Introduction to C

Operating Systems UE 2022W

David Lung, Florian Mihola, Andreas Brandstätter, Axel Brunnbauer, Peter Puschner

Technische Universität Wien
Computer Engineering
Cyber-Physical Systems

2022-10-03

Part I

Introduction

History

- 1964: MIT, General Electrics, Bell Labs and AT&T wanted to create a new operating system (Multics)
  - 1969: Too expensive ⇒ Bell Labs quits

History

- 1964: MIT, General Electrics, Bell Labs and AT&T wanted to create a new operating system (Multics)
  - 1969: Too expensive ⇒ Bell Labs quits
- Group around Ken Thompson (Bell Labs) is looking for alternatives to Multics and wanted to create the OS in assembler

- not portable
- time consuming
- prone to errors

```
movl    -8(%ebp, %edx, 4), %eax
movl    -4(%ebp), %eax
movl    (%ecx), %edx
leal    8(,%eax,4), %eax
leal    (%edx,%eax,2), %eax
```

History

- 1964: MIT, General Electrics, Bell Labs and AT&T wanted to create a new operating system (Multics)
  - 1969: Too expensive ⇒ Bell Labs quits
- Group around Ken Thompson (Bell Labs) is looking for alternatives to Multics and wanted to create the OS in assembler

- not portable
- time consuming
- prone to errors

```
movl    -8(%ebp, %edx, 4), %eax
movl    -4(%ebp), %eax
movl    (%ecx), %edx
leal    8(,%eax,4), %eax
leal    (%edx,%eax,2), %eax
```

- Alternatives to assembler were needed. C was developed as successor to the language B, ALGOL (ALGOrithmitc Language)

Why C ?

- Past:
  - Portability
  - Extensibility with libraries

Why C ?

- Past:
  - Portability
  - Extensibility with libraries
- Today:
  - Performance (compare OS-kernel: Windows, Linux, BSDs, ...)
  - Many libraries are available
  - Programming hardware
  - Computer graphics and games
  - Modern languages/interpretors are written in C (Python, Perl, Ruby, ...)
  - A lot of compilers generate C-code (e.g., Matlab/Simulink)

Standards

- 1978: De facto standard by Ritchie and Kernighan in the book The C Programming Language

[1] http://gcc.gnu.org

Standards

- 1978: De facto standard by Ritchie and Kernighan in the book The C Programming Language
- 1989: C-89 / ANSI-C

[1] http://gcc.gnu.org

Standards

- 1978: De facto standard by Ritchie and Kernighan in the book The C Programming Language
- 1989: C-89 / ANSI-C
- 1999: C-99
  - Not supported by all compilers
  - Even gcc does not fully support it
  - This standard is used for OSUE lab exercises

```
$ gcc -std=c99 -pedantic -Wall \
      -D_DEFAULT_SOURCE -g -c filename.c
```

[1] http://gcc.gnu.org

Standards

- 1978: De facto standard by Ritchie and Kernighan in the book The C Programming Language
- 1989: C-89 / ANSI-C
- 1999: C-99
  - Not supported by all compilers
  - Even gcc does not fully support it
  - This standard is used for OSUE lab exercises

```
$ gcc -std=c99 -pedantic -Wall \
      -D_DEFAULT_SOURCE -g -c filename.c
```

- 2011: C-11

[1] http://gcc.gnu.org

Standards

- 1978: De facto standard by Ritchie and Kernighan in the book The C Programming Language
- 1989: C-89 / ANSI-C
- 1999: C-99
  - Not supported by all compilers
  - Even gcc does not fully support it
  - This standard is used for OSUE lab exercises

```
$ gcc -std=c99 -pedantic -Wall \
      -D_DEFAULT_SOURCE -g -c filename.c
```

- 2011: C-11
- today: new quasi-standard (at least in the free/open source community) with gcc[1] and gnu extensions
  - However, some gnu-extensions are specified only informally
  - Recently LLVM/clang appeared as a potential successor to gcc

[1] http://gcc.gnu.org

Hello, C World

```
#include <stdio.h>

int main(void)
{
  printf("Hello, C World\n");

  return 0;
}
```

Compilation

- Source code needs to be translated to machine code
- Code → pre-processor → compiler → linker

```
$ gcc -o prog prog.c # all done in one step
$ ./prog # start the program
```

