Folien zur Vorlesung Programmierung für Naturwissenschaften, Sommersemester 2015

Stefan Kurtz
Arbeitsgruppe Genominformatik
Zentrum für Bioinformatik
Universität Hamburg

June 4, 2015

C for Java Programmers
A Programming Course

Stefan Kurtz

Research Group for Genome Informatics
Center for Bioinformatics Hamburg
University of Hamburg

June 4, 2015

Credits & References I

Credits

- original slides from Indranil Gupta, Department of Computer Science, Cornell Univ. Ithaca, NY, USA.
- slides are part of a course in operating systems
- conversion from ppt to latex beamer format by Dirk Willrodt

Credits & References II

Further Reading

- A tutorial on pointers and arrays in C, Ted Jensen, version 1.2, `http://pw1.netcom.com/~tjensen/ptr/pointers.htm`.
- C for Java Programmers, J. Maassen, `http://faculty.ksu.edu.sa/jebari_chaker/papers/C_for_Java_Programmers.pdf`
- C++ for Java Programmers, Tom Pittman, `http://www.ittybittycomputers.com/Courses/Prior/ADS/C4Java.htm`
- Programming, Problem Solving, and Abstraction with C, Revised Edition, 2013, Alistair Moffat, ISBN 9781486010974, Pearson `http://people.eng.unimelb.edu.au/ammoffat/ppsaa/`
- Head First C, David Griffiths, Dawn Griffiths, 2012, O'Reilly Media, PDF freely available at `http://it-ebooks.info/book/704/`

Overview

- Why learn C after Java?
- A brief background on C
- C preprocessor
- Modular C programs

Index I

Index II

Index III

Why learn C (after Java)?

- C is high-level and low-level language
  - useful for algorithm development and user interfaces to operating system (OS) kernel and device drivers
- better control of low-level mechanisms
  - memory allocation, specific memory locations
- performance often better than Java
  - usually more predictable than Java
- Java hides many details needed for high performance computing:
  - memory management responsibility
  - explicit initialization and error detection
  - generally, more lines for same functionality

Why learn C?

- being multi-lingual is good!
- C and C++ are widespread languages:
  - MS-Windows: written in C++, kernel in C
  - Mac OS X: written in Objective C, kernel in C
  - Linux: mostly written in C, KDE is all C++
  - Java compiler and interpreter: written in C
  - Most embedded systems: written in C
  - large fraction of software for applications in the sciences is written in C

History

- C
  - developed by Dennis Ritchie in late 1960s and early 1970s
  - C is procedural language
  - originally developed as systems programming language
    - make operating system portable across hardware platforms
- C++
  - Bjarne Stroustrup (Bell Labs), 1980s
  - object-oriented features
- Java
  - James Gosling in 1990s, originally for embedded systems
  - object-oriented, like C++
  - much more high-level than C
  - much of syntax is taken from C

Advantages and disadvantages of C

C advantages

- direct access to operating systems primitives (system calls)
- fewer library issues – just execute a function

C disadvantages

- **A**pplication **P**rogrammer **I**nterface (API) is not fully portable with subtle differences between different platforms
- memory handling by the programmer is a burden and needs experience (most common source of errors in existing programs)
- preprocessor can lead to obscure errors

Java vs. C

Java	**C**
object-oriented	function-oriented
strongly-typed	types can be overridden
polymorphism (+, ==)	limited polymorphism (integer/float)
classes for name space	(mostly) single name space, file oriented
macros are external, rarely used	macros common (preprocessor)
layered I/O model	byte-stream I/O

Java vs. C

Java	C
automatic memory management	function calls (C++ has some support)
no pointers	pointers (memory addresses) common
by-reference, by-value parameters	by-value parameters
exceptions, exception handling	error codes: `if (f() < 0) error` OS signals
concurrency (threads)	library functions
length of array	on your own
string as type	strings = byte arrays (`char s[LEN+1]`)
dozens of common libraries	libraries are OS-defined

Java vs. C

- Java program is
  - collection of classes
  - class containing main method is starting class
  - running `java` *StartClass* invokes *StartClass*`.main` method
  - JVM (Java Virtual Machine) loads other classes as required
- C program is
  - collection of functions
  - one function – `main()` – is starting function
  - running executable starts `main()` function
  - typically, single program with all user code either
    - linked at compile time or
    - loaded from dynamic libraries (`.dll`, `.so`)

Java vs. C

```
public class hello {
  static public void
  main(String args[])
  {
    System.out.println
      ("Hello world");
  }
}
```

```
#include <stdio.h>
int main(int argc,char *argv[])
{
  printf("Hello, World\n");
  return 0;
}
```

- note that parameters `argc` and `argv` are not used

⇒ warning when compiling the program

Simple example

```
#include <stdio.h>
int main(void)
{
          /* print out a message */
  printf ("Hello World. \n \t and you ! \n ");
  return 0;
}
```

```
$ hello.x
Hello, World
        and you !
$
```

- `hello.x` is the executable compiled from `hello.c`
- `$` is command line prompt after which we type a shell command

Dissecting the example

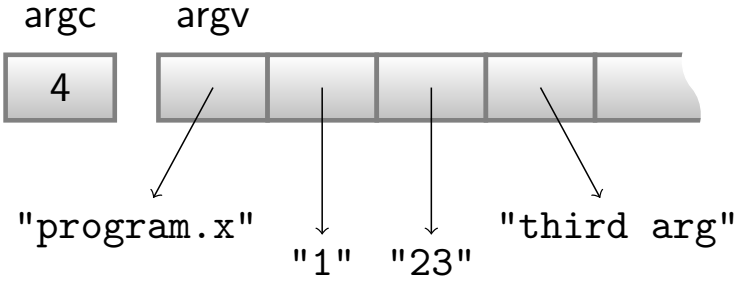
- `#include <stdio.h>`
  - include header file `stdio.h`
  - # lines processed by pre-processor (part of compiler)
  - no semicolon at end of pre-processor command
  - lower-case letters only – C is case-sensitive
  - </>-bracket is used for system header file
- `int main(void){ ... }` is the only code executed
- `printf("/*␣message␣you␣want␣printed␣*/");`
- \n = newline, \t = tab
- \ in front of other special characters within `printf`.
  - `printf("let's␣output␣\"Hello,␣World\"\n");`
- the `int` value in `return 0` is the return value
  - convention: 0 means success, $\neq 0$ some error
  - better use constants `EXIT_SUCCESS`/`EXIT_FAILURE` defined in `stdlib.h`
- instead of `return` one can use `exit(..)` which works for any function (not just `main`)
- `exit(..)` immediately stops execution of the program

Executing the C program

- usually C-programs get input (e.g. options and files) via command line

⇒ `int main(void)` is rarely used

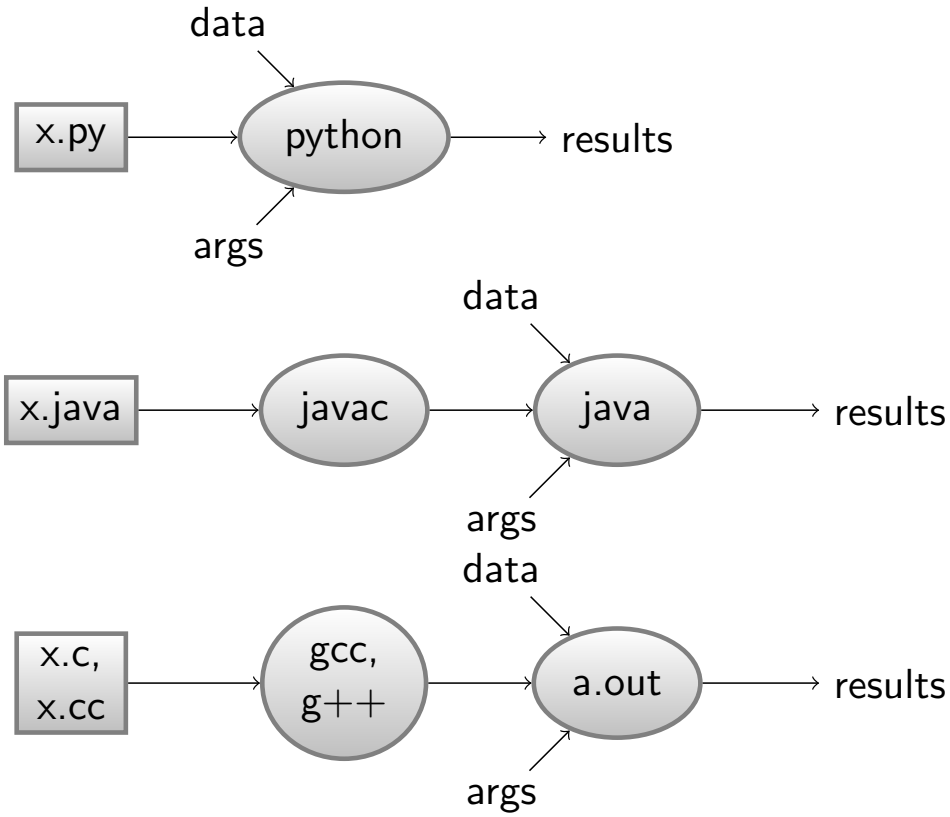
- more common: `int main(int argc, char *argv[])`
  - `argc` is the argument count
  - `argv` is the argument vector
    - array of strings with command-line arguments
- name of executable and space-separated arguments
- `$ program.x 1 23 'third arg'`



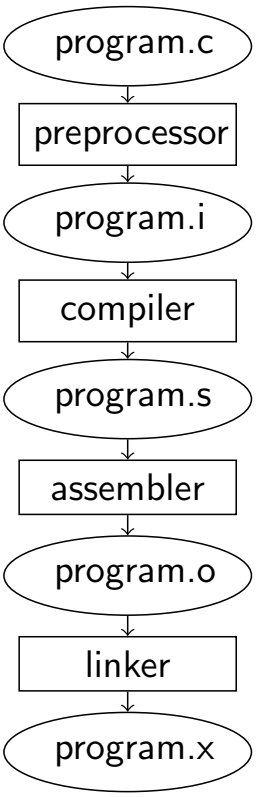
Executing programs in different languages

- scripting languages are usually interpreted
  - perl, python, ruby-interpreters read script, and executes it
  - sometimes, just-in-time compilation – invisible to user
- Java programs are semi-interpreted:
  - `javac` converts `foo.java` into `foo.class`
  - not machine-specific
  - *byte codes* are then interpreted by the Java Virtual Machine (JVM)
- C programs are normally compiled and linked by compiler

Executing programs in different languages



The different steps of compiling a C-program I



macro substitution
strip of comments
expansion of included files

`gcc -Wall -E program.c`
only runs prepro.

syntax and semantics checks
first level translation

option `-save-temps`
delivers `.i`/`.s` files

low level instructions
independent of the concrete processor

translation to machine code for CPU
(in ELF[1] format), not yet executable

link of function calls with their definitions

The C compiler gcc

- gcc invokes C compiler
- gcc translates C program into executable for some target
- default file name for executable is `a.out`
- executable is executed by OS and hardware

```
$ gcc hello.c
$ a.out
```

```
Hello, World
```

```
$ gcc -Wall -Werror -O3 -o hello.x hello.c
$ hello.x
```

```
Hello, World
```

gcc: command-line switches

option	what it does
`-o file`	output file for object or executable
`-Wall`	all warnings – always use it
`-c`	compile single module (non-main)
`-g`	insert debugging code (for gdb and valgrind, always use it)
`-p`	insert profiling code
`-l`	specify library
`-E`	preprocessor output only

Using gcc

- one-stage compilation
  - pre-process, compile and link: `gcc -o hello.x hello.c`
- two-stage compilation
  - pre-process & compile: `gcc -g -c hello.c`
  - link: `gcc -o hello hello.o`
- linking several modules:
  - `gcc -g -c a.c` → `a.o`
  - `gcc -g -c b.c` → `b.o`
  - `gcc -o hello.x a.o b.o`
- using the math library
  - `gcc -g -o calc.x calc.c -lm`
- recommendation: use `clang` instead of `gcc`
- `clang` error messages are less cryptic
- see `http://clang.llvm.org/get_started.html`
- `clang` is installed on computers in student pool
- statements about `gcc` hold for `clang` as well

Using a Makefile

- save typing the calls to the compiler (usually many times)

```
CC=clang
CFLAGS=-Wall -Werror -g -O3

all:test.x\
    hello.x

%.x:%.c
        ${CC} ${CFLAGS} -o $@ $<

clean:
        rm -f *.o *.x
```

- make tries to compile `test.x` from `test.c` and `hello.x` from `hello.c`.
- if the C-files do not exist, then an error is reported.
- if `test.x` exists and is newer than `test.c`, then nothing needs to be done for `test.x`.

