpy(name2, "Zeke");
  strcpy(title, "This is the title.");

  printf("     %s\n\n", title);
  printf("Name 1 is %s\n", name1);
  printf("Name 2 is %s\n", name2);

  if(strcmp(name1, name2) > 0)
            /* returns 1 if name1 > name2 */
    strcpy(mixed, name1);
  else
    strcpy(mixed, name2);

  printf("The biggest name alphabetically is %s\n", mixed);

  strcpy(mixed, name1);
  strcat(mixed, " ");
  strcat(mixed, name2);
  printf("Both names are %s\n", mixed);
}
```

```
     This is the title.

Name1 is Rosalinda
Name2 is Zeke
The biggest name alphabetically is Zeke
Both names are Rosalinda  Zeke
```


References

Some material for these foils and some of the examples have been taken from the following on-line books on C (there are many more):

- *C Programming*, Steve Holmes: http://www.strath.ac.uk/IT/Docs/Ccourse/
- *C language tutorial*: http://www.graylab.ac.uk/doc/tutorials/C/index.htm
- *Programming in C*, A. D. Marshall: http://www.cs.cf.ac.uk/Dave/C/CE.html

For how C was developed, read the tutorial written by Brian W. Kernighan in 1974:

- *Programming in C: A Tutorial*, B. W. Kernighan: http://www.lysator.liu.se/c/bwk-tutor.html