**Concurrent Systems (ComS 527)**

Ali Jannesari

Department of Computer Science

Iowa State University, Spring 2023

# C/C++ PROGRAMMING LANGUAGE

# C PROGRAMMING

# Outline

Introduction

Getting started

Types, arithmetic operators

Control flow

Functions and program

Pointers and arrays

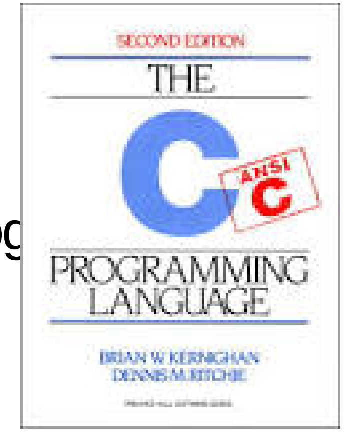
Structures

Input and output

# Literature

Brian W. Kernighan and Dennis M. Ritchie: *The C programming language*

https://hassanolity.files.wordpress.com/2013/11/the_c_prog amming_language_2.pdf



# History of C

- 1966 Martin Richards develops BCPL
- 1969 Ken Thompson and Dennis Ritchie develop B
- 1969-1973 Dennis Ritchie develops C
- 1978 Kernighan & Ritchie publish 1. Edition of "The C programming language". Extensions to C and introduction of the I/O standard library
- 1989 ANSI published a C Standard (ANSI C)
- 1990 ANSI C becomes an ISO standard (C90)
- 1995 C95
- 1999 C99
- 2011 C11

# Comparison with Java

Syntax and key words are very similar

Main differences

- C-Pointers not available in Java
- No classes and objects in C
  - C is pure procedural programming
- Memory management
  - Manual allocation and deallocation / no garbage collection
- No inbuilt string data type in C
- C has preprocessor
- Different standard library functions (affects e.g. I/O)

# Overview

Introduction

Getting started

Types, arithmetic operators

Control flow

Functions and program

Pointers and arrays

Structures

Input and output

# A first program

```
#include <stdio.h>

int main()

{

  printf( “Hello world!\n” );

}
```

The slides use this font for source code, commands or filenames

# Compilation and linking

Compilation:

- Translates the source code into machine code
- Results in an object file with extension `.o`
- Does not resolve accesses to functions/variables in other object files or libraries

Linking:

- Binds one/multiple object files and libraries to an executable
- Resolves accesses to functions/variables in other object files

# Compilation and linking in one step

Various compilers exist:

- `gcc` (GNU C compiler) is an open source compiler
- `icc` Compiler from Intel
- `pgcc` Compiler from Oracle
- `xlc` Compiler from IBM
- Cray compiler

You can compile and link a program in one step:

```
gcc <source file> -o <executable name>
```

E.g.

```
gcc hello.c -o hello
```

# Compilation and linking in separate steps

For large programs with many source files, it can take time to recompile everything

- Recompile only modified files
- `-c` tells the compiler to compile only and do not link
- Default object name has the same base name. E,g.

```
gcc hello.c -c
```

- Results in `hello.o`

Link object files with:

```
gcc <object files> -o <executable name>
```

E.g.

```
gcc f1.o f2.o f3.o f4.o f5.o fx.o -o myprog
```

# Useful compiler arguments

| | |
|---|---|
| `-g` | Add debug information |
| `-o` | Output file name |
| `-c` | Only compile, no linking |
| `-I` | (capital i) Path to search for header files |
| `-L` | Path to search for libraries |
| `-l` | (lowercase L) libraries that should be linked |

# The `make` tool

Building large programs is complex and takes long

`make` manages the build process

It requires a so called `Makefile`

- contains the description how to build a program

Rebuilds only the parts of the program that changed

# Our first make file

```
all: hello

hello: hello.c
	gcc hello.c -o hello
```

**Note:** You need one tabulator in front of `gcc`. Spaces do not work.

# Makefile targets

A Makefile consists of one or multiple targets:

```
target-name: dependency-list
    commands
```

Italic code are a general description. E.g. *`commands`* need to be substituted by the actual commands Non-italic parts are meant literally.

The name of a target is usually the name of the output file

The dependency list is a list of file names or other targets on which this target depends

- The target is only rebuild if at least one file of the dependency list was modified

Arbitrary number of commands possible

- Commands must have a leading tabulator

# Invoke `make`

```
make [-f <my_makefile>] [target]
```

- If no Makefile is specified via `-f`, it searches in the current working directory for a file named `Makefile`
- If no target is specified, it builds the target `all`
- If `all` does not exist, it builds the first target in the `Makefile`

# Debugging

gdb

- Available on most Linux systems
- Command line debugger
  - Works also via ssh connections on remote hosts
- Compile executable with –g for debug symbols

Startup command

```
gdb <binary>
```

After starting gdb, it shows a command line

```
(gdb)
```

# Execution in gdb

| Command | Description |
|---|---|
| `run <program arguments>`<br>`r <program arguments>` | Starts execution of the binary |
| `continue`<br>`c` | Resume execution |
| `next`<br>`n` | Executes until next line of code<br>Does not step into functions |
| `step`<br>`s` | Executes until next line of code<br>Steps into functions |
| `quit`<br>`q` | Exit gdb |

# Breakpoints and watches

| Command | Description |
|---|---|
| `break <function name>`<br>`b <function name>` | Adds a breakpoint at the entry of a function |
| `break <sourcefile>:<linenumber>`<br>`b <sourcefile>:<linenumber>` | Adds a breakpoint at the source code line |
| `watch <condition>` | Suspend execution when a condition is met, e.g. `i == 3` |
| `delete` | Deletes all breakpoints |
| `delete <breakpoint number>` | Delete specific breakpoint |
| `clear <function>` | Deletes all breakpoints in the given function |
| `info break` | Lists all breakpoints |

# Analysis

| Command | Description |
|---|---|
| `where` | Shows current line number and function |
| `backtrace`<br>`bt` | Prints the current stack |
| `backtrace full` | Prints local variables |
| `print <variable name>`<br>`p <variable name>` | Print value of specifyed variable |

- Shows status at current point of execution
- When an error occurs, gdb interrupts execution and allows inspection of the current location

# Thread management

| Command | Description |
|---|---|
| `info threads` | Lists all threads in use |
| `thread <thread id>` | Switches to specific thread |
| `thread apply all <command>` | Applies command to all threads |
| `thread apply <id list> <command>` | Applies command to specified threads |

- Commands are usually applied to only one thread
- Analysis shows data only for one thread

# Info and help

| Command | Description |
|---|---|
| `help` | Prints help topics |
| `help <topic>` | Prints help for topic |
| `help <command>` | Prints help for command |

- List of presented commands is incomplete

# Overview

Introduction

Getting started

Types, arithmetic operators

Control flow

Functions and program

Pointers and arrays

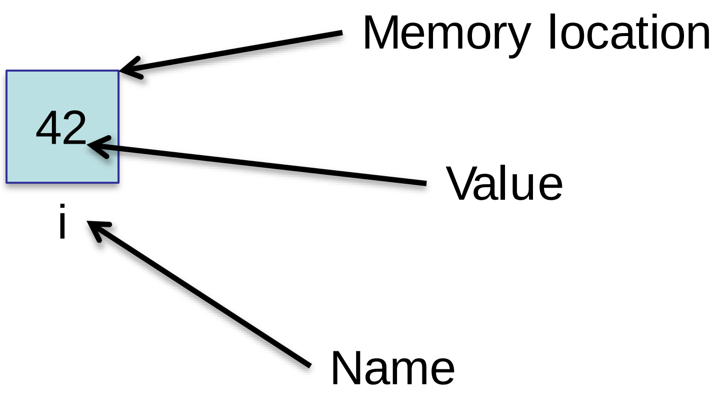
Structures

Input and output

# What is a variable?

Variables are names for memory locations to store data

Usually, we can refer to the variable by its name



# Variable declaration

Variables must be declared before they are used

The declaration tells

- How much memory is needed to store the variable
- How to interpret the bit-pattern
- Which instructions to use.
  - E.g. Integer and floating point arithmetic have different machine codes

A declaration consists of

*type variable_name*;

In C, all statements end with a semicolon.

E.g.

```
int my_int;
```

# Variable names

Consist of the characters

- A-Z
- a-z
- 0-9
- _ (underscore)

Must not start with a number

| Name | Valid? |
| --- | --- |
| a | yes |
| _ | yes |
| K99 | yes |
| 4bar | no |
| my-var | no |
| My_vAR | yes |
| _5Ghf34__ | yes |

# Base types

| Type | Description |
|---|---|
| `int` | Integer number, size depends on the host system, typically 32-bit |
| `char` | A single byte, holds one character |
| `float` | Floating point number, 32-bit |
| `double` | Double-precision floating point number, 64-bit |
| **Modifiers** | **Description** |
| `unsigned` | Always positive or zero. Applies to `int` and `char` |
| `long` | Applies to `int`. At least as long as `int`. |
| `short` | Applies to `int`. At least 16-bit. At most as long as `int`. |

- If modifiers are used, the `int` can be omitted
- E.g. `short my_var;` instead of `short int my_var;`
- C has no in-built string type but uses arrays of `char`

# Initialization and assignment of variables

Uninitialized variables have an undefined value

The assignment operator is ‘=‘

The value on the right is assigned to the variable on the left

```
int a, b;          // Values of a and b are undefined
b = 5;             // b has now the value 5
a = b;             // the value of b is copied to a
b = b + 4;         // b has now the value 9
int c = a + b;     // c has the value 14
```

# Constants

| Example | Type |
|---|---|
| 3343 | `int` |
| 3L | `long` |
| ‘s’ | `char` |
| 3.0f | `float` |
| 27.0 | `double` |
| “some string” | A string |

# Special characters

| Character | Description |
|---|---|
| \n | newline |
| \t | horizontal tab |
| \v | vertical tab |
| \\ | backslash |
| \b | backspace |
| \’ | single quote |
| \” | double quote |

The backslash escape a character

Used to represent “unprintable” characters

# Arithmetic operators

| Operator | Description |
|---|---|
| `A + B` | addition |
| `A - B` | subtraction |
| `A * B` | multiplication |
| `A / B` | division |
| `A % B` | modulo, no floating point numbers |
| `A += B` | Equals `A = A + B` |
| `A -= B` | Equals `A = A - B` |
| `A *= B` | Equals `A = A * B` |
| `A /= B` | Equals `A = A / B` |
| `A++` | Equals `A += 1` |
| `A--` | Equals `A -= 1` |

# Precedence

| Precedence | Operator |
|---|---|
| Highest | () |
| | ++, -- |
| | *, / |
| | +, - |
| Lowest | =, +=, -=, *=, /= |

Example

```
a = b + c * (d + ++e);
```

1. Increment e
2. Add d+e
3. Multiply result with c
4. Add result to b
5. Assign result to a

In general

- Follow mathematical rules
- Unary operators have high precedence
- Calculations first, assignments last
- You can force precedence with brackets
- If unsure, use brackets

# Notes for arithmetic with char variables

For C, `char` is just an integer with one byte

- Characters have a numeric value
- Arithmetic with `char` variables is integer arithmetic

```
char c = 'a';
c += 3;          // Now c has the value 'd'
c = 'z';
c = c + c + c;   // Now c has the value 'n'
```

# Expressions with mixed types

If different types appear in one operation, the compiler automatically converts one value before the operation

- It is only for the operation, the variable type does not change
- The shorter type is converted to the longer type
- Integer types are converted to floating point types
- With assignment operators, the value is converted to the target type
  - If this violates one of the other rules some compiler issue warnings
- If automatic conversion is impossible, the compiler reports an error

```
int a;
long b;
a = b;  // b to int
```

```
int a;
long b;
a + b;  // a to long
```

```
int a;
double b;
a + b;  // a to double
```

# Caveats when using mixed types

```
int a = 5, b = 2;
double d = a / b; // d has now the value 2.0
```

First `a / b` is executed:

- `a` and `b` are both integers
- No conversion necessary
- Uses integer arithmetic and result is an integer with the value 2

Then the result is assigned to `d`:

- `d` is a double
- the integer value 2 is converted to a double

# Explicit type casts

A type cast converts the value of a variable to a specific type before the operation

Write the new type in round brackets before the variable

- Casts have a higher precedence than arithmetic operators

```
int a = 5, b = 2;
double d = (double) a / (double) b;
// d has now the value 2.5
```

# Overview

Introduction

Getting started

Types, arithmetic operators

Control flow

Functions and program

Pointers and arrays

Structures

Input and output

# Blocks

Curly braces group statements into a compound statement, or block.