Error reporting in gcc

- multiple sources for errors
  - from preprocessor: e.g. missing include files
  - from parser: syntax errors (e.g. missing bracket) or semantic errors (e.g. missing var-declaration)
  - from assembler: rare
  - from linker: missing libraries or undefined symbols
- if gcc gets confused, hundreds of messages will appear on stderr
  - redirect stderr to stdout and pipe it into less (in own terminal window)

    `make 2>&1 | less`
  - edit files containing errors in another terminal window
  - fix first errors, leave less with 'q' and then retry make – ignore the rest
- `gcc -Wall` will produce an executable (possibly) with warnings
  - never ignore warnings – they are almost always helpful and reveal possible mistakes, as e.g.
    `if (x = 0)` vs. `if (x == 0)`

gcc linker errors

- produces object code for each module
- assumes that references to external names will be resolved later
- undefined symbols will be reported when linking:

```
undefined symbol    first referenced in file
_print              program.o
ld fatal: Symbol referencing errors
No output written to file.
```

- here a function `print` was called in `program.c` but not defined in any of the linked object files

C preprocessor

- The C preprocessor (cpp) is a macroprocessor which
  - manages a collection of macro definitions
  - reads a C program and transforms it
  - example:

```
#define MAXVALUE 100
#define CHECK(X) ((X) < MAXVALUE)
if (CHECK(i))
{
  i = i + 2;
}
```

cpp transforms this to the following:

```
if ((i) < 100)
{
  i = i + 2;
}
```

C preprocessor

- preprocessor directives start with # at beginning of line:
  - define new macros
  - input files with C code (typically definitions)
  - conditionally compile parts of file
- `gcc -E` shows output of preprocessor
- can be used independently of compiler (for example, for textprocessing; not really recommended, better use scripting languages)

C preprocessor

```
#define NAME constantexpression
#define NAME (PARAM1,PARAM2,...) expression
#undef SYMBOL
```

- replaces `NAME` with constant or expression
- textual substitution
- symbolic names for global constants
- in-line functions (avoid function call overhead)

  mostly unnecessary for modern compilers
- type-independent code, e.g. `#define MAX(X,Y) ((X) > (Y) ? (X) : (y))` works for integers and floating point values

C preprocessor

- Example: `#define MAXLEN 255`
- Lots of system `.h` files define macros
- invisible in debugger
- `getchar()`, `putchar()` in stdio library

△ caution: do not treat macros like function calls

```
#define VALID(X) ((X) > 0 && (X) < 20)
if (VALID(x++))
{
  /* do sth */
}
VALID(x++) → ((x++) > 0 && (x++) < 20)
```

- variable `x` is incremented once or twice depending on its value (probably not intended)

C preprocessor – file inclusion

```
#include "filename.h"
#include <filename.h>
```

- inserts contents of filename into file to be compiled
- `"filename.h"` relative to current directory
- `<filename.h>` relative to `/usr/include`
- `gcc -Isrc/include` specifies to also look for header files in `src/include`
- header files are mostly used to import function prototypes
- examples:
  - `#include <stdio.h>`
  - `#include "mydefs.h"`
  - `#include "/home/alice/defs.h"`
- `#include` of files with absolute path (as in last example) not recommended (program will not work in other directory structures)

C preprocessor – conditional compilation

```
#if expression
code segment 1
#else
code segment 2
#endif
```

- preprocessor checks value of expression
- if true, outputs code segment 1, otherwise code segment 2
- often used for machine or OS-dependent code
- can be used to comment out chunks of code (instead of using /* ... */)

```
#define OS Linux /* better use -DOS=`uname -s` */
...
#if OS == Linux
  printf("Linux!");
#else
  printf("Something else");
#endif
```


C preprocessor - ifdef

- for expressions which consist of an identifier use `#ifdef`:

```
#ifdef USEDB
code segment 1
#else
code segment 2
#endif
```

- preprocessor checks if name `USEDB` has been defined by
  - `#define USEDB` or
  - option `-DUSEDB` to compiler
- if so, use code segment 1, otherwise code segment 2
- if `USEDB` is already defined and one wants it to be undefined use:

```
#undef USEDB
```

Advice on preprocessor

- limit use as much as possible
  - subtle errors
  - not visible in debugging
  - code hard to read
- much of it is historical baggage
- there are better alternatives for most uses, e.g.
  - `#define INT16 short` → type synonyms
  - `#define MAXLEN 100` → const variable (restricted to certain type)
- limit use to `.h` files, to isolate OS & machine-specific code

Comments

- `/* any text until */`
- `//` C++-style comments
- convention for longer comments:

```
/*
 AverageGrade()
 Given an array of grades, compute the average.
*/
```

- avoid ** boxes – hard to edit, usually look ragged

Numeric data types

type	32 bit bytes	32 bit range	64 bit bytes	64 bit range
`char`	1	-128 to 127	1	-128 to 127
`short`	2	-2^{15} to $2^{15}-1$	2	-2^{15} to $2^{15}-1$
`int`	4	-2^{31} to $2^{31}-1$	4	-2^{31} to $2^{31}-1$
`long`	4	-2^{31} to $2^{31}-1$	8	-2^{63} to $2^{63}-1$
`long long`	8	-2^{63} to $2^{63}-1$	8	-2^{63} to $2^{63}-1$
`float`	4	3.4^{-38} to 3.4^{38}	4	3.4^{-38} to 3.4^{38}
`double`	8	1.7^{-308} to 1.7^{308}	8	1.7^{-308} to 1.7^{308}

- `unsigned` versions of integer types (same bits, different interpretation)
- `char` is one byte which stores 1 "character", but only true for ASCII and other western char sets
- `float`/`double`: positive values have negative counterpart with sign bit set

Explicit and implicit conversions

- many expressions and assignments contain variables of different type

$\Rightarrow$ different types require type conversions

- often implicit type conversion suffices

```
int i, j = 12;          /* i not initialized, only j */
float f1, f2 = 1.2;

i = (int) f2;          /* explicit: i <- 1, 0.2 lost */
f1 = i;                /* implicit: f1 <- 1.0 */

f1 = f2 + (float) j; /* explicit: f1 <- 1.2 + 12.0 */
f1 = f2 + j;           /* implicit: f1 <- 1.2 + 12.0 */
```

Explicit and implicit conversions

- implicit: e.g., `int i; short s; char c; i = s + c;`
- promotion: `char` → `short` → `int` → `long`
- if one operand is `double`, the other is made `double`
- if one operand is `float`, the other is made `float`, ...
- explicit conversion via type casting: `(type)`
- almost any conversion does something, but not necessarily what is intended

```
int x = 100000;
short s;

s = x;
printf("%d␣%hd\n", x, s);
```

```
100000 -31072
```

- value 100000 cannot be stored in 16 bits ⇒ some bits get lost ⇒ incorrect value in `s`

Boolean values

- C does not have boolean values
- emulate as `int` or `char`, with values 0 (false) and non-zero (true)
- best to use `#include <stdbool.h>`
- contains definitions:

```
typedef char  bool;    /* type synonym */
#define false 0
#define true  1
```

- be careful when testing booleans

```
int check = x > 0;
if (check == true) {...}
```

- If `x` is positive, check will be non-zero, but not necessarily `1`

⇒ better write

```
if (check) {...}
```

User-defined types

- `typedef` gives names to types:

```
typedef short int      SmallNumber;
typedef unsigned char  Byte;

SmallNumber x;
Byte b;
```


Enumerated types

- define new integer-like types as enumerated types:

```
typedef enum
{
  Serine, Leucine, Tyrosine, Tyrosine, ..., Glycine
} Aminoacid;

typedef enum {Adenine, Cytosine, Guanine, Thymine}
             Nucleotide;

typedef enum {false, true} bool;
```

- `Aminoacid`, `Nucleotide` and `bool` are new type-identifiers
- their values are listed in the declaration and treated like integers
  - one can add, subtract – even `Serine + Glycine`
  - cannot directly print as symbol (requires extra code)
  - but debugger generally will show them as their names

Variables

- C does not support declaration of classes and creation of instances of classes (objects)
- all data is stored in variables and there are operators to manipulate the values of the variables
- variables for C's primitive types are defined very similarly:

```
short int x;
char  ch;
float pi = 3.1415;
float f, g;
```

- variables defined in {} block are visible and active only in block
- variables defined outside a block are global (persist during program execution), but may not be globally visible (when declared `static`)

Variables and data

- in C a variable can be considered a container holding a value
  - data is stored as a multiples of CPU words
- value of uninitialized variable is (mostly) undefined – treat as random
  - compiler warns you about uninitialized variables

    `ch = 'a'; x = x + 4;`
- every variable in C has
  - a name and data type (specified in definition)
  - an address (its relative location in memory)
  - a size (number of bytes of memory it occupies)
  - visibility (parts of program that can refer to it)
  - lifetime (period during which it exists)

Variables and data

- warning: do not try to access data after its lifetime, as in the following examples (`&` is address operator):

```
int *foo_param(int x)
{

  return &x;
}
ptr1 = foo_param(z);
ptr2 = foo_localvar();
```

```
int *foo_localvar(void)
{
  int y = 5;
  return &y;
}
```

- `ptr1` is pointer to memory cell for copy of `z`, which does not exist after the function call
- `ptr2` is pointer to memory cell for variable `y`, which does not exist after the function call
- both pointers refer to memory cells after their lifetime

```
*ptr1 = 17; /* write value to invalid address */
*ptr2 = 42; /* write value to invalid address */
```


The sizeof operator

The size of a type, e.g. `int`, or a variable, e.g., `int x`

- can be obtained via `sizeof(int)` or `sizeof(x)`
- has data type `size_t`
- has a value which is a small(ish) integer
- is measured in bytes

```
#include <limits.h> /* for CHAR_BIT */
#define BITS(X) (sizeof (X) * CHAR_BIT)
#define ISSIGNED(TYPE) ((TYPE) -1 < 0)

int main(void)
{
  printf("this␣is␣a␣%lu-bit␣machine\n",BITS(void *));
  printf("size_t␣is␣%ssigned\n",ISSIGNED(size_t) ? "" : "un");
  return EXIT_FAILURE;
}
```

- `boolexpr ? expr1 : expr2` is conditional expression

Pointers: addresses and dereferencing

Addresses

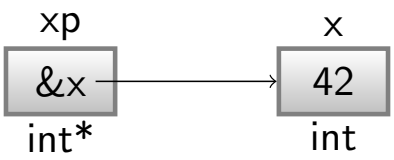
The memory address of a variable, e.g. declared by `T x` where `T` is a type

- can be obtained via the address operator `&`: `&x` is the address of `x`
- `&x` is of type `T *`
- `&x` has a value which is a large (4/8 byte) unsigned integer (usually, we do not want to see it)

Dereferencing

- If a variable `T *xp` has value `&x`, the expression `*xp` dereferences the pointer and refers to `x`, thus has type `T`

```
int x, *xp; xp = &x;
```



Pointers

- if `p` contains the address of a variable, then `*p` allows you to use that variable
- `*p` is treated just like normal variable

```
int a, b, *c, *d;
/* *d = 17; BAD */
d = &b;
*d = 17;
a = 2; b = 3; c = &a; d = &b;
if (*c == *d) printf("Same value\n");
*c = 3;
if (*c == *d) printf("Now same value\n");
c = d;
if (c == d) printf ("Now same address\n");
return 0;
```

void pointers

- generic pointer
- unlike other pointers, can be assigned to any other pointer type:
  - `void *v;`
  - `char *s = v;`
- pointer arithmetic not possible
  - `v++; /*fails */`
- dereferencing void pointers not possible
  - `printf("sizeof(*v)=%lu\n",sizeof(*v)); /*fails */`

Control Structures

- same as Java
- sequencing: ;
- grouping: `{...}`
- selection: `if`, `switch`
- iteration: `for`, `while`

Sequencing and grouping

- `statement1; statement2; statement n;`
  - executes each of the statements in turn
  - a semicolon after every statement
  - not required after a {...} block
- declaration statements
  - variables may be declared only at beginning of block
  - declarations are optional

The `if` statement

- same as Java

```
if (condition_1) {statements_1}
else if (condition_2) {statements_2}
else if (condition_{n-1}) {statements_{n-1}}
else {statements_n}
```

- evaluates statements until there is one with nonzero result
- executes corresponding statements