Compilation

- Source code needs to be translated to machine code
- Code → pre-processor → compiler → linker

```
$ gcc -o prog prog.c # all done in one step
$ ./prog # start the program
```

Single steps (fyi only):

- pre-processor:

```
$ gcc -E prog.c
```

Compilation

- Source code needs to be translated to machine code
- Code → pre-processor → compiler → linker

```
$ gcc -o prog prog.c # all done in one step
$ ./prog # start the program
```

Single steps (fyi only):

- pre-processor:

```
$ gcc -E prog.c
```

- Compiler, linker:

```
$ gcc -v -o prog prog.c
[..]
<..>/cc1 [..] prog.c [..] -o /tmp/ccpMJ9ab.s
[..]
as -V -Qy -o /tmp/ccdR6Ueb.o /tmp/ccpMJ9ab.s
[..]
<..>/collect2 [..] -o prog [..] crtn.o
```

Comments

```
/* I am a comment in C-89 */

// I am a comment in C-99 standard
// I end at the end of the line

/* multi-line comments
   require the old syntax */
```

Code

- comment (functions, etc.)
- structure (indent, line breaks, etc.)

Part II

Variables & Constants

Definition

- For variables memory space needs to be reserved (depending on the data type)
- The name is set
- This happens at the definition
- The definition of a variable must happen only one time in the code

```
int i; // Integer variable i, declaration + definition

// Function declaration + definition:
int f(void)
{
    ...
}
```

Declaration

- Variables have a type
- The compiler needs to know this type
- This is done with the declaration

```
extern int j; // declared, but defined somewhere else

/* Function declaration
   (but not defined, i.e. no body): */
int f(void);
```

Declaration

- Variables have a type
- The compiler needs to know this type
- This is done with the declaration

```
extern int j; // declared, but defined somewhere else

/* Function declaration
   (but not defined, i.e. no body): */
int f(void);
```

- The declaration can happen several times
- Not each declaration is also a definition
- However, each definition is also a declaration
- The term declaration is often not distinguished from the term definition → declaration is used for both

Initialization

- Although the variable already has its memory, its value is still undefined (unless it was placed in an pre-initialized memory at compile time)
- Initialization assigns a value to a variable
- Assignment is done with **=**

```
int k = 23; /* declaration, definition
               and initialization */
```

Examples

```
int i; /* declaration and definition
          of a single integer variable */

int i, j, k; // -"- of multiple integers at once

int i, j = 23, k = 42; /* same, but some variables
                          are initialized */

int i, char b; // incorrect syntax

int i; char b; /* correct, declares and defines an
                  integer and a character variable */

int i = 4; char b = 'A'; // same with initializations
```

Byte in C99-standard

A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined.

ISO/IEC 9899:TC3, Committee Draft – September 7, 2007

Integral Number

- char: 1 byte (according to the standard a byte does not have to have 8 bit of length). Is often used to store characters and strings

Integral Number

- char: 1 byte (according to the standard a byte does not have to have 8 bit of length). Is often used to store characters and strings
- short int: min. 16 bit
- int: often 32 or 64 bit
- long int: min. 32 bit
- long long int: min. 64 bit. Since C-99

Integral Number

- char: 1 byte (according to the standard a byte does not have to have 8 bit of length). Is often used to store characters and strings
- short int: min. 16 bit
- int: often 32 or 64 bit
- long int: min. 32 bit
- long long int: min. 64 bit. Since C-99
- Actual size is available in `<limits.h>`

Integral Number

- char: 1 byte (according to the standard a byte does not have to have 8 bit of length). Is often used to store characters and strings
- short int: min. 16 bit
- int: often 32 or 64 bit
- long int: min. 32 bit
- long long int: min. 64 bit. Since C-99
- Actual size is available in `<limits.h>`
- C-99 introduced standardized types (`<stdint.h>`): e.g., `uint32_t`, `int8_t`, ...