- A block is syntactically equivalent to a single statement
- Everywhere, were you can put a statement, you can also put a block

```
{
  int a;
  a++;
  printf("something");
}
```

No semicolon after the right brace that ends a block.

# The if statement

```
if (expression)
   statement;
else
   statement;
```

The else branch is optional and can be omitted

Example:

```
if (n > 3) n=5;
else {
   a = 4;
   b = 5;
}
```

# Boolean values in C

C has no boolean type

Any non-zero value is counted as true

Zero is counted as false

# Logical operators

| Operator | Description |
|---|---|
| A == B | True if A and B are equal |
| A != B | True if A and B are unequal |
| A > B | True if A is greater than B |
| A < B | True if A is smaller than B |
| A >= B | True if A is greater or equal than B |
| A <= B | True if A is smaller or equal than B |
| !A | True if A is false |
| A && B | True if A and B are true |
| A \|\| B | True if A or B are true |

# Precedence

| Precedence | Operators |
| --- | --- |
| high | brackets |
| | ! |
| | arithmetic operators |
| | ==, !=, >, <, <=, >= |
| | \|\|, && |
| low | =, +=, -=, *=, /= |

# Caveats with the comparison operator

This is valid C-code:

```
if ( a = b )
{
 printf( "b is non-zero!\n" );
}
```

a = b is an assignment. Thus, we assign the value of b to a

The result of an assignment is the assigned value. Thus, this is true if b is non-zero

Do not mistake the assignment operator '=' if you want the comparison operator '=='

## Nested ifs (1)

The `else` belongs to the innermost `if` that has no `else`, yet

```
if (n > 0)
   if (a < 100)
      n = a;
   else
      a = n;
```

# Nested ifs (2)

Force `else` to the outer `if` with braces

```
if (n > 0)
{
   if (a < 100)
      n = a;
}
else
   a = n;
```

## While loops

```
while( expression )
  statement;
```

Executes a statement until expression becomes false

Evaluates expression before first iteration

```
do
  statement;
while( expression );
```

Evaluates expression after first iteration

Executes statement at least once

# While-loop example

Print the numbers 0 to 99 to the screen

```
int i = 0;
while( i < 100 )
{
  printf( "%d\n", i );
  i++;
}
```

# For loops (1)

General format:

```
for( expr1; expr2; expr3 )
   statement;
```

`expr1` is executed at the beginning of the loop

`expr2` is executed at the beginning of every iteration

- If it is false, the loop ends

`expr3` is executed at the end of every iteration

# For loops (2)

General format:

```
for( expr1; expr2; expr3 )
   statement;
```

It is possible to omit any of the expressions

- The semicolon must stay

If `expr2` is omitted, the condition is always true

- it becomes an infinite loop

```
for( ;; ) //infinite loop
```

# For loops (3)

Usual use case:

```
int i;
for( i=0; i < 100; i++ )
  printf( "%d\n", i );
```

# Break statement

```
break;
```

Terminates the innermost loop or switch statement

Execution resumes after the loop or switch statement

```
while( 1 )
{
   n++;
   if ( n > 5 ) break;
}
```

# Continue statement

```
continue;
```

Terminates the current iteration of the innermost loop

Execution resumes at the beginning of the next iteration

```
for (i=0; i<100; i++)
{
  if ( i == 57 ) continue;
  printf( “%d\n”, i );
}
```

Print the numbers 0 to 99, but not 57

# The switch statement

```
switch( expression )
{
   case const-expr: statements
   case const-expr: statements
   default: statements
}
```

`const-expr` must be constant integer values

Execution falls through

- Executes also all following cases
- Usually exit the switch with a break

# Switch example

```
int digit=0, whitespace=0, other=0, c;
while( (c = getchar()) != EOF ) {
  switch ( c ) {
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
   digit ++;
   break;
  case ' ': case '\t': case '\n':
   whitespace ++;
   break;
  default: other++;
}}
```

`getchar()` reads one character from the keyboard

# The ? operator

*`expr1 ? expr2 : expr3;`*

First, evaluates `expr1`

- If `expr1` is true, the value of the expression is `expr2`
- If `expr1` is false, the value of the expression is `expr3`
- The branch that is not assigned is not evaluated

Precedence is higher than assignments but lower than logical and arithmetic operators

```
max = a > b ? a : b;
```

Is equivalent to

```
if ( a > b ) max = a;
else max = b;
```

# Overview

Introduction

Getting started

Types, arithmetic operators

Control flow

Functions and program

Pointers and arrays

Structures

Input and output

# Functions (1)

Functions are the building blocks of a program

Allow to reuse code for reappearing algorithms

Decomposing a program into multiple functions

- Provides code structure
- Increases maintainability
- Avoid 100+ line functions, if possible

Programming is the art of decomposing a large task into appropriate building blocks.

# Functions (2)

```
return_type function_name ( argument_list )
{
 declarations and statements
}
```

This is the header of the function

This is the body of the function

Function names follow the same rules as variable names

The argument list and body may be empty:

```
return_type function_name()
{}
```

# Return statement

The return type can be every C-type

```
return expression;
```

If the execution encounters a return statement,

- It evaluates the expression
- Converts the result to the return type of the function
- Exits the function
- Returns the results of the expression

If the function reaches its end without encountering a return statement the return value is undefined

- Some compilers issue warnings

# Return type void

If a function has return type void it returns no value.

It still can have return statements

- but they must have no expression

```
void foo()
{
 return;
}
```

# Parameter lists

Comma-separated list of variable definitions

The variables are initialized with the value passed on function call

```
int pow( int base, int exponent )
{
  int i, result = 1;
  for( i=0; i < exponent; i++ ) result *= base;
  return result;
}

int my_var = pow( 4, 6 );
```

## Calling a function

The parameter values are assigned to the parameters in the function definition in the order of their appearance

The return value can be ignored

```
int foo()
{
  return 2;
}
```

```
float square(float v)
{
  return v*v;
}
```

```
int a, b = foo();

a = 2 + pow( b, b );

square( foo() ) + pow( 2*b+1, 3 ) / 4;
```

# The main function

Every program needs a function named `main`

```
int main( int argc, char** argv )
```

The program execution starts with main

The return value of `main` is the return value of the program

- Usually, returning a non-zero value means the program terminated with an error

`argc` contains the number of program arguments

`argv` contains the program arguments

- We will explain later what the ** means

You can leave the parameter list empty: `int main()`

# Function declaration (1)

You can call only functions that are declared before

- One possibility is to write the functions that are calling a function after the function that is called
- Not possible if mutual calls or a cycle of calls exist

```
int foo( int n )
{
  if (n > 0)
    return bar( n-1 );
  return 0;
}
```

```
int bar( int n )
{
  if (n < 0)
    return foo( n+1 );
  return n;
}
```

# Function declaration (2)

Solution: Add a function declaration in front of the implementation

- Consists of the function header followed by a semicolon

```
int bar( int n );

int foo( int n ) {
  if (n > 0)
    return bar( n-1 );
  return 0;
}
int bar( int n ) {
  if (n < 0)
    return foo( n+1 );
  return n;
}
```

# Scope

The scope of a symbol defines where it is accessible

- A symbol is everything that has a name
- variables, functions, types, …

A symbol needs to be declared before it accessed

A scope can be restricted further

- It is good style to restrict scope
- Increases maintainability and readability
- Avoids side-effects
- Avoids name clashes

# Global variables

A variable is global if it is declared outside of any function

Variable is constructed at program start

Variable is deleted at program finalization

The variable can be accessed throughout the whole program

```
int global_var = 5;
int main() {
  printf( “global_var = %d\n”, global_var );
}
```

# Local variables

Local variables are declared within a code block.

- Statements enclosed in curly brackets are code blocks
- Function bodies are code blocks

```
{
 int i;
 // do something
}
```

Variable is constructed at entry of the block

Variable is deleted at exit of the block

Access to the variable is only valid within the block

Parameter variables are local variables of the function body

# Static variables

Local variables do not keep their value between visits

If a variable is declared `static`

- It keeps its value between calls
- Only the first visit executes the initialization
- The initialization value must be a constant

```
int foo() {
  static int counter = 0;
  return ++counter;
}
```

Returns the number of times the function was called

# Variables with same name

If on same scope, it results in a compiler error

A variable in an inner scope overshadows variables in an outer scope

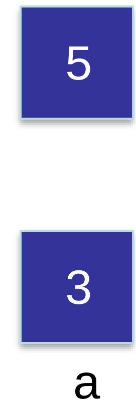
# Scope example (1)

```
int a = 5;
int main() {
  printf( "a = %d\n", a );
  int a = 3;
  printf( "a = %d\n", a );
  {
    int a = 1;
    printf( "a = %d\n", a );
  }
 printf( "a = %d\n", a );
}
```


Global variable is constructed
The `printf` knows only the global a and prints `a = 5`

# Scope example (2)

```
int a = 5;
int main() {
  printf( "a = %d\n", a );
  int a = 3;
  printf( "a = %d\n", a );
  {
    int a = 1;
    printf( "a = %d\n", a );
  }
 printf( "a = %d\n", a );
}
```




A new local variable `a` is created

The global variable is overshadowed

`a` refers to the local variable

The program prints `a = 3`

# Scope example (3)

```
int a = 5;
int main() {
  printf( “a = %d\n”, a );
  int a = 3;
  printf( “a = %d\n”, a );
  {
    int a = 1;
    printf( “a = %d\n”, a );
  }
 printf( “a = %d\n”, a );
}
```

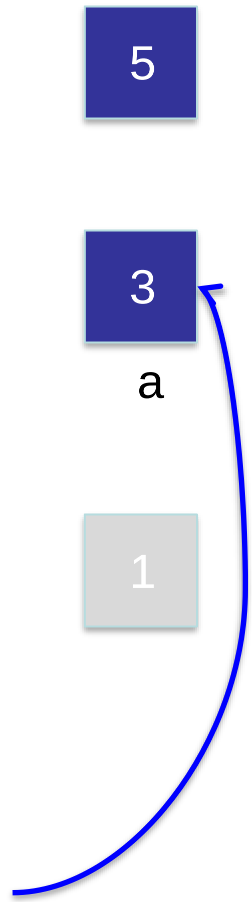
A variable `a` is created
`a` refers to the innermost variable
The program prints `a = 1`

# Scope example (4)

```
int a = 5;
int main() {
  printf( “a = %d\n”, a );
  int a = 3;
  printf( “a = %d\n”, a );
  {
    int a = 1;
    printf( “a = %d\n”, a );
  }
 printf( “a = %d\n”, a );
}
```

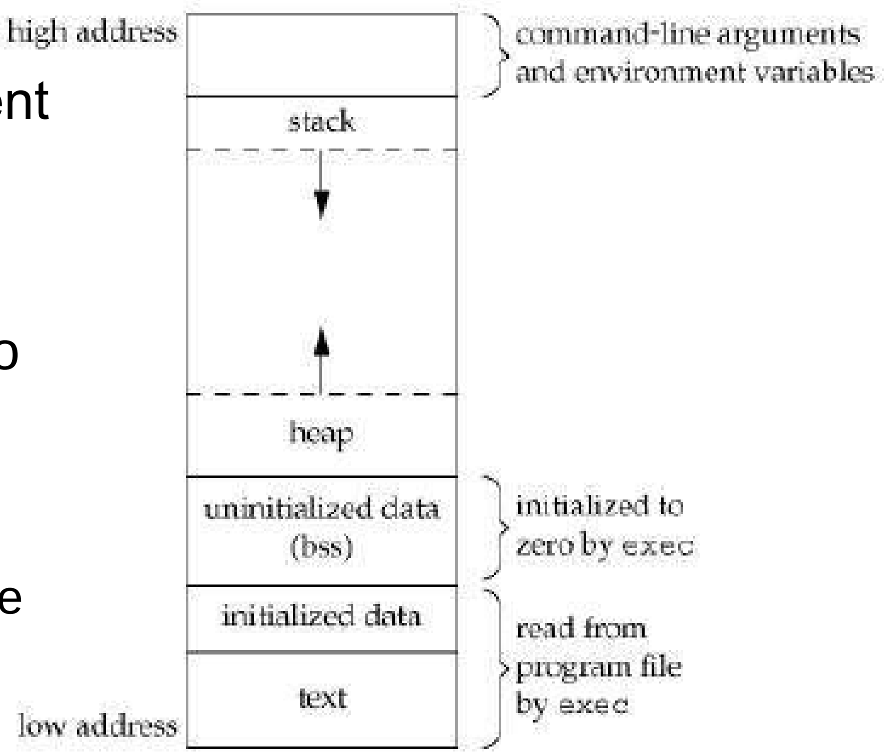
With the end of the block the the variable exists no more

The innermost variable that is still visible is



# Program organization in memory

- When a program is loaded into memory, it's organized into different segments. One of the segment is **DATA segment.** The Data segment is further sub-divided into two parts:
  - **Initialized data segment:** All the global, static and constant data are stored here.
  - **Uninitialized data segment(BSS):** All the uninitialized data are stored in this segment.