The if statement

- can omit {}, but be careful

```
#include <stdio.h>
int main(void)
{
  int x = -1,
      y = 1;

  if (x > 0)
    printf("x␣>␣0!");
    if (y > 0)
      printf("x␣and␣y␣>␣0!");

  return 0;
}
```

- indentation is misleading here: inner `if`-statement is always executed as it is not dependent on outer `if`-statement

The switch statement

- allows choice based on a single value

```
switch(expression)
{
  case const_1: statements_1; break;
  case const_2: statements_2; break;
  default: statements_n;
}
```

- effect: evaluates integer expression
- no floating point or string expression allowed
- looks for case with value matching value of integer expression
- executes corresponding statements (or `default`-statement)

The switch statement I

```
typedef enum {Rain, Snow, Sun} Weather_tag;

  Weather_tag w;
  int input;

  if (argc != 2 || sscanf(argv[1],"%d",&input) != 1 ||
      input < 0 || input > (int) Sun)
  {
    fprintf(stderr,"%s␣<weathernum>\nRain=%d,Snow=%d,Sun=%d\n",
            argv[0],(int) Rain,(int) Snow,(int) Sun);
    return EXIT_FAILURE;
  }
  w = (Weather_tag) input;
  switch(w)
  {
    case Rain: printf("bring␣umbrella␣");
    case Snow: printf("wear␣jacket\n"); break;
    case Sun:  printf("wear␣sunscreen\n"); break;
    default:   printf("strange␣weather\n");
  }
```


The switch statement II

```
$ weather_case.x 0
bring umbrella wear jacket
$ weather_case.x 1
wear jacket
$ weather_case.x 2
wear sunscreen
```

Repetition

- C has several control structures for repetition

statement	repeats an action...
`while(c) {}`	zero or more times, while condition is $\neq 0$
`do {...} while(c)`	one or more times, while condition is $\neq 0$
`for (init; condition; update)`	zero or more times, with initialization and update

The `break` statement

- break allows early exit from one loop level

```
for (init; condition; update)
{
  statements1;
  if (condition2) break;
  statements2;
}
```

The continue statement

- continue skips to next iteration, ignoring rest of loop body
- does execute next statement

```
for (init; condition1; next)
{
  statement1;
  if (condition2) continue;
  statement2;
}
```

- often better written as if with block

```
for (init; condition1; next)
{
  statement1;
  if (!condition2)
  {
    statement2;
  }
}
```

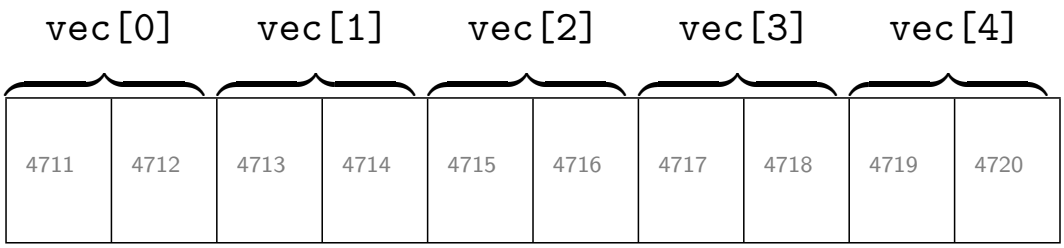

Structured Data

- structured data can be declared in three different ways:

notation	property
`[]`	array of values of same type
`struct`	combination of possibly different types
`union`	combination of possibly different types occupying same space (one of)

Arrays

- arrays are defined by specifying an element type and number of elements
  - `short vec[5];`
  - `char str[30];`
  - `float m[64];`
- for array containing n elements, indexes are $0 \ldots n-1$
- stored as linear sequence of elements at consecutive addresses
- each array element requires bs bytes where bs is the size of the base type in bytes
- example: array `short vec[5]` is stored at address 4711:



Arrays

- C does not keep track of the size of arrays (i.e., no length attribute)
- `int x[10]; x[10] = 5; /*may work (for a while) */`
- in the block where static array `a` is defined:
  - `sizeof a` gives the number of bytes in `a`
  - `sizeof a/sizeof a[0]` gives length of `a`

```
int i, sum = 0, a[] = {1,3,5,7,13};
for(i = 0; i < sizeof a/sizeof a[0]; i++)
  sum += a[i];
```

- when an array is passed as a parameter to a function
  - the size information is not available inside the function
  - array size is typically passed as an additional parameter
    - `printArray(a, VECSIZE);`
  - or as part of a `struct` (best, object-like)
  - or globally
    - `#define VECSIZE 10`

Arrays

- array elements are accessed using the same syntax as in Java: `array[index]`
- example (iteration over array using index `i`): ...

```
int sum = 0, i;
for (i = 0; i < VECSIZE; i++)
{
  sum += vec[i];
}
printf("sum␣of␣numbers␣from␣0␣to␣%d:␣%d\n",
       VECSIZE-1, sum);
```

...

- C does no bounds checking
- e.g. `vec[i-1000]` will not generate a compiler warning
- if you are lucky, the program crashes with `Segmentation fault (core dumped)`
- if you are not lucky, it will produce incorrect results

Arrays

- C references arrays by the address of their first element
- `array` is equivalent to `&array[0]`
- can also iterate through arrays using pointers:

```
int *v, sum = 0;
for (v = vec; v < vec + VECSIZE; v++)
{
  sum += *v;
}
printf("sum␣of␣numbers␣from␣0␣to␣%d:␣%d\n",
       VECSIZE-1, sum);
```

Examples using arrays: partial sums

- consider an array a of length n over some integer base type
- for all $k, 0 \leq k \leq n-1$ the sum $p_k = \sum_{i=0}^{k} a[i]$ is the kth partial sum
- if $a = [2, 4, 1, 5, 0, 5]$, then $[p_0, p_1, \ldots, p_5] = [2, 6, 7, 12, 12, 17]$.
- goal: for all $k, 0 \leq k \leq n-1$ compute the kth partial sum in `a[k]`
- obvious equalities:

$$p_0 = \sum_{i=0}^{0} a[i] = a[0]$$

$$p_k = \sum_{i=0}^{k} a[i] = \left(\sum_{i=0}^{k-1} a[i] \right) + a[k] = p_{k-1} + a[k] \text{ for } k > 0$$

- hence, if we have stored p_{k-1} in $a[k-1]$ (which is the case for $k = 1$), we only have to add $a[k]$ and $a[k-1]$ and store the sum in $a[k]$:

```
unsigned long k;
for (k = 1; k < n; k++)
  a[k] += a[k - 1];
```

```
int *ap;
for (ap = a+1; ap < a+n; ap++)
  *ap += *(ap-1);
```


Examples using arrays: reverse order of array elements (index access)

- suppose we have an `int *arr` with `len>0` elements
- then the following code fragment reverses the order of the array elements

```
unsigned long fwd, bck;

assert(len > 0);
for (fwd = 0, bck = len - 1; fwd < bck; fwd++, bck--)
{
  int tmp = arr[fwd];
  arr[fwd] = arr[bck];
  arr[bck] = tmp;
}
```

- note the condition `len > 0` which is checked by the macro `assert`
- need to include system header `assert.h`

Examples using arrays: reverse order of array elements (pointer access)

- here is some equivalent code using pointers instead of indexes to access the array elements:

```
int *fwd, *bck;

assert(len > 0);
for (fwd = arr, bck = arr+len-1;
     fwd < bck; fwd++, bck--)
{
  int tmp = *fwd;
  *fwd = *bck;
  *bck = tmp;
}
```


Few words on functions

- up until now we have only looked at code fragments
- input data was usually described informally
- in the real world the code appears as part of functions which
  - have a name
  - a list of parameters
  - return type
- here are the complete functions for the partial sums-computation and a function computing the average value of an array of `double`s.

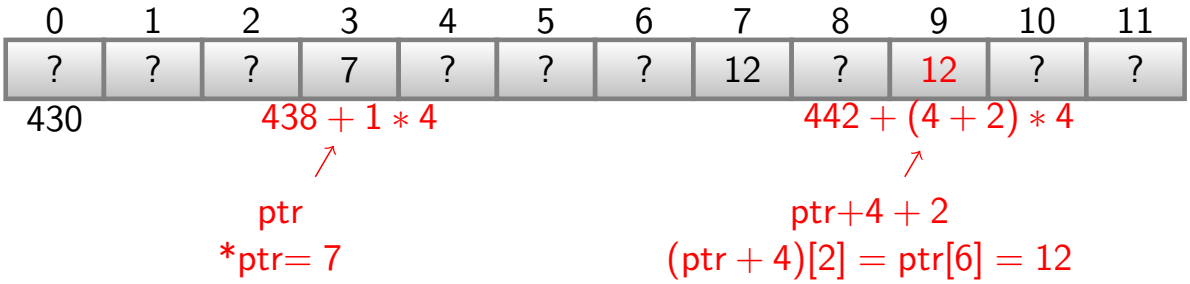
```
static void partialsums(int *a,
                        unsigned long n)
{
  unsigned long k;
  for (k = 1; k < n; k++)
    a[k] += a[k - 1];
}
```

- from now on we mostly show code within functions

```
double average_double(const double *arr,
                      unsigned long len)
{
  const double *ptr;
  double sum = 0.0;

  for (ptr = arr; ptr < arr + len; ptr++)
  {
    sum += *ptr;
  }
  return sum/len;
}
```

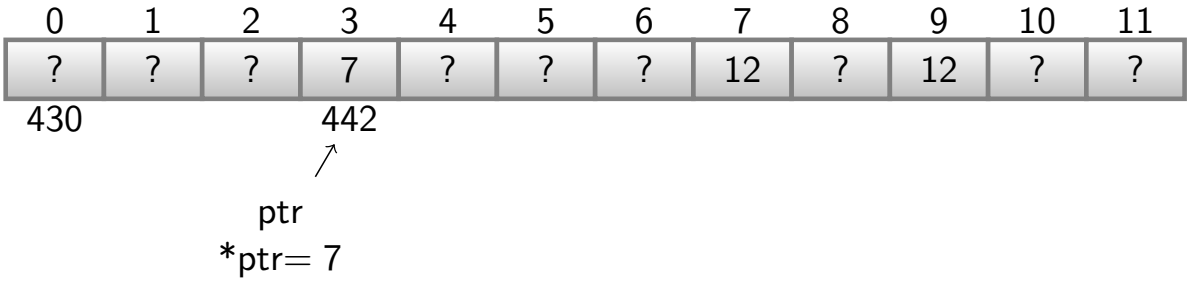
Arrays, pointers and address arithmetic

```
int month[12]; /* month is a pointer to base address, say 430*/

month[3] = 7; /* month address + 3 * sizeof(int)
                 => int at address (430+3*4) becomes 7 */
ptr = month + 2; /* ptr points to month[2],
                    => ptr is now (430 + 2 * sizeof(int))= 438 */
ptr[5] = 12; /* ptr address + 5 * sizeof(int)
                => int at address (438+5*4) becomes 12.
                => month[7] is now 12 */
ptr++; /* ptr becomes 438 + 1 * sizeof(int) = 442 */

(ptr + 4)[2] = 12; /* accessing ptr[6] i.e., month[9] */
```



Arrays, pointers and address arithmetic

- now, `month[6]`, `*(month+6)`, `(month+4)[2]`, `ptr[3]`, `*(ptr+3)` are all the same integer variable at index 6 (which is currently undefined).