Integral Number

- char: 1 byte (according to the standard a byte does not have to have 8 bit of length). Is often used to store characters and strings
- short int: min. 16 bit
- int: often 32 or 64 bit
- long int: min. 32 bit
- long long int: min. 64 bit. Since C-99
- Actual size is available in `<limits.h>`
- C-99 introduced standardized types (`<stdint.h>`): e.g., `uint32_t`, `int8_t`, ...
- All types have signed and unsigned variants (e.g. signed int, unsigned int), by default everything is signed

Integral Number

- char: 1 byte (according to the standard a byte does not have to have 8 bit of length). Is often used to store characters and strings
- short int: min. 16 bit
- int: often 32 or 64 bit
- long int: min. 32 bit
- long long int: min. 64 bit. Since C-99
- Actual size is available in `<limits.h>`
- C-99 introduced standardized types (`<stdint.h>`): e.g., `uint32_t`, `int8_t`, ...

- All types have signed and unsigned variants (e.g. signed int, unsigned int), by default everything is signed
- Literals can be declared hexadecimal (0x as prefix) and octal (0 as prefix), e.g., 0x10 (16 in decimal), 024 (20 in decimal)

Range of Values

- Signed variables have another range of values than unsigned variables
- The following ranges of values are not specified by the standard, they are used for presentation purposes

Type	signed	unsigned
char	-128 to 127	0 to 255
short int	-32.768 to 32.767	0 to 65.535
long int	-2.147.483.648 to 2.147.483.647	0 to 4.294.967.295
...	...	...

Real Numbers

Floating point numbers:

- float: single precision
- double: double precision
- long double: extended precision

Real Numbers

Floating point numbers:

- float: single precision
- double: double precision
- long double: extended precision
- There is no statement about the internal representation in the standard

Real Numbers

Floating point numbers:

- float: single precision
- double: double precision
- long double: extended precision
- There is no statement about the internal representation in the standard
- Signed and unsigned are not differentiated → it's always signed

sizeof

- The operator `sizeof` is used to obtain the memory consumption of a type

```
int i;
printf("%lu byte(s)\n", sizeof i);
printf("%lu byte(s)\n", sizeof (int));
```

Constants

```
const int i = 23; // C constant

#define MYCONST 23 // pre-processor constant;
                   // all ocurrences are replaced
                   // with 23 by the pre-processor
```

- `const` defines a typed constant in the code. Should/Can not be changed
- `MYCONST` is replaced by the pre-processor

Scope

- Variables are visible only within their block

```
#include <stdio.h>

int main(void)
{
  int i = 23, j = 42;
  {
    int i; // redeclaration of i within a new block
    i = 2323; // assigning the local i

    printf("%d, ", i);
    printf("%d, ", j);
  }
  printf("%d\n", i); /* in this block the value
                        of i has not changed */
  return 0;
}
```

Scope

- Variables are visible only within their block

```
#include <stdio.h>

int main(void)
{
  int i = 23, j = 42;
  {
    int i; // redeclaration of i within a new block
    i = 2323; // assigning the local i

    printf("%d, ", i);
    printf("%d, ", j);
  }
  printf("%d\n", i); /* in this block the value
                        of i has not changed */
  return 0;
}
```

```
$ 2323, 42, 23
```

C99

- Before C-99, variables had do be declared at the beginning of a block
- With C-99 (which we are using) this is no longer required

```
#include <stdio.h>

int main(void)
{
   /* i, j not at the beginning of the block */
   for (int i = 0; i < 10; ++i)
   {
      printf("%d\n", i);
      int j = 23;
      printf("%d\n", j);
   }
   return 0;
}
```

static

- `static` assigns to a variable a fixed memory space, its state remains
- A static variable cannot be accessed from an outside block or file

```
#include <stdio.h>

void foo()
{
  static int i = 23;
  printf("%d, ", i);
  i = i + 1;
}
```

```
int main(void)
{
  foo();
  foo();
  foo();
  return 0;
}
```

static

- `static` assigns to a variable a fixed memory space, its state remains
- A static variable cannot be accessed from an outside block or file

```c
#include <stdio.h>

void foo()
{
  static int i = 23;
  printf("%d, ", i);
  i = i + 1;
}
```

```c
int main(void)
{
  foo();
  foo();
  foo();
  return 0;
}
```

```
$ 23, 24, 25,
```

extern

- Declares variables which are defined in another file

inc.c

```
int g_variable = 1;
[..]
  g_variable++;
[..]
```

dec.c

```
extern int g_variable;
[..]
  g_variable--;
[..]
```

volatile

- Variable can change outside of the program context
- Important for hardware oriented programming (e.g., interrupt handler that change the values of variables)