Source: https://stackoverflow.com/

# Source files organization

C source files have the extension `.c`

Larger programs consist of multiple source files

If you call functions from other source files

- The functions are not declared before you used them
- The compiler complains about unknown functions

Solution

- Put a declaration of the functions at the top of every file
- Means you have to copy the function declaration multiple times

# Header files

Better

- Put the declarations in a header file
- Tell the compiler to include the header file
  - The C-preprocessor copies the content of the included file at the position of the include directive
  - Header files have the extension `.h`
  - Guarantees that all sources have the same declarations

`#include <filename>`

- Searches only in the include path

`#include "filename"`

- Searches in the include path and the source path

# Standard header

The C standard defines a library of standard functions

The linker automatically links the standard library

The functions are declared in a set of standard headers

To use the standard functions you need to include the header

E.g. `printf` is declared in `stdio.h`

# Include path

The compiler knows path to standard headers

Add search directories for header file to compiler with `-I` *`dir`*

```
gcc -I/path/to/my/header hello.c -o hello
```

# Global variables in other files

When two source files contain a declaration of the same global variable

- The compiler will create two distinct global variables with the same name
- The linker complains about two symbols with the same name

The `extern` keyword tells the compiler

- The variable is defined in another source file
- Do not create a new variable
- Access the variable from the other source file

Thus, for global variables that are used in multiple source files

- One source file declares the variable without external
- Put an extern variable declaration in a header

# Global variable example

my_global_vars.c

```
int global_var;
```

my_global_vars.h

```
extern int global_var;
```

Any other source file

```
#include “my_gloabl_vars.h”
```

# Static functions

A static function can not be called from other source files

- Avoids name clashes between helper functions in different source files
- Ensures that the interface is not hijacked by someone else

```
static int foo()
{
}
```

# Overview

Introduction

Getting started

Types, arithmetic operators

Control flow

Functions and program

Pointers and arrays

Structures

Input and output

# Memory organization

A typical computer has one long array of memory cells

They can be manipulated individually or in contiguous groups

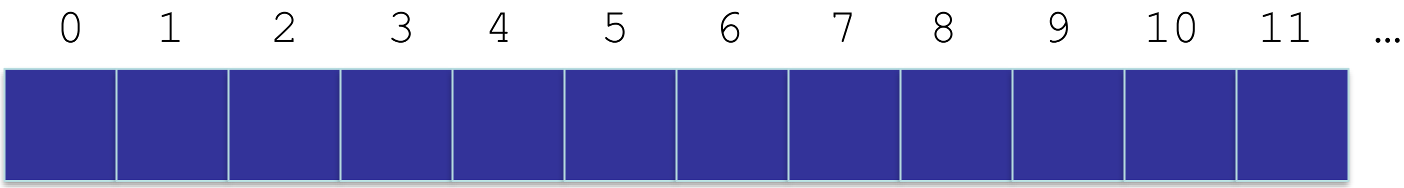
Every cell can store one byte

The cells are enumerated

The number of a cell is its address

If a program wants to access a memory location it needs the address

The compiler/linker translates variable names into addresses



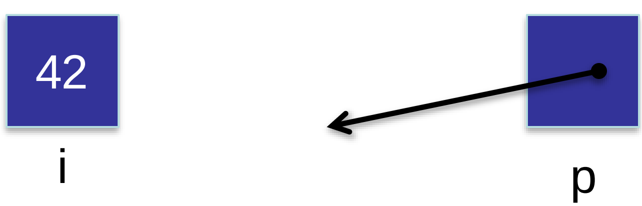
# Pointer (1)

With the reference operator &, we can get the address of a variable:

```c
int i = 42;
printf( "%p", &i}; // Prints address of i
```

Pointers are variables that store memory addresses

Pointers can only point to variables of a specific type



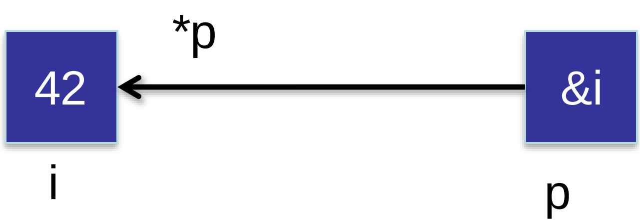

```c
int* p;
```

Creates a variable to store an address to an integer variable

# Pointer (2)

We need to assign an address of another variable.

```
p = &i;
```

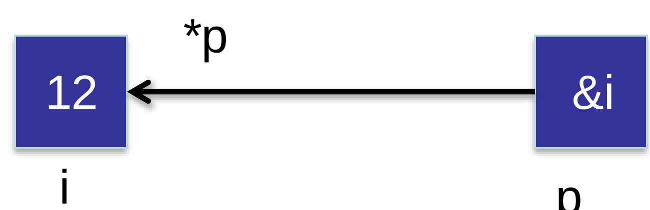



We can access the value to which p points with the dereference operator *

```
printf( "%d", *p ); // Prints 42
```

If we assign a value to `*p`, we implicitly alter `i`

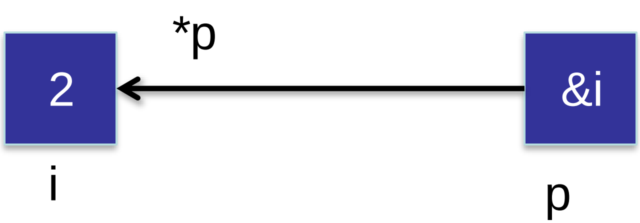
```
*p = 12;
printf( "%d", i ); // Prints 12
```



# Pointer (3)

If we pass a pointer as parameter to the function, we copy an address

```
void func( int* p ) {
  *p = 2; // assigns a new value to i
}
func( &i );
```

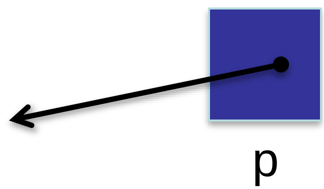



Avoid side-effects as much as possible

- Side-effects may lead to unexpected program behavior

# Pointer (4)

Caution: The creation of a pointer does not allocate memory for a value




```
int* p;
*p = 42;
```

Results in errors

`p` stores an arbitrary address

Writing something means writing to an arbitrary memory location

Leads to hard-to-find bugs or segmentation faults

# Null pointer

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

`NULL` is a pointer to the address 0

It is defined in `stddef.h`

It marks an invalid pointer

Allows checks whether a pointer is valid

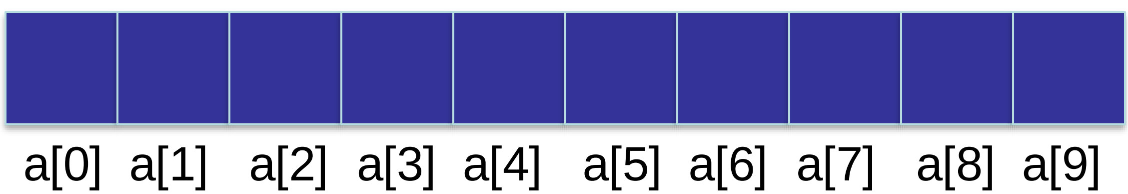
```c
if ( p == NULL ) { … }
```

# Arrays

```
int a[10]
```

Defines an array of 10 `int` variables

The elements of the array have the names `a[0]` … `a[9]`



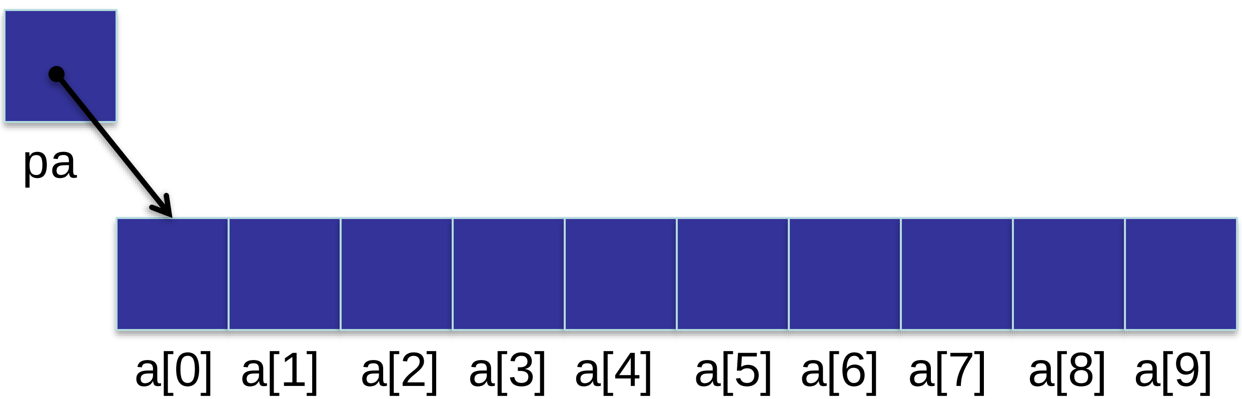

Initialization with a constant

```
int b[3] = {0, 1, 2};
```

# Address arithmetic (1)

```
int* pa = &a[0];
```

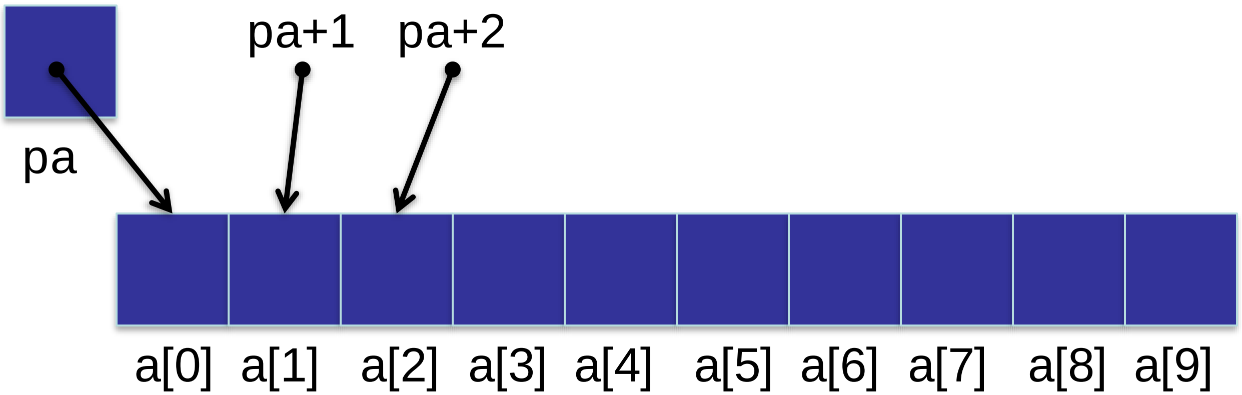
Creates a pointer to the first element of the array