Memory allocation

- up until now we have only seen variables referring to data with a fixed size over their lifetime (static data)
- C allows to dynamically create space for variables in memory using
  - `malloc`
  - `calloc`
  - `realloc`
- size of space is determined when variable is created or possibly resized

Memory allocation with malloc

- `void *malloc(size_t size);` returns block of memory of given size
- add `#include <stdlib.h>` when using the function
- `malloc` may fail if address space is occupied ⇒ always check if `NULL` is returned
- lifetime of allocated memory ends once it is freed, with `free`.

```
int *copy_array(const int *arr,unsigned long len)
{
  unsigned long idx;
  int *copy = malloc(sizeof *copy * len);

  if (copy == NULL)
  {
    fprintf(stderr,"malloc(%lu) failed\n",sizeof *copy * len);
    exit(EXIT_FAILURE);
  }
  for (idx = 0; idx < len; idx++)
    copy[idx] = arr[idx];
  return copy;
}
```

Memory allocation with initialization

- note: `malloc` does not initialize data
- `void *calloc(size_t n, size_t size_of_elem)` does initialize (to zero)
- example: the following function returns an array `dist`
- this represents the distribution of values in array `arr`
- `arr` has length `len` and only contains values $\leq$ `maxval`

```
unsigned long *integer_dist(const unsigned long *arr,
                            unsigned long len,
                            unsigned long maxval)
{
  const unsigned long *ptr;
  unsigned long *dist = calloc(maxval + 1,sizeof *dist);

  assert(arr != NULL && dist != NULL);
  for (ptr = arr; ptr < arr + len; ptr++)
    dist[*ptr]++;
  return dist;
}
```


Resizing allocated memory

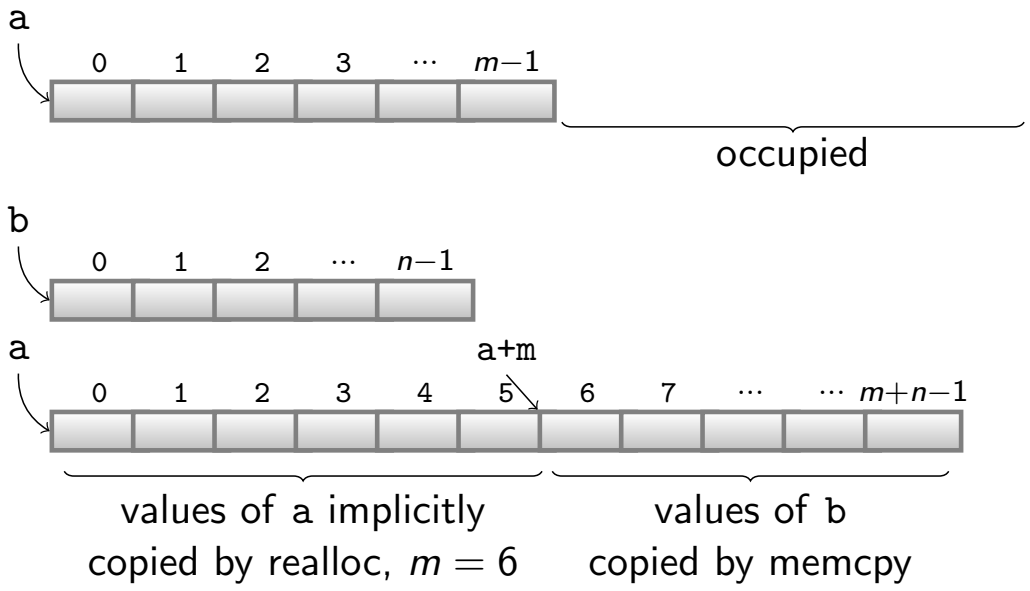
- change size of memory: `void *realloc(void *ptr, size_t size)`
- `ptr` points to existing block, `size` is new size
- new pointer may be different from old, but content is copied

```
int *append_int(int *a,unsigned long m,
                const int *b,unsigned long n)
{
  assert(a + m <= b || b + n <= a);/* no overlaps */
  if (n > 0)
  {
    a = realloc(a,sizeof *a * (m+n));
    memcpy(a+m,b,sizeof *b * n);
  }
  return a;
}
```

- `realloc(NULL,size)` is equivalent to `malloc(size)`

The effect of realloc and memcpy

```
int *append_int(int *a,unsigned long m,
                const int *b,unsigned long n)

⋮
    a = realloc(a,sizeof *a * (m+n));
    memcpy(a+m,b,sizeof *b * n);
```

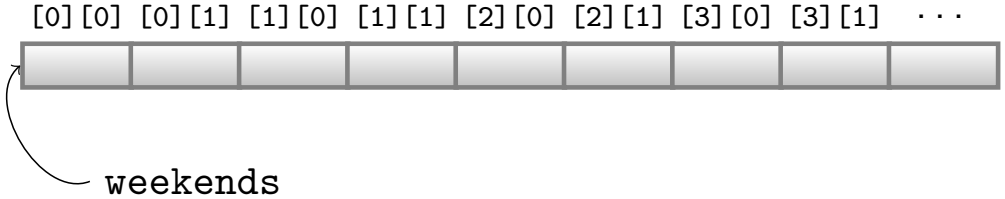


2-D arrays (size at compile time)

```
const int weeks = 52;
const int wedays = 2;
int weekends[weeks][wedays]; /* 52 rows, 2 columns */

printf("sizeof␣weekends=%lu=weeks*wedays*sizeof␣**weekends"
       "␣=␣%d␣*␣%d␣*␣%lu\n",
        sizeof weekends,weeks,wedays,sizeof **weekends);
```

Output:

```
sizeof weekends=416=weeks*wedays*sizeof **weekends = 52 * 2 * 4
```



When accessing e.g. `weekends[w][d]` for some `w` and `d` the compiler determines the index $2 \cdot w + d$ at address $(2 \cdot w + d) \cdot$ `sizeof int`

2-D arrays (size at runtime)

```
/* Allocates a new 2-dimensional array with dimensions <rows> x
   <cols> and returns a pointer to the newly allocated space. */
int **intarray2dim_malloc(unsigned long rows,
                          unsigned long cols)
{
  unsigned long idx;
  int **a2dim = malloc(sizeof *a2dim * rows); /* row pointers */
  a2dim[0] = malloc(sizeof **a2dim * rows * cols); /* values */
  for (idx = 1UL; idx < rows; idx++)
    a2dim[idx] = a2dim[idx-1] + cols;
  return a2dim;
}

void intarray2dim_delete(int **a2dim)
{
  if (a2dim != NULL)
  {
    free(a2dim[0]); /* free array of values */
    free(a2dim);    /* free array of row pointers */
  }
}
```


2-D arrays (size at runtime)

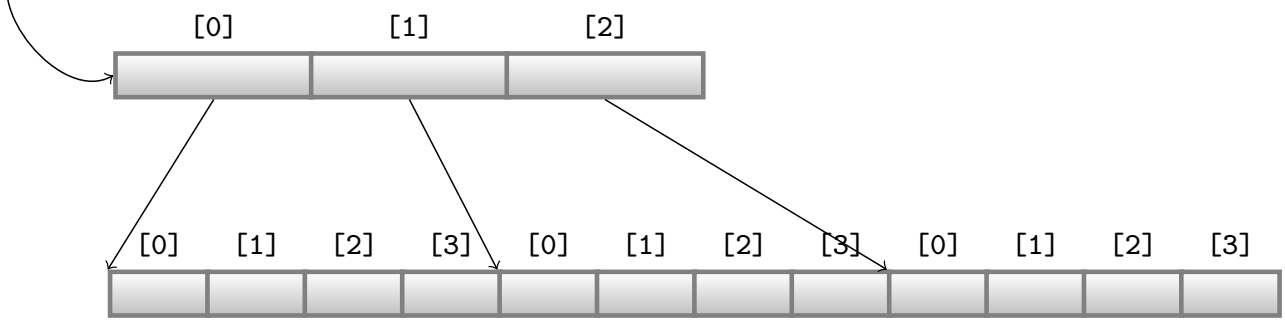
```
void intarray2dim_example(unsigned long m,unsigned long n)
{
  unsigned long r, c;
  int **a2dim;

  /* create m x n int array */
  a2dim = intarray2dim_malloc(m, n);

  /* use a2dim in conventional way via a2dim[r][c] */
  for (r = 0; r < m; r++) {
    for (c = 0; c < n; c++)
      a2dim[r][c] = r * c;
  }
  intarray2dim_delete(a2dim);
}
```

2-D arrays (size at runtime)

The size of the 2-dim array is

```
sizeof (*a2dim) * rows + sizeof (**a2dim) * rows * cols
= sizeof (int *) * rows + sizeof (int) * rows * cols
```

a2dim with 3 rows and 4 columns



- alternative: do not allocate memory for extra column pointers.
- problem: cannot use notation `a2dim[r][c]`

⇒ programmer has to take care of computing the index for given row and column indexes

generic 2-D arrays (size at runtime, via macros)

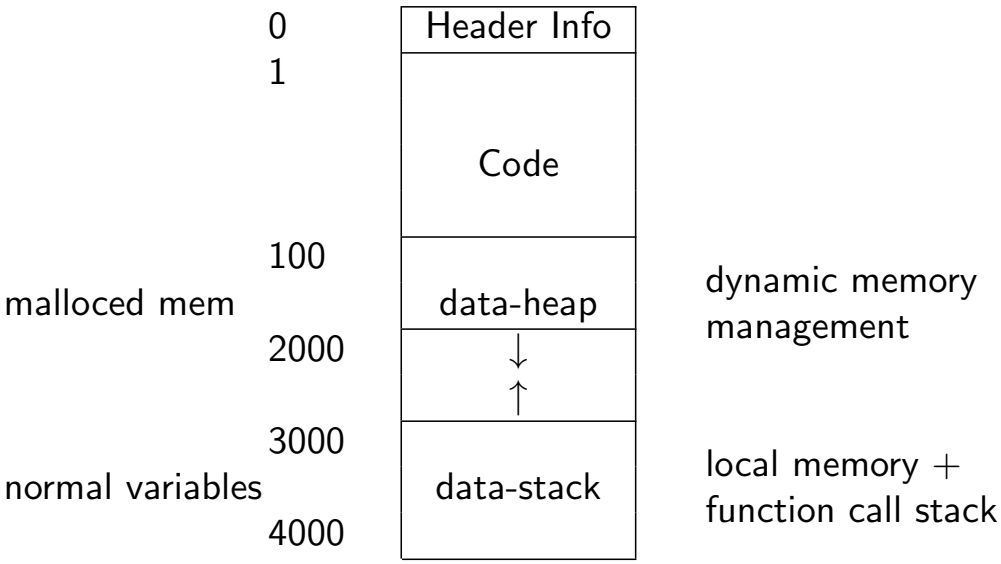
```
/* Allocates a new 2-dimensional array with dimensions
   <ROWS> x <COLS> and assigns a pointer to the newly
   allocated space to <ARR2DIM>. Size of each element is
   determined from type of the <ARR2DIM> pointer. */
#define array2dim_malloc(ARR2DIM, ROWS, COLS)\
        {\
          unsigned long idx;\
          ARR2DIM = malloc(sizeof (*ARR2DIM) * (ROWS));\
          (ARR2DIM)[0]\
            = malloc(sizeof (**ARR2DIM) * (ROWS) * (COLS));\
          for (idx = 1UL; idx < (ROWS); idx++) \
            (ARR2DIM)[idx] = (ARR2DIM)[idx-1] + (COLS);\
        }

#define array2dim_delete(ARR2DIM)\
        if ((ARR2DIM) != NULL)\
        {\
          free((ARR2DIM)[0]);\
          free(ARR2DIM);\
        }
```

2-D arrays (size at runtime)

```
void array2dim_example(unsigned long m,unsigned long n)
{
  unsigned long row, column;
  double **a2dim;

  /* create a m x n double array */
  array2dim_malloc(a2dim, m, n);

  /* use a2dim in conventional way via a2dim[row][column] */
  for (row = 0; row < m; row++) {
    for (column = 0; column < n; column++)
      a2dim[row][column] = 3.14 * row * column;
  }
  array2dim_delete(a2dim);
}
```


Memory layout of programs

- program, dynamic data and static data are stored in different areas of the memory:



- numbers may be units of kilobytes
- height is not proportional to real size

structs

- similar to fields in Java object/class definitions
- components can be any type (recursive only using pointers)
- accessed using the dot-operator `structvar.field`
- example:

```
struct {int x;
        char y;
        float z;} rec;
rec.x = 3;
rec.y = 'a';
rec.z = 3.1415;
```


structs

- struct types can be defined
  - by `struct` definition (see above)
  - by wrapping the `struct` definition inside a `typedef` (see below)
- examples:
  - `struct complex {double real; double imag;};`
  - `typedef struct {double real; double imag;} Complex;`
  - `struct complex a, b;`
  - `Complex c, d;`
- `a` and `b` have the same size, structure and type
- `a` and `c` have the same size and structure, but different types

structs

- overall size is sum of element sizes, plus padding for alignment (to obtain a multiple of the word size)

```
struct {
  char x;
  int y;
  char z;
} s1;

printf("sizeof(s1)=%lu␣bytes\n",
    (unsigned long) sizeof s1); /*sizeof(s1)=12 bytes*/

struct {
  char x, z;
  int y;
} s2;

printf("sizeof(s2)=%lu␣bytes\n",
    (unsigned long) sizeof s2); /*sizeof(s2)=8 bytes*/
```


structs - example

```
struct person {
  char *name;
  int age;
  float height;
  struct { /* embedded structure */
    int month;
    int day;
    int year;
  } birth;
};

struct person me, class[60];
me.birth.year = 1977;
/* array of info about everyone in class */
class[0].name = "Smith"; class[0].birth.year = 1971;

return (me.birth.year > class[0].birth.year) ? 0 : 1;
```

structs with bitfields

- often used to model real memory layout, e.g.,

```
typedef struct {
  unsigned int negative:1;
  unsigned int exponent:8;
  unsigned int mantissa:23;
} IEEEfloat; /* bit representation of float */
```

labeled integers denote size in bits (e.g., `exponent:8`)