[2] https://lwn.net/Articles/234017/

volatile

- Variable can change outside of the program context
- Important for hardware oriented programming (e.g., interrupt handler that change the values of variables)
- (fyi only:) The implementation of volatile is compiler specific; a 'clean' solution uses Memory Barriers[2]

[2] https://lwn.net/Articles/234017/

volatile

- Variable can change outside of the program context
- Important for hardware oriented programming (e.g., interrupt handler that change the values of variables)
- (fyi only:) The implementation of volatile is compiler specific; a 'clean' solution uses Memory Barriers[2]

```
volatile char keyPressed = ' ';
long count = 0;
while (keyPressed != 'x') {
   ++count;
}
```

Without `volatile`, the while-loop would by optimized to `while(1)` by the compiler, because from the compiler's point of view the variable never changes

[2] https://lwn.net/Articles/234017/

Example

```
extern const volatile unsigned long int rt_clk;
```

Example

```
extern const volatile unsigned long int rt_clk;
```

A "'long int"' variable, no sign, values can't be assigned (but the value can be read), the value can change outside of the program context and it is defined somewhere else

Increment/Decrement

- Using **++** and **- -** variables can be incremented or decremented by one

Increment/Decrement

- Using `++` and `--` variables can be incremented or decremented by one
- Prefix (`++i`) und postfix (`i++`) are possible:
  - Prefix operator in/decrements, returns new value
  - Postfix operator in/decrements, returns old value
- Use prefix operator if possible (also with regard to C++)

Increment/Decrement

- Using `++` and `--` variables can be incremented or decremented by one
- Prefix (`++i`) und postfix (`i++`) are possible:
  - Prefix operator in/decrements, returns new value
  - Postfix operator in/decrements, returns old value
- Use prefix operator if possible (also with regard to C++)

```
int n;
int m = 0;
n = ++m;
$ n = 1, m = 1
```

```
int n;
int m = 0;
n = m++;
$ n = 0, m = 1
```

Ordering/Associativity

```
c = sizeof (x) + ++a / 3;
```

```
c = (sizeof (x) + ((++a) / 3));
```

Ordering/Associativity

```c
c = sizeof (x) + ++a / 3;
```

```c
c = (sizeof (x) + ((++a) / 3));
```

```c
a = 5 / 2 * 3;
```

```c
a = (5 / 2) * 3; /* left to right */
```

Ordering/Associativity

```
c = sizeof (x) + ++a / 3;
```

```
c = (sizeof (x) + ((++a) / 3));
```

```
a = 5 / 2 * 3;
```

```
a = (5 / 2) * 3; /* left to right */
```

```
i = 3;
a = i + i++;
/* i == 4, a == ? (according to the standard
 * it depends on the compiler implementation!) */
```

Ordering/Associativity

```c
c = sizeof (x) + ++a / 3;
```

```c
c = (sizeof (x) + ((++a) / 3));
```

```c
a = 5 / 2 * 3;
```

```c
a = (5 / 2) * 3; /* left to right */
```

```c
i = 3;
a = i + i++;
/* i == 4, a == ? (according to the standard
 * it depends on the compiler implementation!) */
```

```c
i = 2;
a = i++ + ++i; /* ??? */
```

Equality- & Logic Operators

Operator	Explanation
$<$	less than
$>$	greater than
$<=$	less than or equal
$>=$	greater than or equal
$! =$	not equals
$==$	equals
$\&\&$	logical and
$\vert\vert$	logical or
!	negation

Equality- & Logic Operators

Operator	Explanation
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
!=	not equals
==	equals
&&	logical and
||	logical or
!	negation

- || and && are evaluated short circuit

Bitwise Operators

Operator	Explanation
&	and
—	or
$\wedge$	exclusive or (xor)
$\sim$	bit-wise complement
$>>$	shift right
$<<$	shift left