# Address arithmetic (2)

```c
int* pa = &a[0];
```

Creates a pointer to the first element of the array




`*pa` accesses the first element

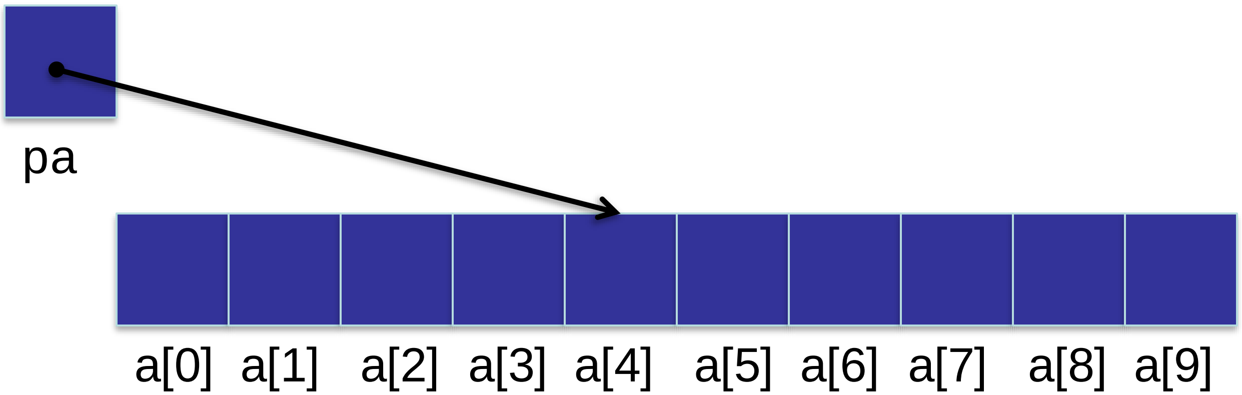
`*(pa + 1)` accesses `a[1]`

`*(pa + i)` accesses `a[i]`

# Address arithmetic (3)

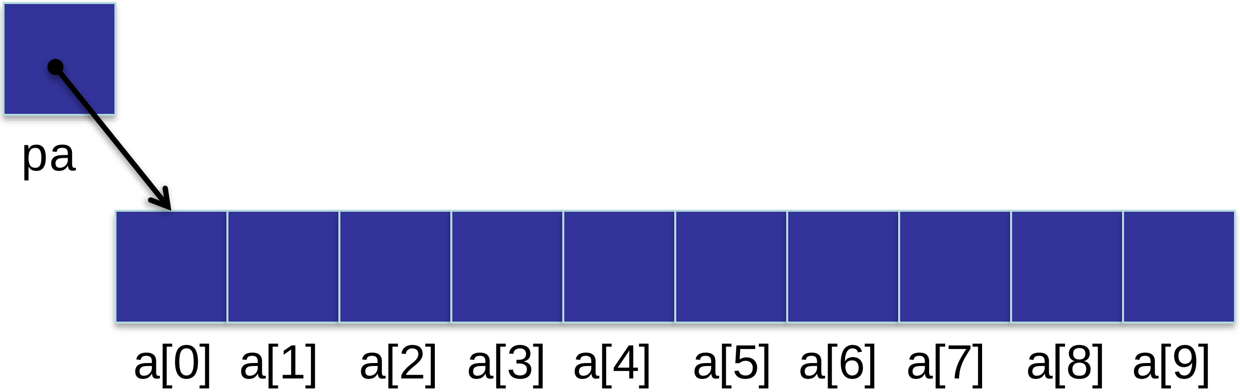
Can use arithmetic operators on `pa`

```
pa += 4;   // Now points to a[4]
```



# Address arithmetic (4)

```
int* pa = a;
// Short form for: int* pa = &a[0];
```




It is possible to use element access with pointers

```
int i = pa[3];  // Assigns a[3] to i
```

# Passing arrays as parameters

The address of the first element is passed

```
#include <stdio.h>
int foo( int* a ) {
  printf( "%p\n", a );
}
int main() {
  int a[10];
  printf( "%p\n", a );
  foo( a );
}
```

Prints 2 times the same address

# Differences between arrays and pointers

```
int a[10]
int* pa = a;
pa++;
```

is legal

But the following is illegal

```
a = pa;
a++;
```

No assignments to and address arithmetic on arrays!

# Pointer type void

```
void* a;
```

Creates a pointer variable with unspecified target type

Can assign any pointer type to a `void*`

No address arithmetic possible on `void*`

- Do not know the size of the elements

Need to cast the pointer, before one can use it

```
double d[10];
a = &d;
* (double*)a = 3.2;
```

Dereference a `double*`

Cast `a` to a `double*`

# Motivation – dynamic allocation (1)

The size of arrays is determined at compile time

- They have a constant size

Sometimes the size is only known at runtime

# Motivation – dynamic allocation (2)

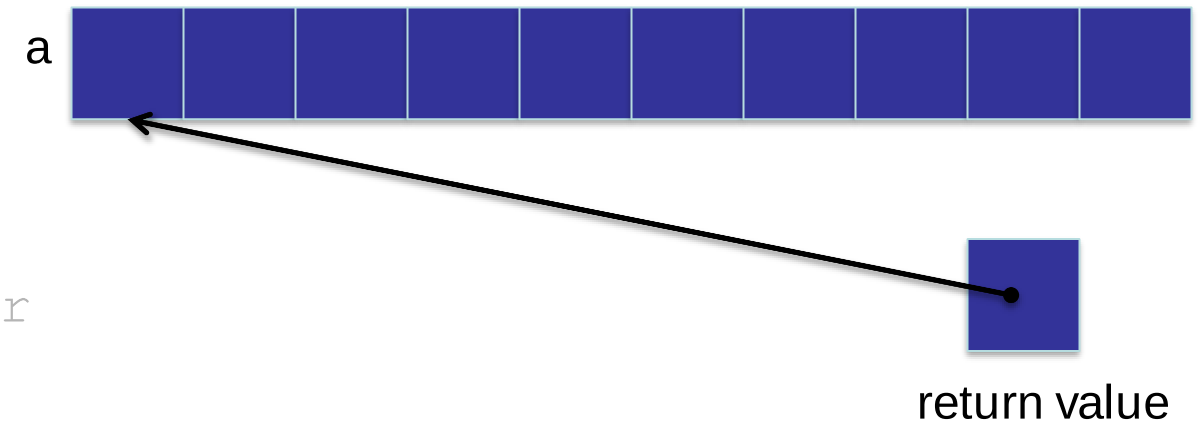
Want to return an array from a function

```
int* foo() {
   int a[10];
   return a;
}

int* b = foo();
b[0] = 1; // Error
```

# Motivation – dynamic allocation (3)

Want to return an array from a function

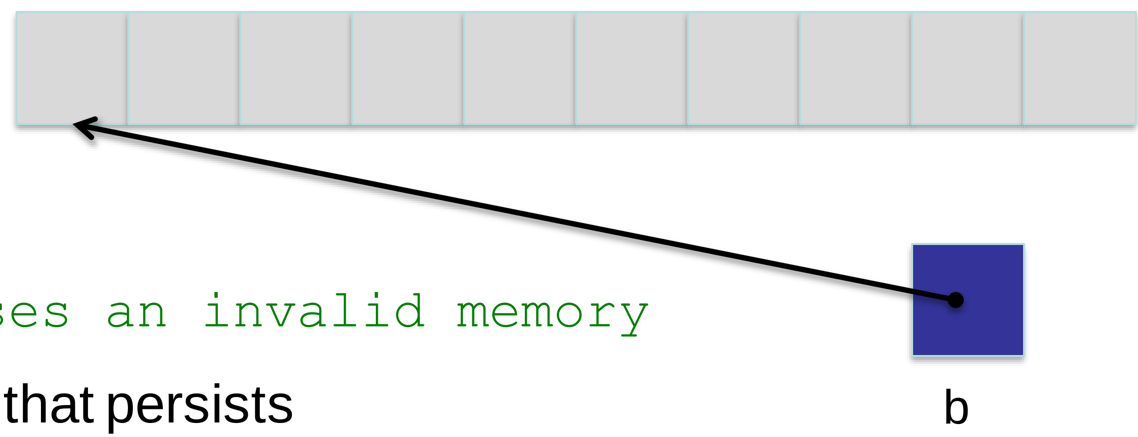
```
int* foo() {
   int a[10];         // Creates an array
   return a;          // Returns the address
}

int* b = foo();
b[0] = 1; // Error
```



# Motivation – dynamic allocation (4)

Want to return an array from a function

```
int* foo() {
  int a[10];
  return a;
} // Deletes a

int* b = foo();
b[0] = 1; // Accesses an invalid memory
```




Want to allocate memory that persists

# Dynamic memory allocation

`void* malloc( size_t size )`

- Allocates a memory block with the specified number of bytes
- Returns a pointer to the start of the memory block
- Is declared in `stdlib.h`
- The content of the allocated memory is undefined

`sizeof( type )`

- Is an operator that returns the number of bytes for `type`

# Free allocated memory

Memory allocated dynamically is not automatically freed

- The programmer must take care to free the memory
- When a program frequently allocates memory but never frees it, it may run out of memory
- Memory management is a source of many errors in C

```
void free( void* p );
```

- Is declared in `stdlib.h`
- Frees the memory of the passed pointer

# Allocation example

Allocates an array of 100 `int`

Initializes it with the number 0 to 99

```
#include <stdlib.h>
int main()
{
   int *p = malloc( sizeof( int ) * 100 );
   int i;
   for( i=0; i < 100; i++ ) p[i] = i;
   free( p );
}
```

# Qualifier const

The `const`-qualifier inhibits modifications to a variable

Assignment only possible at initialization

```
const double pi = 3.14159265359;
pi = 4; // Compiler error
```

# Constant parameters

When passing pointers to functions

- Modifications in the function affects the caller
- Side effects are bad

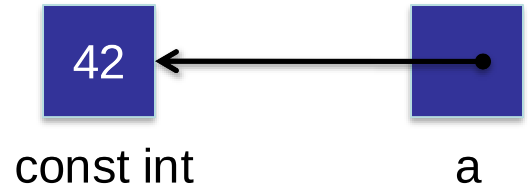
Make parameter `const`

- Ensures that no side-effects happen on this variable
- Making pointer parameters `const` is part of a clean style

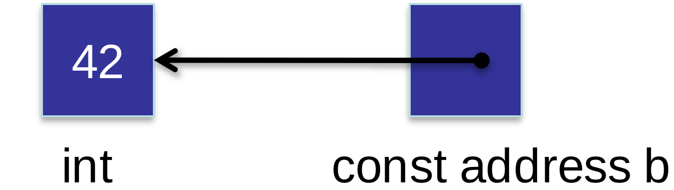
```
void foo( const int*  a )
{
  *a = 1; // Compiler error
}
```

# `const int*` and `int* const`

```
int i;
const int* a;
a = &i; // allowed
*a = 0; // error
```




```
int* const b = &i; // initialization
b = NULL; // error
*b = 0; // allowed
```



# Strings in C

C has no in-build data type for strings

C-Strings are `char`-arrays

The end of the string is marked with the `'\0'`-character

- For a string with n characters you need an array with n+1 elements

Initialization with a string constant

```
char* str = "constant C-string";
```

# String operations in C

The C standard library contains a set of string handling functions

Most of them are declared in `string.h`

| Function | Description |
|---|---|
| `strlen` | Returns the length of a string |
| `strdup` | Duplicates a string |
| `strcpy` | Copies the content of a string into another string |
| `strcmp` | Compares the content of two strings |
| `strcat` | Concatenate two strings |

# Example – string duplication

```
char* strdup( char* source ) {
  int length = strlen( source ) + 1;
  char* c, str = malloc( length * sizeof( char ) );
  for ( c = str; *source != '\0'; source++){
    *c = *source;
    c++;
  }
  *c = '\0';
  return str;
}
```