Dereferencing pointers to struct elements

- pointers to `structs` are very common:

```
struct person *sp = malloc(sizeof *sp); int y;
(*sp).age = 42; /* * is dereferencing operator */
y = (*sp).age;
```

- note: `*sp.element` does not work, as it means `*(sp.element)`
- reason: infix operator `.` has higher precedence than prefix operator `*`
- abbreviated and preferred alternative: use infix operator `->` for accessing a structure component (right argument) referred to by a pointer (left argument)

```
sp->age = 42;
y = sp->age;
```

Example: parsing and storing dates I

```
typedef struct
{
  unsigned int year;
  unsigned char month, day;
} Date;

Date *date_tab_parse(unsigned long *numofdates)
{
  int y, m, d; /* dates have format yyyy-mm-dd */
  unsigned long ln, nextfree = 0, allocated = 0;
  Date *date_tab = NULL, *date_ptr;

  /* read from stdin */
  for (ln = 1UL; scanf("%d-%d-%d\n",&y,&m,&d) == 3; ln++)
  {
    if (y < 0 || m < 1 || m > 12 || d < 1 || d > 31)
    {
      fprintf(stderr,"line␣%lu:␣illegal␣number␣in␣date\n",ln);
      exit(EXIT_FAILURE);
    }
```


Example: parsing and storing dates II

```
    if (nextfree >= allocated)
    {
      allocated += 128; /* constant increment */
      date_tab = realloc(date_tab,
                         sizeof *date_tab * allocated);
    }
    date_ptr = date_tab + nextfree;
    date_ptr->year = (unsigned int) y;
    date_ptr->month = (unsigned char) m;
    date_ptr->day = (unsigned char) d;
    nextfree++;
  }
  *numofdates = nextfree;
  return date_tab;
}
```

Unions

- like structs but with keyword `union`:

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

- all components of `union` occupy same memory space
- can hold different types at different times
- overall size is largest of elements (`char *` in the union above)
- for the example above:
  - 4 bytes on 32-bit machines
  - 8 bytes on 64-bit machines

Functions

Why?

- division of long program into smaller manageable parts
- modularization simplifies coding and debugging
- improvement of reusability of code

How?

- abstract program code via parameters to functions which are passed
  - by value
  - by reference
- return values to the calling function
  - as value
  - as reference

Functions

- prototypes and functions (cf. Java interfaces)

```
int putchar(int c); /* forward declaration: prototype */
(void) putchar((int) 'A');  /* function call */
int putchar(int c)  /* implementation */
{
  /* do something interesting here */
}
```

- if defined before use in same file, no need for prototype
- typically, prototype defined in `.h` file
- always include `.h` file in actual definition file to allow compiler detecting inconsistencies between function headers (of forward declaration) and actual declaration
- compiler will produce warning if function is not declared before and prototype of function is missing

Functions

- static functions and variables are only accessible in the file they appear in

```
static int x; /* only accessible in this file, global var */
static int times2(int c) /* only accessible in this file*/
{
  return c * 2;
}
```

- compare protected class members in Java classes

Functions – const arguments

- indicates that argument will not be changed
- used for pointer arguments:

```
size_t mystrlen(const char *s)
{
  const char *sp; /* const as string will not change */

  for (sp = s; *sp != '\0'; sp++)
    /* Nothing */ ;
  return (size_t) (sp - s);
}
```

- attempts to change array referenced by pointer ⇒ compiler warning

```
void encode_inplace(char *s)
{
  char *sp;

  for (sp = s; *sp != '\0'; sp++)
    *sp = *sp + 1; /* => no const parameter */
}
```


Functions – const arguments

```
int main(void)
{
  char *s;

  printf("filename␣\"%s\"␣has␣length␣%lu\n",__FILE__,
           (unsigned long) mystrlen(__FILE__));
  s = strdup(__FILE__); /* dup as it will be modified */
  printf("encode_inplace(%s)=",s);
  encode_inplace(s);
  printf("%s\n",s);
  free(s);
  return EXIT_SUCCESS;
}
```

Output

```
filename "mystrlen.c" has length 10
encode_inplace(mystrlen.c)=nztusmfo/d
```

Pointers and functions: what does this C program do?

```
typedef struct Listelem Listelem;

struct Listelem
{
  int data;
  struct Listelem *next;
};

void list_add(Listelem *head, Listelem *tail, int data)
{
  if (tail == NULL) /* empty list */
  {
    head = tail = malloc(sizeof *tail);
    head->data = data; head->next = NULL;
  } else /* non-empty list */
  {
    tail->next = malloc(sizeof *tail);
    tail = tail->next; tail->data = data; tail->next = NULL;
  }
}
```


What does this C program do – cont'd?

```
int list_delete_first(Listelem *head, Listelem *tail)
{ /* return data of first list-element and delete it */
  if (head != NULL)
  {
    int data = head->data;
    Listelem *temp = head->next;
    free(head); head = temp;
    return data;
  }
  return -1;
}

int main(void)
{
  Listelem *start = NULL, *end = NULL;

  list_add(start, end, 2); list_add(start, end, 3);
  printf("First␣element:␣%d\n", delete(start, end));
  return EXIT_SUCCESS; /* prints First element: -1 */
}
```

Pointers and functions: what does this C program do?

- problem mainly with `add`: assignments to `head` and `tail` does not have an effect outside of function
- reason: parameters are passed as values, i.e. the value of each parameter is copied and inside the function only the copy can be accessed

$\Rightarrow$ the pointers are copies of addresses of the corresponding variables `start` and `end` in `main`

- modification of the values of the copies only has an effect inside the function
- first solution to fix problem: use pointer to pointer, but this would be difficult to read
- better solution: combine `head` and `tail` in a `struct`

The corrected program

```
typedef struct
{
  Listelem *head, *tail;
} List;

void list_add(List *l, int data)
{
  if (l->tail == NULL)
  {
    l->head = l->tail = malloc(sizeof *(l->tail));
    l->head->data = data; l->head->next = NULL;
  } else
  {
    l->tail->next = malloc(sizeof *(l->tail));
    l->tail = l->tail->next;
    l->tail->data = data;
    l->tail->next = NULL;
  }
}
```

The corrected program

```
List *list_new(void)
{
  List *l = malloc(sizeof *l);
  l->head = l->tail = NULL;
  return l;
}

int list_delete_first (List *l)
{ /* return data of first list-element and delete it */
  if (l != NULL && l->head != NULL)
  {
    Listelem *temp = l->head->next;
    int data = l->head->data;
    free(l->head);
    l->head = temp;
    return data;
  }
  return -1; /* empty list */
}
```


The corrected program

```
void list_delete(List *l)
{
  if (l != NULL)
    free(l);
}

int main(void)
{
  List *l = list_new();

  list_add(l, 2);
  list_add(l, 3);
  printf("First␣element:␣%d\n", list_delete_first(l));
  while (list_delete_first(l) >= 0) /* Nothing */;
  list_delete(l);
  return 0;
}
```

Output:

```
First element: 2
```

Program with multiple files

- to reuse functions in different contexts (e.g. `main`-programs) these should be implemented in an own file:

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "mypgm.h"
int main(void) {
  myproc();
  return EXIT_SUCCESS;
}
```

mypgm.c

```c
#include <stdio.h>
#include "mypgm.h"
void myproc(void) {
  /* ... some code */
}
```

mypgm.h

```c
void myproc(void);
```

  - library headers
    - <Standard>
    - "User-defined"

Data hiding in C: a concrete example

- C does not have classes or private members, but this can be approximated
- implementation in `list.c` defines real data structure:

```c
struct List
{
  struct List *next;
  int data;
};

List *list_new(void) /* use type List from list.h */
{
  return NULL;
}

void list_show(const List *lst)
{
  const List *ptr;

  for (ptr = lst; ptr != NULL; ptr = ptr->next)
    printf("%d\n",ptr->data);
}
```

Data hiding in C (continued)

```
static List *list_newelem(int data)
{
  List *elem = malloc(sizeof(*elem));

  assert(elem != NULL);
  elem->next = NULL;
  elem->data = data;
  return elem;
}

List *list_append(List *lst,int data)
{
  if (lst == NULL)
  {
    lst = list_newelem(data);
  } else
  {
    List *ptr;
    for (ptr = lst; ptr->next != NULL; ptr = ptr->next)
      /* Nothing */;
    ptr->next = list_newelem(data);
  }
  return lst;
}
```


Data hiding in C (continued)

```
void list_delete(List *lst)
{
  List *ptr = lst;

  while (ptr != NULL)
  {
    List *tmp = ptr;

    ptr = ptr->next;
    free(tmp);
  }
}
```

Data hiding in C: an example for lists

- header file `list.h` defines public data:

```
#ifndef LIST_H /* some identifier only used here */
#define LIST_H
typedef struct List List;
List *list_new(void);
List *list_append(List *lst,int data);
void list_show(const List *lst);
void list_delete(List *lst);
#endif
```

- and finally `testlist.c` implements a test in `main`:

```
#include "list.h"
int main(void)
{
  int data;
  List *lst = list_new();
  for (data = 1; data<10; data++)
    lst = list_append(lst,data);
  list_show(lst);
  list_delete(lst);
  return 0;
}
```


Function without return value and void-pointer

- function that does not return anything is declared as `void`
- constant functions have a `void` argument list
- special pointer `*void` can point to anything

```
#include <stdlib.h>
#include <stdio.h>
void *fun(void)
{
  printf("the␣big␣void\n");
  return NULL;
}
int main(void)
{
  (void) fun();
  return EXIT_SUCCESS;
}
```

Overloading functions – var. arg. list

- Java:

```
void product(double x, double y);
void product(vector x, vector y);
```

- C does not support this, but allows variable number of arguments:

```
debug("%d␣%f", x, f);
debug("%c", c);
```

- declared as

```
void debug(char *fmt, ...);
```

- at least one known argument (`fmt` here)
- the three dots

  `...`

  denote the variable arguments which are accessed via macros, described next

Overloading functions: example

```
#include <stdarg.h>  /* necessary to include */
double product(int number, ...)
{
  va_list list;
  double p = 1.0;
  int i;

  va_start(list, number); /* arg to start with */
  for (i = 0; i < number; i++)
    p *= va_arg(list, double); /* access double parameters
                                  following number */
  va_end(list); /* no further args */
  return p;
}
```

- note: `product(2,3,4);` won't work, but `product(2,3.0,4.0);`
- `product(3,3.14,4.2,5.4);` also works

Function pointers

- task: a complicated computation delivers some result and this should be processed in different ways
- problem: we want to be flexible in how it is processed
- solution: decouple program code computing a particular result (e.g. in an inner loop) from code further processing the result

⇒ use function pointers

- `returnType (*ptrName)(arg1, arg2, ..., argn);`
- for example, `int (*fp)(double)` is a pointer `fp` to a function that returns an integer and has one double argument
- `double * (*gp)(int)` is a pointer to a function that returns a pointer to a double and has one `int` argument
- `int func(); /*function returning integer*/`
- `int *func(); /*function returning pointer to integer*/`
- `int (*func)(); /*pointer to function returning integer*/`
- `int *(*func)(); /*pointer to function returning pointer to int*/`

Function pointers

```
#include <math.h>
#include <assert.h>

int f(void)
{
  return 3;
}

double *g(int val)
{
  double *d = malloc(sizeof *d);
  assert(d != NULL);
  *d = sqrt((double) val);
  return d;
}
```

- after compilation of this program code, the object code for `f` and `g` is stored in the program segment (see slide 83)

Function pointers

```
int main(void)
{
  int i, (*fp)(void);
  double *d, * (*gp)(int);

  fp = f; /* pointer to program segment in RAM */
  gp = g; /* pointer to program segment in RAM */
  i = fp();
  i++;
  printf("i=%d\n",i);
  d = gp(17);
  printf("gp(17)=%.2f\n",*d);
  free(d);
  return EXIT_SUCCESS;
}
```

Output:

```
i=4
gp(17)=4.12
```


Function pointers as function arguments

```
#include <stdio.h>

void myproc (int d);
void mycaller(void (*f)(int), int param);

int main(void)
{
  myproc(10);            /* call myproc with parameter 10*/
  mycaller(myproc, 10); /* and do the same again */
  return 0;
}
void mycaller(void (*f)(int), int param)
{
  f(param); /* call function f with param */
}
void myproc (int d)
{
  printf("%d\n", d); /* ... do something with d */
}
```

Function pointers (replace a switch statement)

problem: apply a scalar value to each entry of matrix; application can be addition, subtraction, multiplication, ...