Bitwise Operators

Operator	Explanation
&	and
—	or
$\wedge$	exclusive or (xor)
$\sim$	bit-wise complement
>>	shift right
<<	shift left

x:	1	0	0	1	1	0	1	1
y:	0	0	0	1	0	0	0	1
x & y:	0	0	0	1	0	0	0	1

Bitwise Operators

Operator	Explanation
&	and
—	or
∧	exclusive or (xor)
∼	bit-wise complement
>>	shift right
<<	shift left

x:	1	0	0	1	1	0	1	1
y:	0	0	0	1	0	0	0	1
x & y:	0	0	0	1	0	0	0	1

- For bitwise and arithmetic operators there are the versions `Op=` (e.g., `i += 5` which is the same as `i = i + 5`)

Shift

- << and >> do bit-wise shifting

```
unsigned char i = 7; /* 00000111 */
i <<= 1; /* 00001110 */
printf("%d\n", i); /* 14 */

/* 128 == 2 to the power of 7 */
printf("%d\n", 1 << 7);
```

Shift

- << and >> do bit-wise shifting

```
unsigned char i = 7; /* 00000111 */
i <<= 1; /* 00001110 */
printf("%d\n", i); /* 14 */

/* 128 == 2 to the power of 7 */
printf("%d\n", 1 << 7);
```

- The behavior of signed variables with negative values is undefined

```
int i = -7;
i <<= 1;
printf("%d\n", i); /* -14 ??? undefined */
```

Example

```
unsigned char a, b, c;
a = 4; b = 2; /* binary: a = 100, b = 010 */
c = a | b; /* c = 6 */
b = a & c; /* b = 4 */
a += 3; /* a = a + 3 = 7 */
b %= 3; /* b = b % 3 = 1 (% .. modulo div) */
b = 0;
if ( (b > 0) && ( (a / b) > 5) ) /* ... */
```

Part III

Control Structures

if

```
if (expression)
   statement
else if (expression)
   statement
else if (expression)
   statement
/* . . . */
else
   statement
```

- In C: 0 is false, everything else is true (even -1)

if

```
if (expression)
   statement
else if (expression)
   statement
else if (expression)
   statement
/* . . . */
else
   statement
```

- In C: 0 is false, everything else is true (even -1)
- Tip: never go without/forget embracing the `statement`-blocks; do also embrace one-line statements with `{}`

goto fail; goto fail;

Apple's libsecurity_ssl, `sslKeyExchange.c`:

```
SSLVerifySignedServerKeyExchange(..)
.
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) !=
      goto fail;
      goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;
.
  fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

goto fail; goto fail;

Apple's libsecurity_ssl, `sslKeyExchange.c`:

```
SSLVerifySignedServerKeyExchange(..)
.
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) !=
      goto fail;
      goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;
.
  fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

References:

- http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c
- http://www.theregister.co.uk/2014/02/25/apple_mac_os_x_10_9_2_ssl/

Thanks to Roland Kammerer for this case study!

goto fail; goto fail;

http://teespring.com/goto-fail-goto-fail

goto fail; goto fail;

- Please, go without `goto` in the regular exercises

goto fail; goto fail;

- Please, go without `goto` in the regular exercises
- **Negative example**:

```
#include <stdio.h>
int main(void)
{
  int i = 0;
loopstart:
  ++i;
  if(i >= 5)
  goto printnum;
contloop:
  if (i < 9)
    goto loopstart;
  goto end;
printnum:
  printf("i is %d\n", i);
  goto contloop;
end:
  return 0;
}
```

switch

```
switch (expression)
{
   case const_expr: statements
   case const_expr: statements
   /* . . . */
   default: statements
}
```

switch

```
switch (expression)
{
   case const_expr: statements
   case const_expr: statements
   /* . . . */
   default: statements
}
```

- Only constant values can be used for equality checks
- A `case` should always end with `break`, otherwise the successing `case`s will be evaluated (see example at the end)
- You should always provide a `default` case

for

```
for (expression1; expression2; expression3)
   statement
```

for

```
for (expression1; expression2; expression3)
   statement
```

- All three expression are not mandatory
- Basic example:
  - `expression1`: Init the counter
  - `expression2`: Check whether the loop should continue
  - `expression3`: Incement the counter

while/do-while

```
while (expression)
   statement

do
   statement
while (expression);
```

while/do-while

```
while (expression)
   statement

do
   statement
while (expression);
```