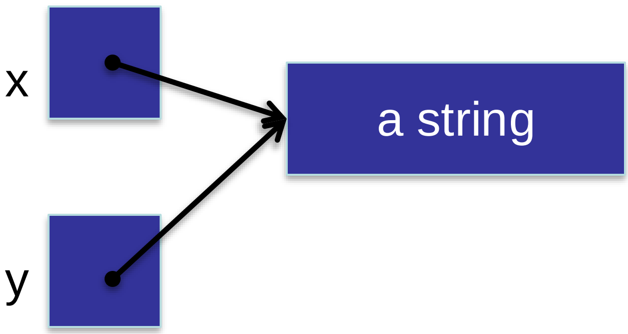
A common source of errors is to forget to terminate a string with the ‘\0’-character

# String pitfalls

```
char* x = "a string";
char* y = x;
```

Does not copy the string

Use `strdup()` for copying strings




```
x == y
```

Does not compare the string content

Is only true if both point to the same memory address

Use `strcmp()` to compare the string content

# Pointer arrays – pointers to pointers

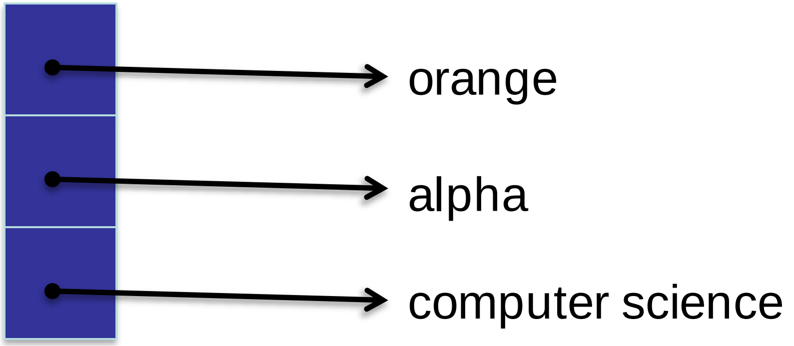
You can create arrays of pointers like for any other variable

Both of the following create an array of 3 `char*`

```
char* a[3];

char** b = malloc( sizeof( char* ) * 3 );
```

Access the 3rd character of the first word

```
char c = a[0][2];

c = b[0][2];
```



# Multidimensional arrays

```
int a[4][4];
```

Defines a two dimensional array

Initialization with a constant

```
int a[2][2] = { {1, 2}, {3, 4} };
```

# Pointers vs. multidimensional arrays

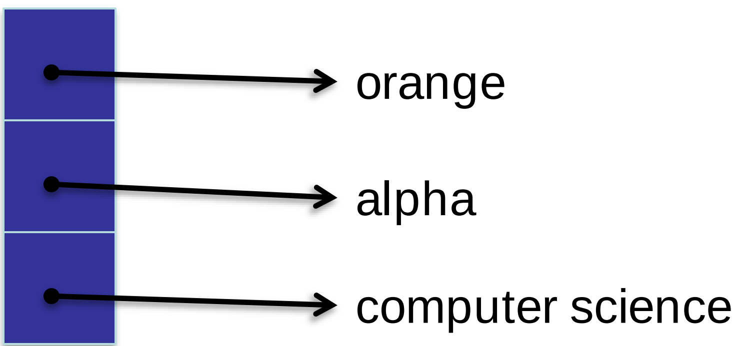
```
char** a = malloc( … );
```

Creates array of undefined pointers

- Need to allocate every element

Row can have a different length

Rows may not be adjacent in memory




```
char b[4][4];
```

Allocates 4x4 elements

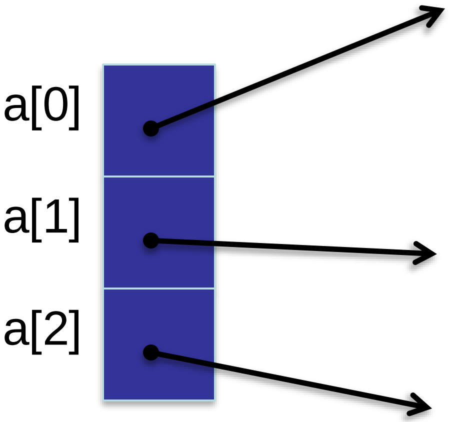
All rows have the same length

One contiguous memory block

| | | | |
|---|---|---|---|
| o | r | a | n |
| a | l | p | h |
| c | o | m | p |
| l | a | k | e |

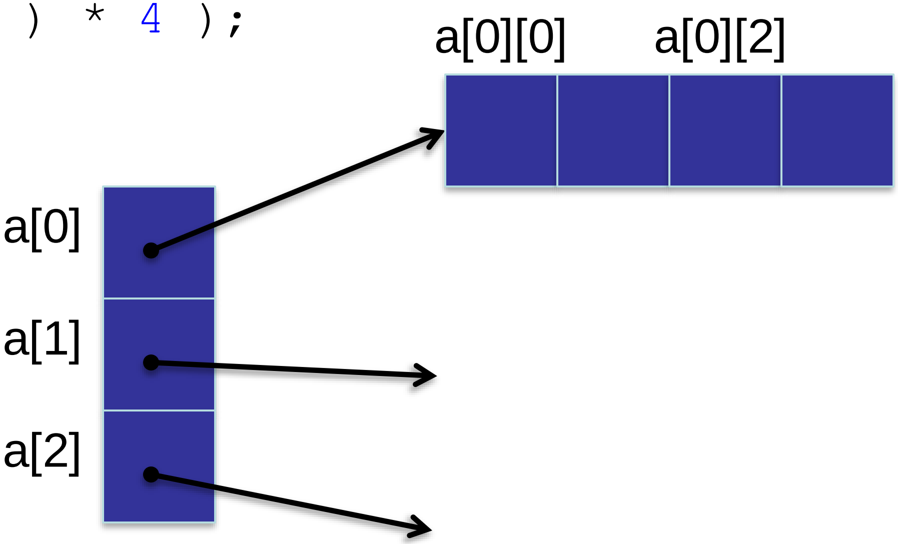
# Allocate memory for an array of pointers

```
#include <stdlib.h>
int main() {
  char* a[3];
  for( int i=0; i<3; i++ ){
    a[i] = malloc( sizeof( char ) * 4 );
  }
  //do something
  for (int i=0; i<3; i++) {
    free(a[i]);
  }
}
```



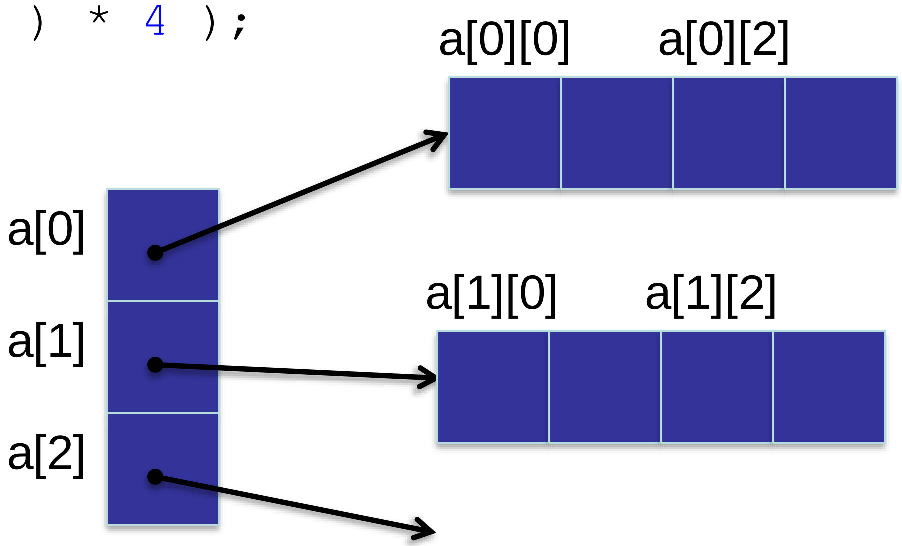
# Allocate memory for an array of pointers

```
#include <stdlib.h>
int main() {
  char* a[3];
  for( int i=0; i<3; i++ ){
    a[i] = malloc( sizeof( char ) * 4 );
  }
  //do something
  for (int i=0; i<3; i++) {
    free(a[i]);
  }
}
```



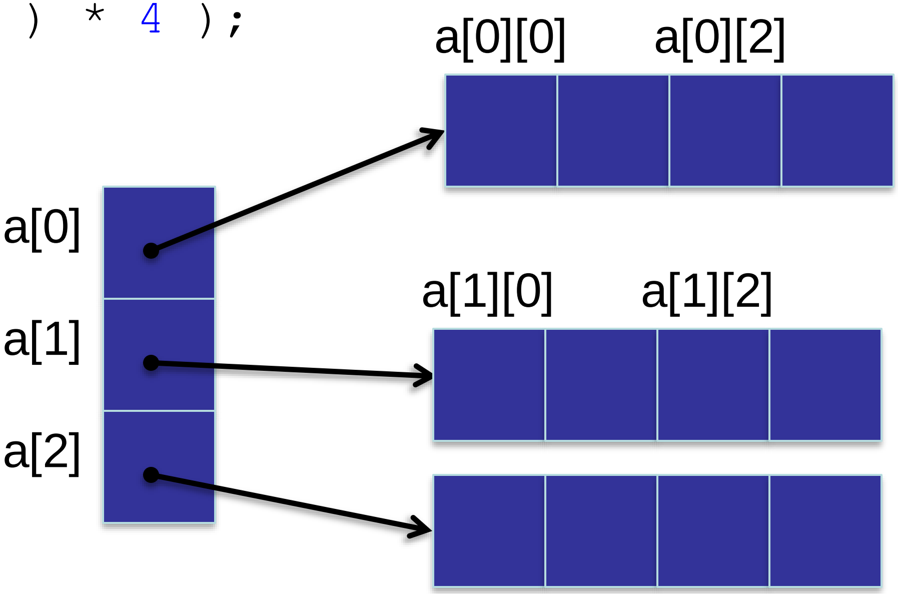
# Allocate memory for an array of pointers

```c
#include <stdlib.h>
int main() {
  char* a[3];
  for( int i=0; i<3; i++ ){
    a[i] = malloc( sizeof( char ) * 4 );
  }
  //do something
  for (int i=0; i<3; i++) {
    free(a[i]);
  }
}
```



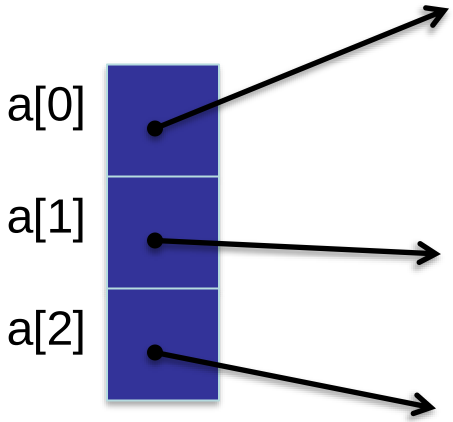
# Allocate memory for an array of pointers

```c
#include <stdlib.h>
int main() {
  char* a[3];
  for( int i=0; i<3; i++ ){
    a[i] = malloc( sizeof( char ) * 4 );
  }
  //do something
  for (int i=0; i<3; i++) {
    free(a[i]);
  }
}
```