```
void applyscalar_via_switch(double **matrix,double scalar,
                            unsigned long rows,
                            unsigned long cols,char opCode)
{
  unsigned long r, c;

  for (r = 0; r < rows; r++)
    for (c = 0; c < cols; c++)
      switch(opCode)
      {
        case '+' : matrix[r][c] += scalar; break;
        case '-' : matrix[r][c] -= scalar; break;
        case '*' : matrix[r][c] *= scalar; break;
      }
}
```

each additional operation (e.g. division) would require to extend the switch statement: `case '/': matrix[r][c] /= scalar; break;`

Function pointers (replace a switch statement)

abstract from how scalar is applied to matrix entry by using function pointer[1]

```
void applyscalar_via_funptr(double **matrix,double scalar,
                            unsigned long rows,
                            unsigned long cols,
                            double (*pt2func)(double,double))
{
  unsigned long r, c;

  for (r = 0; r < rows; r++)
    for (c = 0; c < cols; c++)
      matrix[r][c] = pt2func(matrix[r][c], scalar);
}
```

- more flexible than switch: no change in `apply_scalar_via_funptr` necessary to satisfy new requirements

[1] Example adapted from `http://www.newty.de/fpt/index.html`

Function pointers (replace a switch statement)

```
double plus    (double a, double b) { return a+b; }
double minus   (double a, double b) { return a-b; }
double multiply(double a, double b) { return a*b; }

int main(void)
{
  unsigned long r, c;
  const unsigned long rows = 1000, cols = 500;
  double **matrix;

  array2dim_malloc(matrix, rows, cols);
  for (r = 0; r < rows; r++)
    for (c = 0; c < cols; c++)
      matrix[r][c] = drand48();
  applyscalar_via_switch(matrix,3.14,rows,cols,'+');
  applyscalar_via_funptr(matrix,3.14,rows,cols,minus);
  applyscalar_via_funptr(matrix,3.14,rows,cols,multiply);
  array2dim_delete(matrix);
  return EXIT_SUCCESS;
}
```


Function pointers (call back functions) I

some function from standard C-library need function as parameter example:

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

- `compar` is a pointer to a function with two `const void`-pointers as arguments, returning an `int`
- `qsort` implements the comparison based Quicksort-algorithm which performs the comparison using the function given as last argument
- in the implementation of `qsort`, a comparison of the *i*th and the *j*th element in the array is performed by calling

```
compar((const void *) (((const char *) base) + width*i),
       (const void *) (((const char *) base) + width*j));
```

Function pointers (call back functions) II

- so if `double` or `Person` is the base type of the array declare functions like these:

```
int doublecmp(const void *aptr, const void *bptr)
{
  double a = *((const double *) aptr);
  double b = *((const double *) bptr);

  return (a < b) ? -1
                 : ((a > b) ? 1 : 0);
}

typedef struct { char *name; int age; } Person;

int personcmp(const void *aptr, const void *bptr) {
  const Person *pa = (const Person *) aptr;
  const Person *pb = (const Person *) bptr;

  return strcmp(pa->name,pb->name);
}
```


Function pointers (call back functions)

```
int main(int argc,char *argv[])
{
  int idx, len;
  double *arr;
  if (argc != 2 || sscanf(argv[1],"%d",&len) != 1 || len <= 0)
  {
    fprintf(stderr,"Usage:␣%s␣<positive␣number>\n",argv[0]);
    exit(EXIT_FAILURE);
  }
  arr = malloc(sizeof (*arr) * len);
  assert(arr != NULL);
  srand48(366292341); /* init random number generator */
  for (idx =0;idx<len;idx++)
    arr[idx] = drand48();
  qsort(arr, (size_t) len, sizeof *arr, doublecmp);
  for (idx=1;idx<len;idx++) /* check for correct order */
    assert(arr[idx-1] <= arr[idx]);
  free(arr);
  exit(EXIT_SUCCESS);
}
```

Function pointers (call back functions)

- here is another application of `qsort` for sorting dates:

```
int date_compare(const void *a,const void *b)
{
  const Date *da = (const Date *) a;
  const Date *db = (const Date *) b;

  if (da->year < db->year)
    return -1;
  if (da->year > db->year)
    return 1;
  if (da->month < db->month)
    return -1;
  if (da->month > db->month)
    return 1;
  if (da->day < db->day)
    return -1;
  if (da->day > db->day)
    return 1;
  return 0;
}
```


Function pointers (call back functions)

```
int main(void)
{
  unsigned long numofdates;
  Date *date_tab, *date_ptr;

  date_tab = date_tab_parse(&numofdates); /* from stdin */
  qsort(date_tab,numofdates,sizeof *date_tab,date_compare);
  for (date_ptr = date_tab; date_ptr < date_tab + numofdates;
       date_ptr++)
  {
    printf("%u-%02u-%02u\n",date_ptr->year,
                            date_ptr->month,
                            date_ptr->day);
  }
  free(date_tab);
  return EXIT_SUCCESS;
}
```

Libraries

- C provides a set of standard libraries for

name	header file	loader option
input/output	`<stdio.h>`	
character types	`<ctype.h>`	
character strings	`<string.h>`	
numerical math functions	`<math.h>`	`-lm`
threads	`<pthread.h>`	`-lpthread`

- functions for input/output as well as functions to manipulate character types and strings are part of the standard C-library
- this is is loaded anyway
- there are many more libraries (e.g. $\approx$ 500 on the Linux systems at the ZBH)

The math library

- `#include <math.h>`
  - careful: `sqrt(5)` without header file may give wrong result!
- `gcc -Wall -Werror -o compute.x main.o f.o -lm`
- uses normal mathematical notation:

Java	C
`Math.sqrt(2)`	`sqrt(2)`
`Math.pow(x,5)`	`pow(x,5)`
`4*math.pow(x,3)`	`4*pow(x,3)`

Characters

- the type `char` is a byte storing ASCII code values (e.g., 65 means 'A', 66 means 'B', ...)

⇒ computation involving characters can treat them as numbers

- the visible character 'A' is just an interpretation of integer 65.
- e.g. `dist[(int) 'A']++;` increments the entry at index 65 in array `dist`
- `ctype.h` declares character classification functions:

`isalnum(ch)`	alphanumeric	`[a-zA-Z0-9]`
`isalpha(ch)`	alphabetic	`[a-zA-Z]`
`isdigit(ch)`	digit	`[0-9]`
`ispunct(ch)`	punctuation	`[~!@#%^&...]`
`isspace(ch)`	white space	`[ \t\n]`
`isupper(ch)`	upper-case	`[A-Z]`
`islower(ch)`	lower-case	`[a-z]`

Strings

- in Java, strings are regular objects
- in C, strings are just char arrays with a '\0' terminator
- `"a␣cat"`=

- a literal string `"a␣cat"`
  - is automatically allocated memory space (in the program segment) to contain the string including the terminating \0
  - has a value which is the address of the first character
  - cannot be changed by the program (common bug!)
- for all strings which are not literals, the programmer is responsible to allocate the appropriate space
  - statically at compile time: `char seq[MAXLEN+1];`
  - dynamically at run time:

```
char *seq = malloc(sizeof *seq * (maxlen+1));
```

Strings

```
char *capitalize(char *s)
{
  s[0] = toupper(s[0]); /* modify first character */
  return s;
}

int main(void)
{
  printf("%s\n",capitalize("a␣cat"));
  return EXIT_SUCCESS;
}
```

- this will lead to a bus error, because the string `"a␣cat"` from the program segment cannot be modified.
- the following will work, as it uses a copy of the string.

```
char *s = strdup("a␣cat");  /* duplicated string */
printf("%s\n",capitalize(s));
free(s);
```


Strings

- we normally refer to a string via a pointer to its first character:

```
char *str = "my␣string";
char *s;
s = &str[0];
s = str; /* equivalent to previous */
```

- we can access the characters via index-access:

```
char *str = "my␣string";
unsigned long i;
for (i = 0; str[i] != '\0'; i++)
  putchar(str[i]);
```

- or via pointer-access:

```
char *s;
for (s = str; *s != '\0'; s++)
  putchar(*s);
```

Copying strings

- copying content vs. copying pointer to content

  `char *s, t[MAXLEN+1]; strcpy(t,"aatta");`

  `s = t` copies the pointer – `s` and `t` now refer to the same memory location

```
s = malloc(sizeof *s * (MAXLEN+1));
strcpy(s, t);
```

  copies content of `t` to `s`

- the following is incorrect (but appears to work!)

```
char mybuffer[100];
mybuffer = "a␣cat";
```

- use `strcpy(mybuffer, "a␣cat")` instead

Copying strings and memory leaks

- be careful with memory implicitly allocated by other functions
- using these functions often leads to memory leaks (i.e. allocated memory is never freed)

```
char *mystrdup(const char *s)  /* equivalent to strdup */
{
  char *u = (char *) malloc(strlen(s) + 1);
  assert(u != NULL);
  strcpy(u, s);
  return u;
}

void showcapitalized(const char *s)
{
  char *copy = mystrdup(s);
  copy[0] = toupper(copy[0]);
  printf("%s\n",copy); /* space copy refers to leaks */
}
```

Library for string-functions

- assumptions:
  - strings are \0-terminated
  - all target arrays are large enough
- `#include <string.h>`

functions:

- `char *strcpy(char *dest, char *source)`
  - copies chars from source array into dest array up to \0
- `char *strncpy(char *dest, char *source, int num)`
  - copies chars; stops after num chars if no \0 before that; appends \0
- `int strlen(const char *source)`
  - returns number of chars, excluding NUL
- `char *strchr(const char *source, int ch)`
  - returns pointer to first occurrence of ch in source; `NULL` if none
- `char *strstr(const char *source, const char *search)`
  - return pointer to first occurrence of search in source

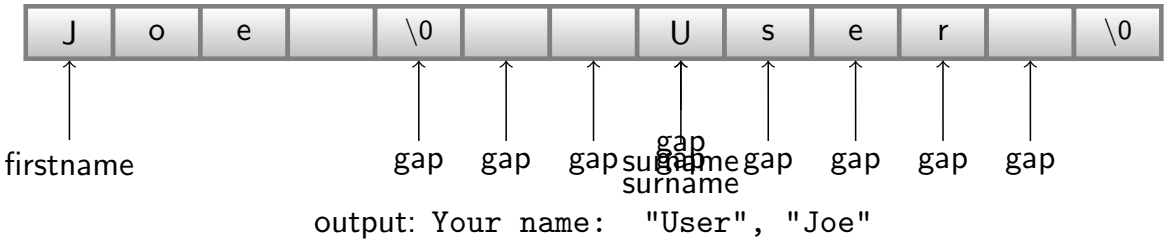
Example string manipulation: exchange firstname/surname

```
char line[100];
char *surname, *firstname, *gap;

printf("Enter␣firstname␣lastname:␣");
if ((firstname = fgets(line,sizeof line,stdin)) == line)
{ /* successful read */
  firstname = line;
  if ((gap = strchr(line,'␣')) == NULL)
    fprintf(stderr,"no␣space␣after␣firstname");
  else {
    *gap = '\0'; /* replace space by \0 */
    gap++; /* start scanning character after \0 */
    while (isspace(*gap)) gap++; /* skip over spaces */
    surname = gap; /* found surname */
    while (!isspace(*gap)) gap++; /* skip over non-spaces */
    *gap = '\0'; /* end of surname */
    printf("Your␣name:␣\"%s\",␣\"%s\"\n",surname,firstname);
  }
}
return 0;
```

```
$ string_manipulate.x
Enter firstname lastname: Joe   User
Your name: "User"  "Joe"
```

Example string manipulation: exchange firstname/surname