- Do-while executes `statement` at least one time

continue/break

- `continue` continues at the next run of the most inner loop
  - `for`-loop: `expression3` is executed, `expression2` is checked
- `break` exits the most inner loop and continues to run the code after the loop
  - `for`-loop: `expression3` is not executed

Example

```
int i;

for (i = 0; i < 10; ++i)
{
  (void) printf("hello\n");
}

switch (input)
{
   case 'a':
   case 'A':
      printf("a or A\n");
      break;
   default:
      printf("Error");
      break;
}

i = 23;
if (i == 42)
{
  printf("i ist 42\n");
}
```

Part IV

Arrays

One Dimensional

- Arrays are used to combine related values of the same type

```
Type name[size];

int myarray[8];
```

- `myarray` stores 8 integer variables
- Indexed from 0 to 7
- `myarray[8]` out-of-bounds

Multi Dimensional

- Arrays can have multi dimensions
- In C it is basically "syntactic sugar"

```
int myarray[2][3];
int myarray2[2][3][4];
```

Initialization

```c
int myarr[2][3]= {
                  {1,2,3},
                  {4,5,6},
                  };
int myarr2[2][3] = {1,2,3,4,5,6};
/* first version is preferred */
int myarr3[] = {1, 2, 3, 4};
/* if the whole array is initialized
   you do not need to declare the size */
```

Strings

- Strings are arrays of characters (char) (in C)
- Strings are terminated with '\0' by definition; this is essential for functions that work on strings to know the end of the string

```
char string[] = "hello, world";
/* string is auto \0 terminated */
char s[6];
s[0] = 'h'; s[1] = 'e'; s[2] = 'l';
s[3] = 'l'; s[4] = 'o'; s[5] = '\0';
char str[] = {'f','o','o','b','a','r','\0'};

printf("%s\n", s); /* prints "hello" */
s[3] = '\0';
printf("%s\n", s); /* prints "hel" */
```

Part V

Functions

Definition of Functions

```
type name(type1 param1, type2 param2, ...)
{
   /* code */
}
```

- Increase the readability, re-usability and maintainability
- Need to be declared before they can be used

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

int main(void)
{
   int i;
   i = add(2, 3);
   /* i == 5 */
   return 0;
}
```

Prototypes

- Like variable, declaration and definition are differentiated
- A prototype represents a declaration and ends with an ';'

```
/* Prototype */
int add(int a, int b);
/* int add(int x, int v); also okay */
/* int add(int, int); also okay */
/* int add(double, int); wrong,
   because int is used later */

int main(void)
{
   int i;
   i = add(2, 3); /* i == 5 */
   return 0;
}

/* now add can be defined after it has been called */
int add(int a, int b)
{
   return a + b;
}
```

Global vs Local

- Local variables get invisible when the function or the block ends
- Global variables (declared outside of functions, normally at the beginning of the source code) are valid and accessible until the program ends
- Local variables mask global variables
- Local variables have a random value at definition
- Global variables are placed at a memory space which is initialized with 0

Example

```
int i;
int j = 23;

void foo()
{
   int j = 42;
   printf("%d\n", j); /* 42 */
}

int main(void)
{
   int k;
   printf("%d\n", j); /* 23 */
   foo();             /* 42 */
   printf("%d\n", i); /* 0 */
   printf("%d\n", k); /* 1863 (random) */
   return 0;
}
```

Part VI

Pointer

Pointer

- In C the values of variables do not need to be accessed via their names
- This can also be done by pointers
- Pointers are no "black magic", they are variables like others
- Difference: they store an address
- This is important for hardware oriented programming (speed increase)
- Unfortunately it is also prone to errors

Pointer

- In C the values of variables do not need to be accessed via their names
- This can also be done by pointers
- Pointers are no "black magic", they are variables like others
- Difference: they store an address
- This is important for hardware oriented programming (speed increase)
- Unfortunately it is also prone to errors
- Even new programming languages have pointers, however they hide it from the programmer

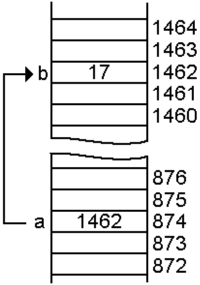
Declaration

- Pointers are declared with `Typ *name`
- The allocated memory does not have the size of `Typ`, instead, it has the size of `Typ *`, in which an address can be stored
- The value, to which a pointer points to, can be accessed with the dereferencing operator *
- The address of a variable can be accessed with the address operator &