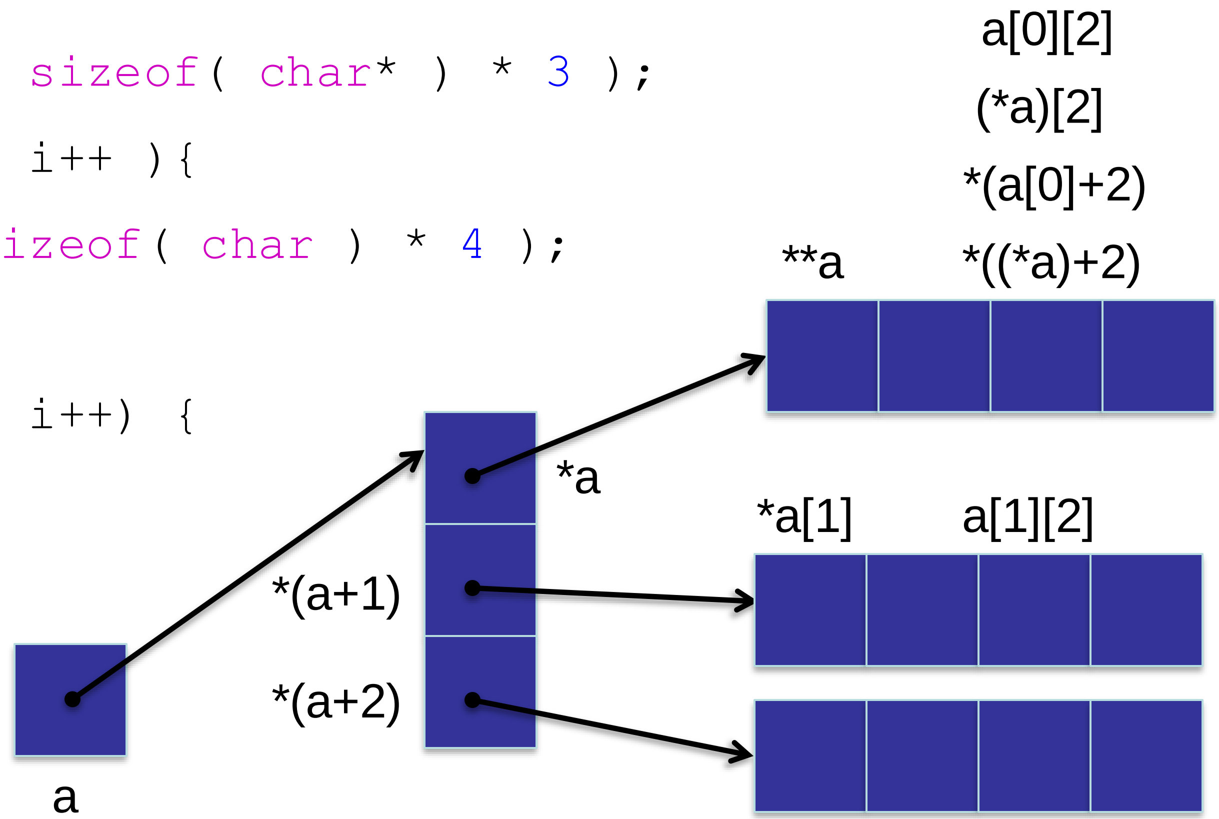
# Allocate memory for an array of pointers

```
#include <stdlib.h>
int main() {
  char* a[3];
  for( int i=0; i<3; i++ ){
    a[i] = malloc( sizeof( char ) * 4 );
  }
  //do something
  for (int i=0; i<3; i++) {
    free(a[i]);
  }
}
```



# Allocate memory for pointer to pointers

```c
#include <stdlib.h>
int main() {
  char** a = malloc( sizeof( char* ) * 3 );
  for( int i=0; i<3; i++ ){
    a[i] = malloc( sizeof( char ) * 4 );
  }
  for (int i=0; i<3; i++) {
    free(a[i]);
  }
  free( a );
}
```



# Command-line arguments

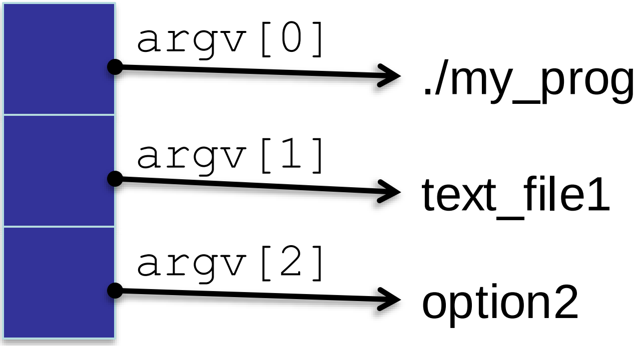
```
int main( int argc, char** argv )
```

Program arguments are passed as an array of `char*`

`argc` provides the number of arguments

`argv` contains the arguments

```
./my_prog text_file1 option2
```




Results in 3 arguments `(argc = 3)`

# Processing program arguments

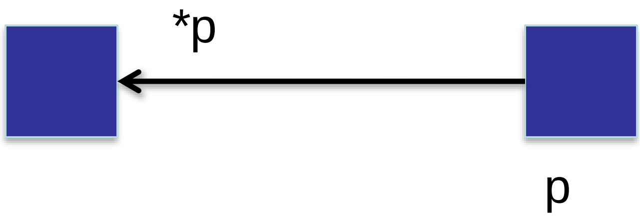
```
#include <string.h>
int main( int argc, char** argv )
{
  for( int i=1; i<argc; i++ )
  {
    if ( strcmp( argv[ i ], "option1" ) == 0 ) { ... }
    else if ( strcmp( argv[ i ], "option2" ) == 0 ) { ... }
    else if ( strcmp( argv[ i ], "option3" ) == 0 ) { ... }
    else { ... }
  }
}
```

# Memory leaks (1)

```
{
   int* p;                    // Creates a pointer
   p = malloc( sizeof( int ) );


}
```

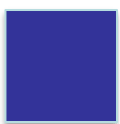
# Memory leaks (2)

```
{
   int* p;                 // Creates a pointer
   p = malloc( sizeof( int ) );
                           // Creates a new int variable
}
```



# Memory leaks (3)

```
{
   int* p;                // Creates a pointer
   p = malloc( sizeof( int ) );
                          // Creates a new int variable
}                         // Deletes the pointer
                          // but the variable still exists
```



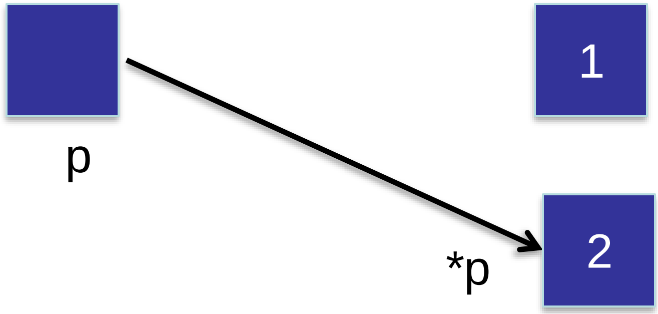
No chance to access or free this memory allocation!

Make sure that you delete all objects you allocated

# Memory leaks (4)

```
int *p = malloc( sizeof( int ));
```


```
p = malloc( sizeof( int ));
```




Memory allocation still exists

Not able to access it again

Delete old memory before allocating new memory

# Dangling pointers (1)

```
int* p;    // Create a pointer variable
{
  int i;   // Create an integer variable
  p = &i;  // Assign address of i to p
}
*p = 42;
```

# Dangling pointers (2)

```
int* p;    // Create a pointer variable
{
  int i;   // Create an integer variable
  p = &i;  // Assign address of i to p
}          // Delete i, p becomes a dangling pointer
*p = 42;   // Error: Accesses non-existing variable
```

# Overview

Introduction

Getting started

Types, arithmetic operators

Control flow

Functions and program

Pointers and arrays

Structures

Input and output

## Structures (1)

A structure is a collection of variables that belong together

```
struct name {
  member variable definitions
};
```

E.g.

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

# Structures (2)

A structure is a new type

Variable declaration of a `struct`

```
struct point a;
```

With initialization

```
struct point b = {1, 2};
```

x y

# Member access

The . operator accesses members of a structure

```
struct point p;
p.x = 1;
p.y = 2;
```

# Pointers and arrays of structures

```
struct point a[3];
```

```
struct point* b = malloc (sizeof(struct point) * 3);
```

Both create an array of 3 point structures

Access to members with a pointer with the `->` operator

```
b->x;
```

# The typedef keyword

```
typedef type newname;
```

Defines a new name for a type.

```
typedef int Integer;

typedef char* String;

Integer i = 1;

String s = “Hello”;
```

Often used for complex data types like structures

```
typedef struct point Point;

Point p = {1, 2};
```

# Type definitions for structures

Define `struct point` and define a new type name for it

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

Equivalent definition in one statement

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

Define type for an unnamed struct

```
typedef struct {
  int x;
  int y;
} Point;
```

# Assignment of structures

Assignments of structures copy the value of all members

```
struct point {
  int x;
  int y;
};
struct point a = {1, 2};
struct point b = a;
printf("x = %d, y = %d\n", b.x, b.y);
```

Output: `x = 1, y = 2`

# Incomplete types

In `example.h`:

```
typedef struct MyStruct
        MyStruct;
void foo(MyStruct *m);
MyStruct* create();
```

`example.c`

```
struct MyStruct {
  int i;
};
```

Implementation of `foo` and `create`

Other files that include `example.h` can use pointers of `MyStruct`

They cannot dereference the pointer

- Compiler does not know the layout `MyStruct`
- The type is incomplete

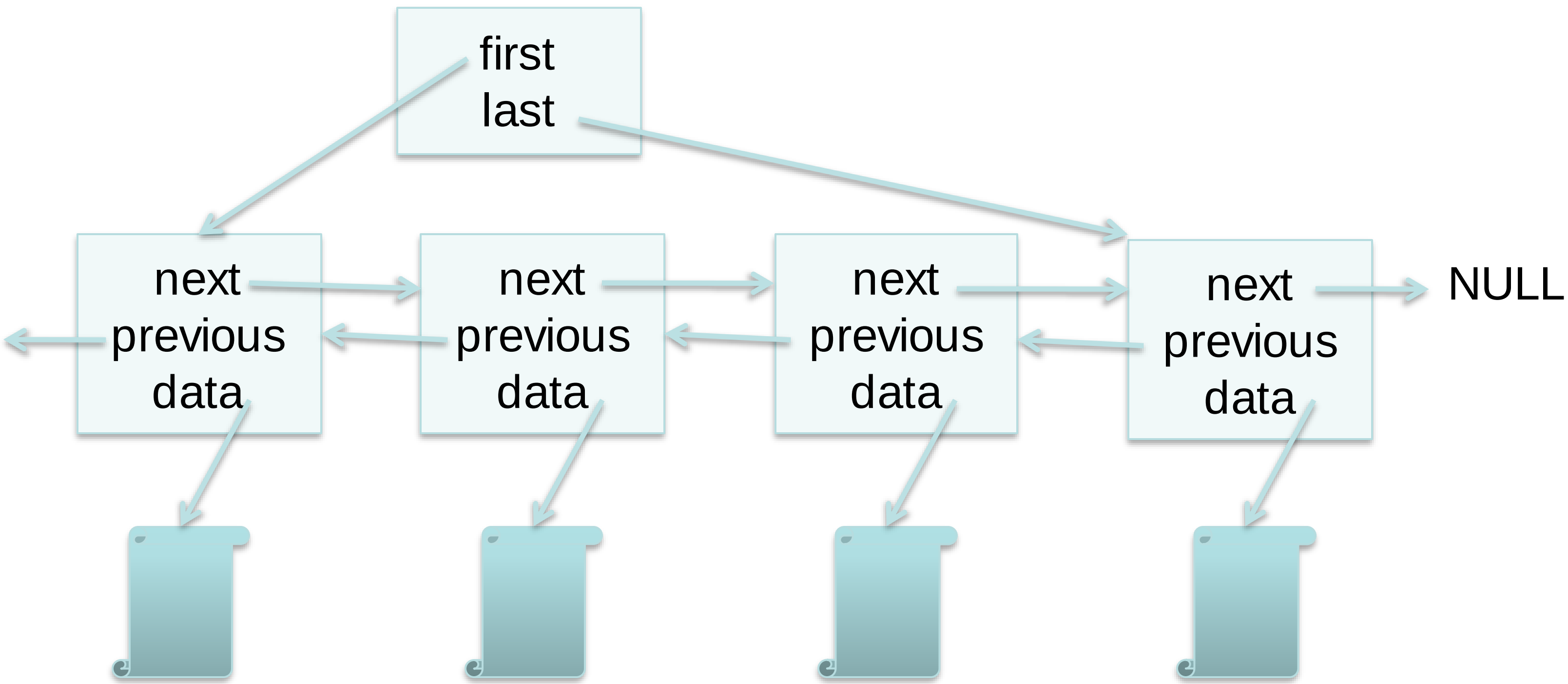
Need to access `MyStruct` through the functions in `example.h`