```
firstname = line;                                                    ⇐
if ((gap = strchr(line,' ')) == NULL)                                ⇐
  fprintf(stderr,"no space after firstname");
else {
    *gap = '\0'; /*replace space by \0 */                            ⇐
    gap++; /*start scanning character after \0 */                   ⇐
    while (isspace(*gap)) gap++; /*skip over spaces */               ⇐
    surname = gap; /*found surname */                               ⇐
    while (!isspace(*gap)) gap++; /*skip over non-spaces */          ⇐
    *gap = '\0'; /*end of surname */                                 ⇐
    printf("Your name: \"%s\", \"%s\"\n",surname,firstname);
}
```



output: `Your name: "User", "Joe"`

Formatted strings

- string parsing: `int sscanf(char *string, char *format, ...)`
  - parse the contents of string according to format
  - placed the parsed items into 3rd, 4th, 5th, ... argument
  - return number of successful conversions
- string formatting: `int sprintf(char *buffer, char *format, ...)`
  - produce a string formatted according to format
  - place this string into the buffer
  - the 3rd, 4th, 5th, ... arguments are formatted
  - return number of successful conversions
- format strings for `sscanf` and `sprintf` contain
  - plain text (matched on the input or inserted into the output)
  - formatting codes (which must match the arguments)
- `sprintf` format string gives template for result string
- `sscanf` format string describes what input should look like

Formatted strings

- Formatting codes for `sscanf`

code	meaning (matches a...)	variable type
`%c`	single character	char
`%d`	integer (optionally signed) in decimal	int
`%u`	unsigned integer in decimal	unsigned int
`%f`	real number (ddd.dd)	float
`%s`	string up to white space	char *

- Modifiers

code	meaning	example
`l`	larger version of given type	`%lu`
`h`	shorter version of given type	`%hd`

- see `man sscanf` for a complete list

Formatted strings

- formatting codes for `sprintf`
- values normally right-justified; use negative field width to get left-justified

Code	meaning	variable type
`%nc`	char in field of `n` spaces	`char`
`%nd`	integer in field of `n` spaces	`int`, `long`
`%n.mf`	real number in width `n`, `m` decimals	`float`, `double`
`%n.mg`	real number in width `n`, `m` digits of precision	`float`, `double`
`%n.ms`	first `m` chars from string in width `n`	`char *`

Formatted strings - examples

```
char *msg = "Hello␣there";
char *nums = "1␣3␣5␣7␣9";
char s[10], t[10];
int a, b, c, n;

n = sscanf(msg, "%s␣%s", s, t);
n = printf("%10s␣%-10s", t, s);
n = sscanf(nums, "%d␣%d␣%d", &a, &b, &c);

printf("%d␣flower%s", n, n > 1 ? "s␣" : "␣");
printf("a␣=␣%d,␣answer␣=␣%d\n", a, b+c);
```

- output:

```
there Hello     3 flowers a = 1, answer = 8
```


The stdio library

- use `#include <stdio.h>` for prototypes
- no compiler flag necessary: compiler links it automatically
- defines `FILE *` type and functions of that type
  - allow read/write access to files in file system
  - represent buffered stream of chars (bytes) to be written or read
- always defines `stdin`, `stdout`, `stderr`

The stdio library: fopen(), fclose()

- Opening and closing `FILE *` streams:
  - `FILE *fopen(const char *path, const char *mode)`
    - open the file called path in the appropriate mode
    - modes: `"r"` (read), `"w"` (write), `"a"` (append), `"r+"` (read & write)
    - include character `b` in mode when to prevent OS-dependent behavior
    - returns a new `FILE *` if successful, `NULL` otherwise
  - `int fclose(FILE *stream)`
    - close the stream FILE *
    - return 0 if successful, EOF if not

stdio – character I/O

- `int getchar()`
  read the next character from `stdin`; returns `EOF` if none
- `int fgetc(FILE *fpin)`
  read the next character from `fpin`; returns `EOF` if none
- `int putchar(int c)`
  write the character `c` onto `stdout`; returns `c` or `EOF` (mostly ignored)
- `int fputc(int c, FILE *fpout)`
  write the character `c` onto `fpout`; returns `c` or `EOF` (mostly ignored)
- `int fread(void *ptr,size_t s,size_t n, FILE *fpin);`
  read `n` items, each of size `s` from `fpin` and store it in memory area referenced by `ptr`; returns number of successfully read items
- `int fwrite(void *ptr,size_t s,size_t n, FILE *fpout);`
  write `n` items, each of size `s` stored in memory area referenced by `ptr` to `fpout`; returns number of successfully written items

stdio – an example computing character distributions

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <ctype.h>

static void distribution_update(unsigned long *dist,
                                const char *progname,
                                const char *filename)
{
  int cc;
  FILE *fp = fopen(filename,"rb");
  if (fp == NULL)
  {
    fprintf(stderr,"%s: cannot open %s\n",progname,filename);
    exit(EXIT_FAILURE);
  }
  while ((cc = fgetc(fp)) != EOF)
    dist[cc]++;
  (void) fclose(fp);
}
```


stdio – an example computing character distributions

```
static void distribution_show(const unsigned long *dist)
{
  int i;

  for (i = 0; i <= UCHAR_MAX; i++)
  {
    if (dist[i] > 0)
    {
      if (iscntrl(i))
        printf("\\%d: %lu\n",i,dist[i]);
      else
        printf("%c: %lu\n",(char) i,dist[i]);
    }
  }
}
```

stdio – an example computing character distributions

```
int main(int argc, char *argv[])
{
  int i;
  unsigned long dist[UCHAR_MAX+1] = {0};

  if (argc < 2)
  {
    fprintf(stderr,"Usage: %s <infile1> ... <infilen>\n",
            argv[0]);
    exit(EXIT_FAILURE);
  }
  for (i = 1; i < argc; i++)
  {
    distribution_update(dist,argv[0],argv[i]);
  }
  distribution_show(dist);
  exit(EXIT_FAILURE);
}
```


stdio — Determining files sizes

- moving file pointer on file

```
int fseek(FILE *f,
          long distance,
          int origin)
```

  - set file offset to an arbitrary value
  - distance from origin; can be negative
  - origin, which can take one of the values:
    - SEEK_SET: start of file
    - SEEK_CUR: current offset
    - SEEK_END: end of file

- deliver the current file offset:

```
long ftell(FILE *f)
```

```
unsigned long file_size(
                const char *inputfile)
{
  long filesize;
  FILE *fp = fopen(inputfile,"rb");

  assert(fp != NULL);
  /* jump to end of file: */
  fseek (fp, 0, SEEK_END);
  /* read current position: */
  filesize = ftell(fp);
  fclose(fp);
  assert(filesize >= 0);
  return (unsigned long) filesize;
}
```

stdio – line I/O

- `char *fgets(char *buf, int size, FILE *fp)`
  - read next line from `fp` into buffer `buf`
  - halts at `'\n'` or after `size-1` characters have been read
  - the `'\n'` is read, but not included in buf
  - returns pointer to `buf` if ok, `NULL` otherwise
- do **not** use `gets(char *)` – possible buffer overflow
- `int fputs(const char *str, FILE *fp)`
  - writes the string `str` to `out`, stopping at '\0'
  - returns number of characters written or EOF

stdio – an example merging files

```
typedef struct
{
  FILE *fp;   /* reference to file */
  char *line; /* buffer for line in file referred to by fp */
  bool valid; /* file can still be processed */
} Fileinfo;

static unsigned long find_smallest_line(const Fileinfo *finfo_tab,
                                        unsigned long numoffiles)
{
  unsigned long f, smallest = numoffiles; /* undefined */

  for (f = 0; f < numoffiles; f++)
  {
    if (finfo_tab[f].valid &&
        (smallest == numoffiles ||
         strcmp(finfo_tab[f].line,finfo_tab[smallest].line) < 0))
    {
      smallest = f;
    }
  }
  return smallest;
}
```

stdio – an example merging files

```
static void mergefiles(char **filelist,int numoffiles,
                       unsigned long maxline)
{
  Fileinfo *finfo_tab = malloc(sizeof *finfo_tab * numoffiles);
  unsigned long f, open_files = 0;

  for (f = 0; f < numoffiles; f++)
  {
    finfo_tab[f].line = malloc(sizeof(char) * (maxline + 1));
    finfo_tab[f].fp = fopen(filelist[f],"r");
    if (finfo_tab[f].fp == NULL)
    {
      fprintf(stderr,"cannot␣open␣file␣%s\n",filelist[f]);
      exit(EXIT_FAILURE);
    }
    finfo_tab[f].valid = true;
  }
  open_files = numoffiles;
```


stdio – an example merging files

```
  for (f = 0; f < numoffiles; f++)
  {
    if (fgets(finfo_tab[f].line,maxline,finfo_tab[f].fp) == NULL)
    {
      finfo_tab[f].valid = false;
      assert(open_files > 0);
      open_files--;
    }
  }
  while (open_files > 0)
  {
    unsigned long smallest = find_smallest_line(finfo_tab,numoffiles);
    assert(smallest < numoffiles);
    printf("%s",finfo_tab[smallest].line);
    if (fgets(finfo_tab[smallest].line,maxline,
              finfo_tab[smallest].fp) == NULL)
    {
      finfo_tab[smallest].valid = false;
      open_files--;
    }
  }
  for (f = 0; f < numoffiles; f++)
    free(finfo_tab[f].line);
  free(finfo_tab);
}
```

stdio – an example merging files

```
int main(int argc,char *argv[])
{
  if (argc <= 2)
  {
    fprintf(stderr,
            "Usage:␣%s␣<file_1>␣<file_2>␣[...file_n]\n",
            argv[0]);
    exit(EXIT_FAILURE);
  }
  mergefiles(argv + 1,argc - 1,1024UL);
  exit(EXIT_SUCCESS);
}
```


stdio – formatted I/O

- `int fscanf(FILE *in, const char *format, ...)`
  - read text from stream according to format
- `int fprintf(FILE *out, const char *format, ...)`
  - write the string to output file, according to format
- `int printf(const char *format, ...)`
  - equivalent to `fprintf(stdout, format, ...)`

A final remark on C

- always initialize anything before using it (especially pointers)
- this does not mean that each variable declaration comes with an initialization
- do not use pointers after freeing them
- do not return a function's local variables by reference
- a static array is also a pointer, but do not assign a value to the pointer
- no exceptions – so check for errors everywhere
  - memory allocation
  - system calls
- Murphy's law, C version: anything that cannot fail, will fail

The next lectures

- up until now we have mostly shown toy examples consisting of short code snippets
- the following lectures will present more extended examples with some scientific background, namely
- case studies: solving numerical and combinatorial problems in C
  - binomial coefficients
  - golden section search
  - cartesian products

Solving Numerical and Combinatorial Problems in C: Game Simulation, Monte Carlo estimation, Approximation of functions, Binomial Coefficients, Golden Section Search and Cartesian Products

Stefan Kurtz

Research Group for Genome Informatics
Center for Bioinformatics Hamburg
University of Hamburg

June 4, 2015

Index I

Simulation[1]

A computer simulation is a computer program ...

- that calculates a hard-to-calculate quantity using statistical techniques; OR
- that models the behavior of a system by imitating individual components of the system and their interactions.

Types of simulation

static evaluation of a quantity or state that is difficult to evaluate by other means, e.g. energy density at specific location, 3D-folding of a protein

dynamic evaluation of quantities that arise from the evolution of a system over time

[1] from lecture notes: "Introduction to Simulation", Stanley B. Gershwin, MIT

Simulation

Types of simulation

static monte carlo

dynamic

- discrete: time is discretized, events have finite number of possible outcomes
- continuous $\Rightarrow$ solution of differential equations

Discrete time simulation

- goal: model a random event that occurs with probability $p \in [0, 1]$
- solution:
  - generate a pseudo-random number r which is distributed uniformly between 0 and 1.
  - $r \leq p \iff$ the event has occurred

for convenience, instead of using the interval $[0, 1]$ one scales to larger integer intervals (e.g. when throwing a dice: $[1, 6]$, or generating random sequences $[1, |\Sigma|]$ over alphabet Σ).