```
int *p; /* I'm a pointer */
int* q; /* Me too */
int* a, b; /* a is a pointer, but
              b is _not_ a pointer */
int *a, b; /* a yep, b nope */
int *a, *b; /* a and b are pointers */
```

Memory Layout

```c
int *a;
int b = 17;
a = &b;
printf("value b: %d\n", b); /* 17 */
printf("address b: %p\n", &b); /* 1462 */
printf("value a: %p\n", a); /* 1462 */
printf("value to which a points to: %d\n", *a); /* 17 */
printf("addresse of a: %p\n", &a); /* 874 */
```



Simple Pointer Arithmetic

```
int ar[5] = {1, 2, 3, 4, 5};
int *p;

p = &ar[0];
/* or */
p = ar; /* ar is no pointer, only the address! */

printf("%d\n", *p); /* 1 */
*p += 22;
printf("%d\n", ar[0]); /* 23 */
p += 1; /* pointer points to the next element */
printf("%d\n", *p); /* 2 */
```

Pointer and man

```
$ man strcpy
```

```c
char *strcpy(char *dest, const char *src);

/* my C code */
char *mysrc = "mystring";
char *mydest = /* does not matter at the moment:
                  we have enough memory space */

strcpy(mydest, mysrc); /* ? */
/* ?? or ?? */
strcpy(*mydest, *mysrc); /* ? */
```

`man`-page are read that way: strcpy needs variables to addresses (`*dest, *src`). Where is this Address? In the pointers! So you do not need do dereference them.
⇒ `(void) strcpy(mydest, mysrc)`

Risks of Pointers

- Pointer arithmetic can get risky if you do not work with care
- Attention: null-pointer dereferencing was the most frequent security problem at Red Hat in 2009[3]

```
int ar[5] = {1, 2, 3, 4, 5};
int *p = &ar[0];

/* no way! */
p += 23; /* that might cause a problem */
printf("%d\n", *p); /* FAIL */
p = NULL;
printf("%d\n", *p); /* FAIL */
```

[3] www.awe.com/mark/blog/20100216.html

Part VII

Preprocessor

Preprocessor

- The preprocessor is called before the compiler run
- Is doing simple replacements in the source code (case sensitive)
- Resulting source code can be viewed by running `gcc -E`
- Motivation
  - Past: defining constants, inline code
  - Today: portability. using compiler specifications

A preprocessors tasks (temporal order, not complete):

- fyi: Trigraph → ASCII (e.g., `??)` replaced with `]`)[4]
- Combining lines that are split by `'\'`
- Replace macros and copy files (`#include`) in the source code

[4] en.wikipedia.org/wiki/Digraphs_and_trigraphs

Replacing Constants

```c
#define ANSWER (42) /* Constant */

printf("ANSWER: %d\n", ANSWER);
```

ends up:

```c
printf("ANSWER: %d\n", (42));
```

There is no replacement in string literals.

Conditional Replacements

#if, #ifdef, #ifndef, #elif, #else, #endif:

```
#ifdef WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif
```

```
#if DEBUG >= 2
printf("debug, debug\n");
#endif
```

Macros

- Complex macros with parameters can be defined

```
#define NRELEMENTS(a) (sizeof(a) / sizeof(a[0]))
```

- Macros should be handled with care! There are a lot of risks and side effects

Risks

```c
#define DOUBLE(a) a+a

int x = DOUBLE(5) * 3;
/*  x = 5 + 5 * 3 <=> 5 + (5 * 3)
=> #define DOUBLE (a) ( (a) + (a) ) */
```

Risks

```c
#define DOUBLE(a) a+a

int x = DOUBLE(5) * 3;
/*  x = 5 + 5 * 3 <=> 5 + (5 * 3)
=> #define DOUBLE (a) ( (a) + (a) ) */
```

```c
#define DOUBLE(a) ( (a) + (a) )

int x = 3;
int y = DOUBLE(++x);
/*  y = ( (++x) + (++x) ) */
```

Part VIII

Material

Material

- C Programming Language - Kernighan & Ritchie
- `https://en.wikibooks.org/wiki/C_Programming`
- `https://de.wikibooks.org/wiki/C-Programmierung`