- Hides implementation details and exposes only interface

# Motivation – self-referential structures

In data structures like linked-lists or trees

Elements need to point to other elements



# Self-referential structures

Structures can have members that are pointers objects of its own type

```
struct LinkedList {
  struct LinkedList* next;
};
```

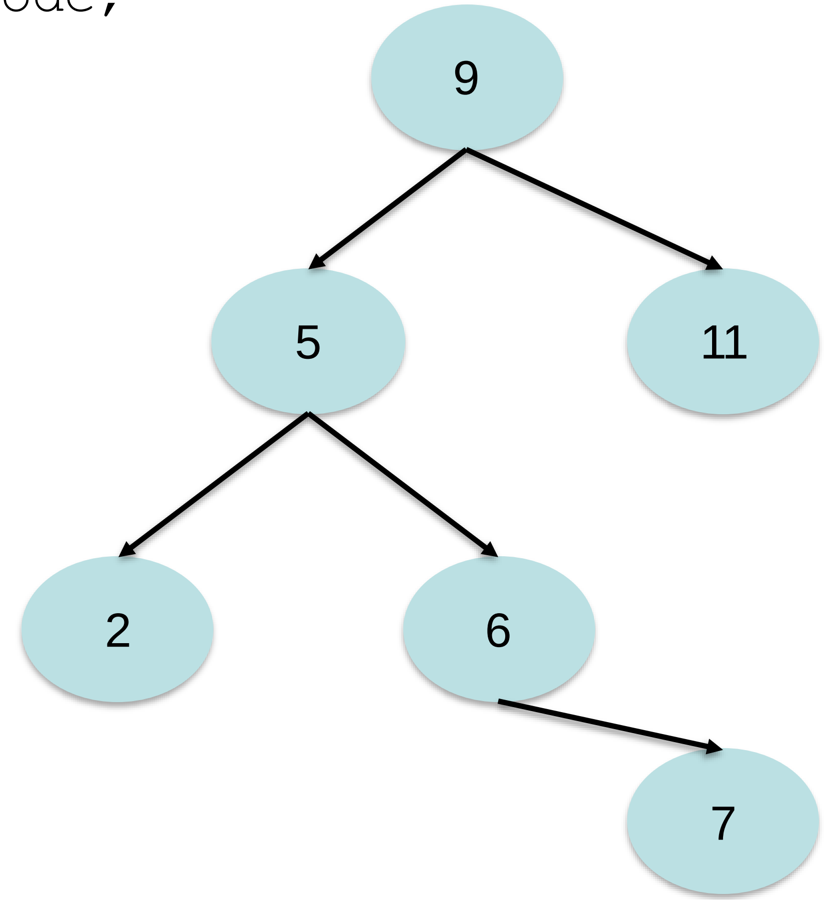
You cannot put a member in a `struct` that is of its own type

- Would lead to infinite recursion and require infinite memory

```
struct BrokenCode {
  struct BrokenCode a;
};
```

# Example – sorted binary tree: data type

```
typedef struct TreeNode TreeNode;

struct TreeNode {
  TreeNode* left;
  TreeNode* right;
  int value;
};

typedef struct {
  TreeNode* root;
} BinaryTree;
```



# Example – sorted binary tree: creation

```
BinaryTree* BinaryTree_create()
{
  BinaryTree* tree = malloc( sizeof( BinaryTree ) );
  tree->root = NULL;
  return tree;
}
```

# Example – sorted binary tree: insertion (1)

```
static void insert_node( TreeNode* parent, TreeNode* node );
void BinaryTree_insert( BinaryTree* tree, int value )
{
  TreeNode* new_node = malloc( sizeof( TreeNode ));
  new_node->value = value;
  new_node->right = NULL;
  new_node->left  = NULL;
  if (tree->root == NULL) {
    tree->root = new_node;
    return;
  }
  insert_node( tree->root, new_node );
}
```

# Example – sorted binary tree: insertion (2)

```
static void insert_node( TreeNode* parent, TreeNode* node )
{
  if( parent->value > node->value )
  {
    if( parent->left == NULL ) parent_left = node;
    else insert_node( parent->left, node );
  }
  else
  {
    if( parent->right == NULL ) parent_right = node;
    else insert_node( parent->right, node );
  }
}
```

# Example – sorted binary tree: deletion

We need to deallocate all `TreeNode` objects to avoid memory leaks, when we free a `BinaryTree` object.

```
static void delete_node( TreeNode* node )
{
   if (node->left  != NULL) delete_node( node->left );
   if (node->right != NULL) delete_node( node->right );
  free( node );
}
void BinaryTree_delete(BinaryTree* tree)
{
   if ( tree->root != NULL ) delete_node( tree->root );
   free( tree );
}
```

# Overview

Introduction

Getting started

Types, arithmetic operators

Control flow

Functions and program

Pointers and arrays

Structures

Input and output

# General remarks on input and output in C

Input and output are not built into the language

Provided through library functions

C implements a simple model

The console, the keyboard, a file, everything is a stream

Each line ends with `'\n'`

A file ends with `EOF`

- `EOF` is defined in `stdio.h`

# Standard input

Usually, the standard input is the keyboard

It is possible to redirect the standard input:

```
./myprog < infile
```

The content of `infile` becomes the standard input for `myprog`

```c
int getchar()
```

- Basic input function
- Returns the next character from standard input
- Returns `EOF` if the end of the file is reached

# Standard output

Usually, the standard output is the screen

It is possible to redirect the standard output:

```
./myprog > outfile
```

Writes the standard output to `outfile`

```
int putchar( int c )
```

- Basic output function
- Writes the character `c` to the standard output
- Returns `EOF` if an error occurred else it returns `c`

# printf

`int printf(char* format, arg1, arg2, … )`

- Converts, formats, and prints arguments to the standard output
- Returns the number of characters printed
- Declared in `stdio.h`

The `format`-string formats the arguments

- Ordinary characters are copies to the output stream
- After % a description starts how to convert a variable
- The printed variables are passed as additional parameters in the order they appear in the `format`-string

# Variable output format

| Symbol | Description |
|---|---|
| % | Starts always with this character |
| - | Optional: Specifies left adjustment |
| A number | Optional: Minimum field width |
| Period and number | Optional: Maximum number of characters for a string / Number of digits after decimal point for floating point numbers |
| h | For short integers |
| l | (uppcase L) for long integers |
| Character | The conversion character specifies the data type of the variable. See next slide |

# Conversion characters

| character | Description |
| --- | --- |
| `d, i` | `int`: decimal number |
| `o` | `int`: unsigned octal number |
| `x, X` | `int`: unsigned hexadecimal number |
| `u` | `int`: unsigned decimal number |
| `c` | `int`: single character |
| `s` | `char*`: Prints a string until '\0' |
| `f` | `double`: Floating point number |
| `p` | `void*`: Adress |

# Examples - printf

| `char* s=“hello, world”;`<br>`double d=3.14;` | Output |
|---|---|
| `printf(“:%s:”, s);` | `:hello, world:` |
| `printf(“:%10s:”, s);` | `:hello, world:` |
| `printf(“:%.10s:”, s);` | `:hello, wor:` |
| `printf(“:%.15s:”, s);` | `:hello, world:` |
| `printf(“:%-15s:”, s);` | `:hello, world   :` |
| `printf(“:%15.10s:”, s);` | `:     hello, wor:` |
| `printf(“:%-15.10s:”, s);` | `:hello, wor     :` |
| `printf(“:%f:”, d);` | `:3.140000:` |
| `printf(“:%6.2f:”, d);` | `:  3.14:` |

# scanf

```
int scanf( char* format, arg1, arg2, … )
```

- Like reverse `printf`
- Reads characters from standard input
- Interprets them according to the format
- Stores the results into the memory location given in the arguments
- All arguments need to be pointers to valid memory locations
- Declared in `stdio.h`

```
double d;
scanf("%f", &d );  // Reads a floating point number
scanf("Result: %f", &d); // Expects "Result: " and a
                         // floating point number
```

# sscanf

```
int sscanf( char* string, char* format, arg1, … )
```

- Like `scanf` but parses `string` instead of standard input

```
#include <stdio.h>
int main() {
  char *s = "Result: 3.14";
  float d;
  sscanf ( s, "Result: %f", &d );
  printf( "d = %.2f\n", d );
}
```

# Open a file

Before a file can be read or written, it needs to be opened

```
FILE* fopen( char* name, char* mode )
```

- Declared in `stdio.h`
- `name` is the name of the file
- `mode` can be one of the following or a combination

| Mode | Description |
|---|---|
| r | read |
| w | Write, old content is discarded |
| a | Append, old content is kept |

```
FILE* fp = fopen( "my_file", "r" ); // read-only
fp = fopen( "my_file", "rw" ); // read and write
```

# Read from a file

`int getc( FILE* fp )`

- Returns the next character from `fp`

`int fscanf( FILE* fp, char* format, arg1, arg2, … )`

- Like `scanf`, but reads from `fp`

`char* fgets( char* line, int maxline, FILE* fp )`

- Reads one line (including `'\n'`)
- At most `maxline-1` characters are read
- The characters are stored in `line`
- The user has to allocate the buffer before
- Returns `NULL` in case of error, else `line` is returned
- The line is termined with `'\0'`

# Write to a file

`int putc( int c, FILE* fp )`

- Writes one character to `fp`

`int fprintf( FILE* fp, char* format, … )`

- Like `printf`, but writes to `fp`

`int fputs( char* line, FILE* fp )`

- Writes a string to `fp`
- Does not need to contain a newline
- Returns the number of written characters

# Close a file

Closing a file frees system resources

Maximum number of open files

Flush buffers

```
int fclose( FILE* fp )
```

# Example – copy

```c
#include <stdio.h>
int main(int argc, char** argv){
  if (argc < 3) {
    printf("Usage: %s source destination\n”, argv[0]);
     return 1;
  }
  FILE* source = fopen( argv[1], "r" );
  FILE* dest = fopen( argv[2], "w" );
  char buffer[256];
  while ( fgets(buffer, 256, source) != NULL ) fputs(buffer, dest);
  fclose( source );
  fclose( dest );
}
```

# Read and write formatted binary data

```
size_t fread ( void* ptr, size_t size,
                    size_t count, FILE * fp )
```

- Reads `count` elements of size `size` from `fp`
- The user has to provide a buffer `ptr` that has at least `size * count` bytes
- Returns the number of bytes read

```
size_t fwrite ( const void * ptr, size_t size,
                    size_t count, FILE * fp )
```

- Writes `count` elements of size `size` from `ptr` to `fp`
- Returns the number of bytes written

# Stdin, stdout, stderr

`stdio.h` defines three constant `FILE*` variables

- `stdin` is the standard input stream
- `stdout` is the standard output stream
- `stderr` is the standard error stream
  - By default, `stderr` prints to the screen
  - Writing errors to `stderr` prints them even if `stdout` is redirected

It is possible to use them with file input/output

- In fact `printf`, `scanf`, etc. are only a redirect to `fprintf`, `fscanf`, etc.

```
fprintf( stdout, "hello world\n" );
```

# fseek

Usually, you read/write a file in sequential order

`fseek` allows to set the current position in a stream

```
int fseek ( FILE * fp, long int offset, int origin )
```

The new position is `offset` bytes from `origin`

`origin` is one of the following constants

| Constant | Reference position |
|---|---|
| `SEEK_SET` | The beginning of the file |
| `SEEK_CUR` | The current position |
| `SEEK_END` | The end of the file |

# ftell

```c
long int ftell ( FILE * stream )
```

- Returns the current position in a stream

## Get the file size

```
#include <stdio.h>

int main()

{

  FILE* fp = fopen( "myfile", "r");

  fseek( fp, 0, SEEK_END );

  long int length = ftell( fp );

  fclose( fp );

}
```

# Status of a stream

```
int ferror ( FILE * stream )
```

- Returns true if an error occurred on a stream

```
int feof ( FILE * stream )
```

- Returns true if the end of a file was reached

# The C preprocessor

The C preprocessor parses the source code before compilation

It performs text substitutions and modifications

Preprocessor directives control actions of the preprocessor

The `#include` directive was already introduced

- Inserts the content of a file at the position of the directive

# #define directive

`#define` *`macro definition`*

Defines a macro

The pre-processor will substitute all occurrences of *`macro`* by *`definition`*

It is often used to define constants

The value is set at one place

Makes sure that it is changed at all occurrences

```
#define BUF_SIZE 16384
char buf[ BUF_SIZE ];
int i;
for( i = 0; i < BUF_SIZE; i++ ) buf[ i ] = i;
```

# #define with parameters

A `define`-directive can have parameters

The text of the parameter is put at the place in the definition where the parameter occurs

Put the parameter in brackets to make sure that the expression that is passed is evaluated correctly and does not interfere with the macro operation

```
#define MAX( a, b ) ( (a) > (b) ? (a) : (b) )
int d = MAX( d, 5*d-6 );
```

Is replaced to

```
int d = ( (d) > (5*d-6) ? (d) : (5*d-6) );
```

# Conditional compilation (1)

```
#if condition
// some source code
#else
// more source code
#endif
```

The preprocessor evaluates *`condition`* at compile time

- *`Condition`* must be a constant

If it is false the lines between `#if` and `#else` are removed

Otherwise the lines between `#else` and `#endif` are removed

The `#else` is optional

# Conditional compilation (2)

`#ifdef` *`macro`*

- True if *`macro`* was defined

`#ifndef` *`macro`*

- True if *`macro`* is undefined

Typical use case to avoid double definitions in header files

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H
// All declarations
#endif
```

# Outlook

The introduction covered the most important parts

Many details are not covered by the introduction

Some topics not covered are (incomplete list):

- Function pointers
- Variable length argument lists
- Bit fields
- Bit-wise operators
- Enums
- Unions

# Important differences between standards

C99

- Free placement of variable declarations
- Variable declaration in first expression of for-loop

```
for( int i=0; i<10; i++)
```

- Definition of integer types with fixed size in `stdint.h`

```
int32_t, uint64_t, int16_t, …
```

- Definition of `bool`, `true` and `false` in `stdbool.h`

C11

- Multithreading support
- Removal of `gets()`
- Definition of larger character types for Unicode support

# C++ SPOTLIGHTS

# Outline

Introduction

References

Classes and objects

Exceptions

Templates

# Literature

Books:

- Stroustrup, *The C++ Programming Language*
- Stroustrup, *Die C++ Programmiersprache*
- Lippman and Lajoie, *C++ Primer*,

Online:

- Bernd Mohr, Programming in C++ (Slides): http://d-nb.info/973093625/34

# History

- 1979 Stroustrup starts to extend C with classes
- 1983 “C with classes” renamed to C++
- 1985 First version of C++
- 1989 Second version of C++, multi-inheritance, abstract classes, static/const methods, …
- 1998 First ISO C++ standard (C++98), templates and STL
- 2003 C++03, clarifications
- 2011 C++11, threading, lambdas, rvalue-references, …
- 2015 C++14, minor extensions to C++11
- 2017 C++17, minor extensions to C++14

# C++

C++ extends C

- Support for object oriented programming
- Generic programming (templates)
- Some more extensions

C++ is compatible to C89

- C includes modifications in C99 and later that are incompatible with C++
- C is still “mostly” compatible with C++

# Overview

Introduction

References

Classes and objects

Exceptions

Templates

# References

References are a new name for a variable.

```
int i = 42;
int &r = i;
```


i, r

If a function declares a parameter as a reference, the variable is just another name for the passed variable.

```
void func( int& r ); // Any modification to
func( i );           // r modifies i, too
```

Here, `r` and `i` use the same memory location

# References vs. Pointers

References have some of the properties of pointers

- Passing objects without creating a copy
- Passed object can be modified inside the function

But avoid some of its problems

- Memory leaks

Use references instead of pointers if possible

# Overview

Introduction

References

Classes and objects

Exceptions

Templates

# Object-oriented programming

Programming paradigm

Abstract idea

- Existence of objects
- Objects have attributes which are described in member variables
- Actions, associated with an object, are described in methods

A class describes the members and method for a certain type of objects

- Classes are types

Objects are instances of a class

- Objects are variables

# Classes

```
class classname {
  // member and method declarations
};
classname A; // Creates an object of the class
```

Example:

```
class Car {
};
Car A;
```

## Members

Members are variables of a class

- Writes a variable definition inside the class
- Access to members like for `structs`

```
class Car {
  double max_speed;
};
Car A;
Car* B = &A;
B->max_speed = A.max_speed;
```

# Methods

Methods are functions that are defined in a class

- Access to methods via . or -> like for members
- Can access member variables of the object
- For implementation the method name is prefixed with `classname::`

```
class Car {
  double max_speed;
   void drive( double distance );
};
void Car::drive( double distance ) {
  double time = distance/max_speed;
}
```

# Overloading

C++ allows multiple methods with the same name

The signature must differ

Allowed

```
class A
{
  foo();
  foo(int a);
  foo(double b);
};
```

Error

```
class A
{
  foo();
  foo(int a);
  foo(int b);
};
```

# Accessibility

Allow access only to interface functions

`private:` Only methods of this class can access the symbol

- Default is private

`protected:` Accessible for methods of this class and derived classes

`public:` Everybody can access the symbol

```
class Car {
private:
  double max_speed;
public:
  void drive( double distance );
};
```

# Inheritance

If one class is a special case from a more general class

- E.g. an apple tree is a special case of a tree
- The class of the special case can be derived from the base class
- The child class inherits all members and methods from the base class
- The accessibility of the members/methods of the base class can be restricted
  - Is the base class public, the accessibility is unchanged

```
class Tree
{
protected:
  double height
};
```

```
class AppleTree : public Tree
{
public:
  double getHeight { return height; }
};
```

## Class Construction (1)

Constructors create a new instance of a class

Constructor definition looks like a method

- With the same name as the class
- Has no return type
- Can have parameters

```
class MyClass {
public:
  MyClass( int arg );
};
```

# Class Construction (2)

The constructor can/should call

- constructors of parent classes
- constructors of members

```
class MyClass : public ParentClass {
private:
  int data;
public:
   MyClass( int arg ) :
       ParentClass( arg ),
       data( 42 ) {}
};
```

# Create local objects

```
class MyClass {
  MyClass();
  MyClass( int i );
};

MyClass A(4);
MyClass B();
MyClass C;  // Invokes MyClass()
```

# Default Constructors

The compiler may generate some constructors:

- The default constructor:
  - If no custom constructor exists
  - Has no arguments
  - Equals a constructor with an empty body
  - Calls default constructor of base classes and members
- The copy constructor
  - If no copy constructor exists
  - Has a reference of an object of the same type as parameter
  - Initializes all members with the values of the copied object
  - Caution: Pointers copy only addresses, not the object. (shallow copy)

# Destructor

A destructor is called when an object is deleted.

Only one destructor can exist for each class

The destructor has the name of the class and a leading '~'

```
class my_class {
public:
  virtual ~my_class();
};
```

If no destructor is provided, the compiler generates a default destructor with an empty body

Base classes usually require virtual destructors

# Dynamic Allocation

```
class MyClass {
  MyClass();
  MyClass( int i );
};
```

Dynamic allocation with `new`

```
MyClass* D = new MyClass(4);
MyClass* E = new MyClass();
```

Dynamically allocated objects must be explicitly deleted

```
delete(D);
delete(E);
```

# Passing objects

```
class myClass;

void foo( myClass A, myClass* B, myClass& C ) {
  // A is a copy of orig, invokes copy constructor
  // B points to orig, changes have side effects
  // C is a new name for orig, have side effects
}
myClass orig();
foo( orig, &orig, orig );
```

# Passing parameters: C/C++ vs. Java

Passing native data types

- Java behaves like C/C++ value copy

Passing objects

- Java behaves like C++ references

# Overview

Introduction

References

Classes and objects

Exceptions

Templates

# Exceptions (1)

Exceptions prove a mechanism to react on exceptional circumstances

- E.g. runtime errors

They are no means to return a value from a function

- Extremely high overhead

# Exceptions (2)

```
try
{
  // Code under inspection
  throw my_exception();
}
catch( my_exception& e )
{
  // Reaction on exception e
}
```

If an exception was caught, execution continues after the `catch` block

- Not after the `throw` statement

# Throw

The `throw` command excepts one parameter that is passed as an argument to the `catch` clause

Can be an arbitrary type

- You can throw basic types, like integers

The C++ standard library provides a base class for all exceptions it throws: `std::exception`

# Catch

A `catch` block catches only exceptions of matching type

A `catch` block must immediately follow the try block

You can chain `catch` blocks

```
try {
  //something
} catch( my_exception& e ) {} // exception by reference
  catch (std::exception& e ){}
  catch (int e){}                // passes value by copy
  catch(…){}                     // catches all exceptions
```

# Overview

Introduction

References

Classes and objects

Exceptions

Templates

# Motivation – templates

Imagine you implement a data structure, e.g. a stack

And later you need the same data structure again, but for another data type

You could copy/paste all the code and change the data type everywhere

- Duplication of code

You could store void pointers to the data objects

- Limits the data types to pointers
- No type checking by the compiler

Wouldn't it be nice to just write a template from which the compiler generates different versions?

# Templates

C++ provides templates

They allow to describe an algorithm in an generic way,

generate multiple versions of this algorithm,

and tell the compiler that in each version of the algorithm it should insert a specific type.

# Example – stack with fixed type

```
class stack {
  int *data;
  int size, top;
public:
  stack(int s) : size(s), top(0){
    data = new int[size];
  }
  virtual ~stack();
  void push( int new_node);
  int pop();
};
```

# Example – stack with template type

```
template <typename T>
class stack {
  T *data;
  int size, top;
public:
  stack(int s) : size(s), top(0){
    data = new T[size];
  }
  virtual ~stack();
  void push( T new_node);
  T pop();
};
```

# Instantiation

Instantiation of non-template class:

```
stack stack_with_fixed_type( 100 );
```

Instantiation of template class:

```
stack<int> stack_with_template( 100 );
```

The code for `stack<int>` is generated when the compiler encounters the instantiation

Some compilers require all template code to be in the header

The `<int>` becomes part of the type name and appears everywhere you would write the type name

# Template features

You can instantiate a template type with another template type

```
stack < my_other_template < int > >
```

Creates a stack object that stores objects of type

```
my_other_template < int >
```

You can have multiple type parameters in a template definition:

```
template <typename A, typename B>
class my_class { … };
```

You can also generate templates for a method:

```
template <typename T>
void sort( T* array, int size );
```

# Standard Template Library (STL)

Part of the C++ Standard

Defines a set of common data structures and algorithms

Makes heavy use of templates

STL classes are in the namespace `std`

# STL containers

Containers are objects that store a collection of objects

Implemented as templates to support different data types

All implement a similar interface

Containers differ in their algorithmic complexity to insert, remove or access an element

Some example STL data structures are:

```
template <typename T> std::vector
template <typename T> std::deque
template <typename key_t, typename value_t>
    std::map
```

# std::vector

The class `std::vector` is like an array that can grow dynamically

- Uses reallocation internally to adapt size

Fast element access

Appending/removing elements at the end is fast

Slow insertion/removal of elements at other positions

```
std::vector<std::string> my_vec;

my_vec[0] = “Daniel”;

std::string s = my_vec[0];
```

# std::map

Maps are associative containers that associate a key with a value

The key can be any comparable type

Keys must be unique

```
std::map<std::string, long> my_map;
my_map[“Daniel”] = 123456;
long n = my_map[“Daniel”];
```

# Iterators (1)

Iterators are objects that point to an element in a container

Iterators provide operators to iterate over the elements of the container

Usage of iterators look similar to pointers

```
std::vector<string> my_vec;
std::vector<string>::iterator i;
for( i = my_vec.begin(); i != my_vec.end(); i++ )
{
  string current = *i; // Use i-> to access members directly
}
```

# Iterators (2)

Iterators are class objects

Overwrite various operators, e.g.

- Smart pointers ->
- Dereference operator *
- Increment operator ++
- Comparison operator ==

Iterator objects are tied to a container and an iteration order

- E.g., different iterator classes for iteration in forward and reverse order

Provide reliable checks for boundaries