Part 1: Simulation of a game[2]

- we consider a simple game of chance, in which a player starts with some initial amount of cash, say x€.
- in each round the player pays 1€ and rolls two dice
  1. if the total of the two dice is 12, then the player wins 6€.
  2. if the total of the two dice is between 8 and 11, then the player wins 2€.
  3. if the total of the two dice is less than 8, then player gets nothing.
- suppose the player plays this game until he/she has no money left, or he/she has a total of y€
- it is not too difficult to analytically determine
  - the probability that the player wins
  - the average number of rounds played before the game is over
- another way to determine these values is to perform a simulation of the game

[2] This section is from *Programming, Problem Solving, and Abstraction with C*, A. Moffat, Pearson Australia, 2013

Simulation of a game

```
typedef struct
{
  unsigned long cash, throws;
} Score; /* current state of game */

static Score throw_dice(bool silent,unsigned long start_cash,
                        unsigned long threshold)
{
  Score score;

  score.cash = start_cash;
  for (score.throws = 1UL;
       score.cash > 0 && score.cash <= threshold;
       score.throws++)
  {
    unsigned long dice1, dice2;

    score.cash--;  /* player pays one unit */
    dice1 = 1UL + drand48() * 6; /* roll first dice */
    dice2 = 1UL + drand48() * 6; /* roll second dice */
```

Simulation of a game

```
    if (dice1 + dice2 == 12)
    {
      score.cash += 6;
    } else
    {
      if (dice1 + dice2 >= 8)
      {
        score.cash += 2;
      }
    }
    if (!silent)
    {
      printf("%3lu␣throw%c␣|%*s*\n",score.throws,
                          score.throws == 1 ? '␣' : 's',
                          (int) score.cash,"");
    }
  }
  return score;
}
```


Simulation of a game

```
static void showresult(const char *key,
                       unsigned long events, /* win/loose */
                       unsigned long numofgames,
                       unsigned long totalthrows)
{
  printf("%5s:␣%lu␣times␣(prob:␣%.2f),␣"
         "average␣number␣of␣throws:␣%.2f\n",
         key,events,(double) events/numofgames,
         (double) totalthrows/events);
}
```

Simulation of a game

```
static void play_games(unsigned long start_cash,
                       unsigned long threshold,
                       unsigned long numofgames)
{
  unsigned long g, loose = 0, loose_totalthrows = 0,
                                  win_totalthrows = 0;

  for (g = 0; g < numofgames; g++)
  {
    Score score = throw_dice(true,start_cash,threshold);
    if (score.cash == 0)
    {
      loose++;
      loose_totalthrows += score.throws;
    } else
      win_totalthrows += score.throws;
  }
  showresult("win",  g - loose,g,win_totalthrows);
  showresult("loose",loose,    g,loose_totalthrows);
}
```


Simulation of a game

```
int main(int argc,char *argv[])
{
  long start_cash, threshold, numofgames;

  if (argc != 4 || sscanf(argv[1],"%ld",&start_cash) != 1 ||
                   start_cash < 1 ||
                   sscanf(argv[2],"%ld",&threshold) != 1 ||
                   threshold < 1 ||
                   threshold < start_cash ||
                   sscanf(argv[3],"%ld",&numofgames) != 1 ||
                   numofgames < 0)
  {
    fprintf(stderr,"Usage:␣%s␣<start_cash>␣<threshold>␣"
                   "<numofgames>\n",argv[0]);
    exit(EXIT_FAILURE);
  }
  srand48(time(NULL)); /* use current time as seed */
```

Simulation of a game

```
  if (numofgames == 0)
  {
    Score score = throw_dice(false,(unsigned long) start_cash,
                             (unsigned long) threshold);
    if (score.cash == 0)
    {
      printf("lost␣all\n");
    } else
    {
      printf("won␣%lu\n",score.cash);
    }
  } else
  {
    play_games((unsigned long) start_cash,
               (unsigned long) threshold,
               (unsigned long) numofgames);
  }
  return EXIT_SUCCESS;
}
```


Simulation of a game

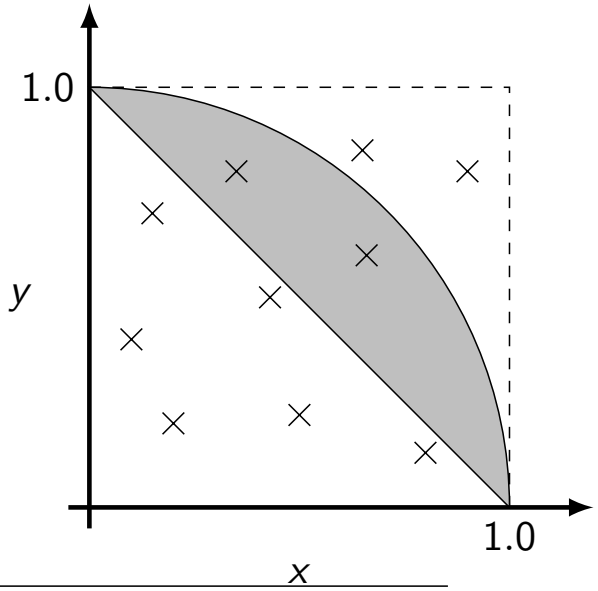
```
$ throw_dice.x 5 20 0
  1 throw  |    *
  2 throws |     *
  3 throws |    *
  4 throws |   *
  5 throws |  *
  6 throws |   *
  7 throws |    *
  8 throws |   *
  9 throws |  *
 10 throws | *
 11 throws |*
lost all
```

```
$ throw_dice.x 5 10 0
  1 throw  |      *
  2 throws |       *
  3 throws |        *
  4 throws |         *
  5 throws |          *
  6 throws |         *
  7 throws |        *
  8 throws |         *
  9 throws |          *
 10 throws |           *
won 11
```

```
$ throw_dice.x 5 20 100000
  win: 12469 times (prob: 0.12), average number of throws: 75.79
loose: 87531 times (prob: 0.88), average number of throws: 37.31

throw_dice.x 5 20 100000
  win: 12450 times (prob: 0.12), average number of throws: 75.36
loose: 87550 times (prob: 0.88), average number of throws: 37.22
```

Part 2: Determining areas by simulation[3]

- we consider a complex geometric shape and the problem to determine the area this shape covers
- consider an arc-shaped region defined by a unit circle:



In this example the area could be calculated as $\frac{\pi}{4} - 0.5 = 0.2854$.

[3]This section is from *Programming, Problem Solving, and Abstraction with C*, A. Moffat, Pearson Australia, 2013

Determining size of areas by simulation

- suppose the shape was more complex:
  - so that no standard geometric formulae is readily available
  - but a criterion to determine if a point lies inside or outside the region of interest
  - in the example above: (a, b) is inside the grey region iff $1 - a \leq b$ and $a^2 + b^2 \leq 1$.
- idea: draw a random sample of all points (a, b), $0 \leq a, b \leq 1.0$ and count how many points are inside the grey area
- this count provides a good estimate of the size of the area
- the method is also called Monte Carlo estimation

Determining size of areas by simulation

```
static bool inside_area(double x,double y)
{
  assert(x <= 1.0 && x >= 0.0 && y <= 1.0 && y >= 0.0);
  return ((1.0 - x) <= y && x * x + y * y <= 1.0)
    ? true : false;
}
```


Determining size of areas by simulation

```
static void estimate_size_of_area(unsigned long maxsteps)
{
  unsigned long steps;

  for (steps = 1; steps <= maxsteps; steps *= 10)
  {
    unsigned long s, inside = 0;

    for (s = 0; s < steps; s++)
    {
      double x = drand48();
      double y = drand48();
      if (inside_area(x,y))
      {
        inside++;
      }
    }
    printf("steps␣=␣%9lu,␣inside␣=␣%9lu,␣ratio␣=␣%.6f\n",
           steps,inside,(double) inside/steps);
  }
}
```

Determining size of areas by simulation

```
int main(int argc, char *argv[])
{
  long maxsteps;

  if (argc != 2 || sscanf(argv[1],"%ld",&maxsteps) != 1
                || maxsteps < 1L)
  {
    fprintf(stderr,"Usage: %s <maxsteps>\n",argv[0]);
    exit(EXIT_FAILURE);
  }
  srand48(time(NULL));
  estimate_size_of_area((unsigned long) maxsteps);
  return EXIT_SUCCESS;
}
```


Determining size of areas by simulation

- A sample of size 10^8 delivers the correct result (in about 3 seconds):

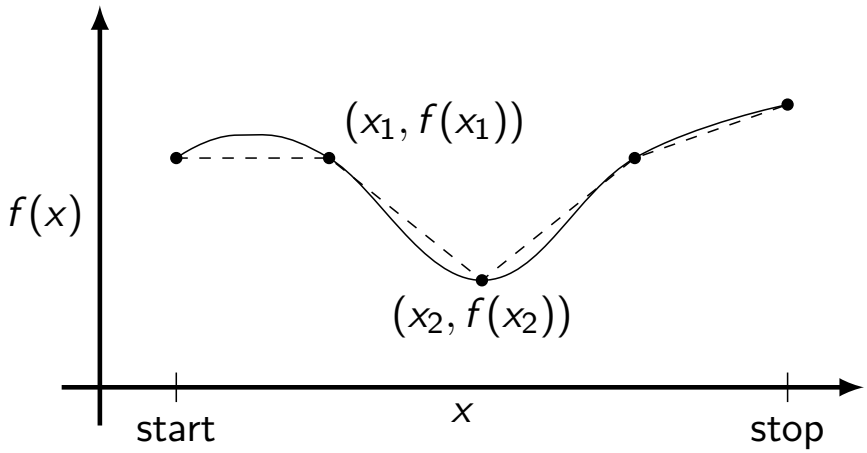
```
$ area_sim.x 100000000
steps =         1, inside =         0, ratio = 0.000000
steps =        10, inside =         1, ratio = 0.100000
steps =       100, inside =        34, ratio = 0.340000
steps =      1000, inside =       299, ratio = 0.299000
steps =     10000, inside =      2810, ratio = 0.281000
steps =    100000, inside =     28373, ratio = 0.283730
steps =   1000000, inside =    286187, ratio = 0.286187
steps =  10000000, inside =   2854821, ratio = 0.285482
steps = 100000000, inside =  28540415, ratio = 0.285404
```

Part 3: Approximation of functions[4]

- consider the function $f(x) = \sin 5x + \cos 10x + \frac{x^2}{10}$
- problem: determine line integral, i.e. the length of the function between some start and stop value.
- application: shaping of metal sheet: determine the total length of metal piece from which a form according to the function can be folded
- for many functions one can apply standard mathematical techniques to solve the problem
- but sometimes it is easier solve the problem by approximation
- idea: approximate the curve by a sequence of straight line segments, see the following illustration.

[4]This section is from *Programming, Problem Solving, and Abstraction with C*, A. Moffat, Pearson Australia, 2013

Approximation of functions



- length of line segments between two points x_1 and x_2 can be computed by evaluating

$$\sqrt{(x_2 - x_1)^2 + (f(x_2) - f(x_1))^2}$$

- $k + 1$ evenly spaced points at constant distance d on x-axes define k line segments
- compute $\sum_{i=1}^{k} \sqrt{d^2 + (f(start + d \cdot (i-1)) - f(start + d \cdot i))^2}$

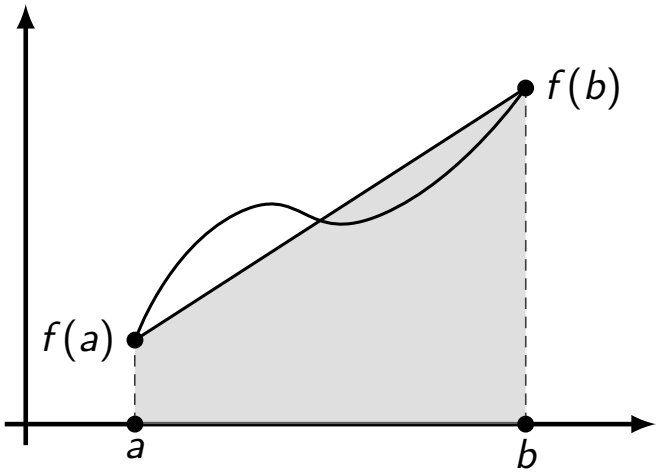
Approximation of functions

$\Rightarrow$ large values of k give good approximation of overall curve length

```
static double approx_line_length(double (*f)(double),
                                 double start,double stop,
                                 unsigned long steps)
{
  double total = 0.0, x1 = start, fx1 = f(start);
  const double dx = (stop - start)/steps, dx_square = dx * dx;
  unsigned long idx;

  for (idx = 0; idx < steps; idx++)
  {
    double fx2 = f(x1 + dx), dy = fx2 - fx1;

    total += sqrt(dx_square + dy * dy);
    fx1 = fx2;
    x1 += dx;
  }
  return total;
}
```


Approximation of functions

- a very similar method can also determine the area under the curve $\Rightarrow$ numerical integration
- approximate the area by a collection of trapezoidal regions
- apply the trapezoidal rule to determine their size (grey shaded area)



$$\begin{aligned}
\text{size of grey area} &= (b-a)\cdot f(a) + \tfrac{1}{2}\cdot(b-a)\cdot(f(b)-f(a)) \\
&= (b-a)\cdot(f(a) + \tfrac{1}{2}\cdot(f(b)-f(a))) \\
&= (b-a)\cdot(f(a) + \tfrac{1}{2}\cdot f(b) - \tfrac{1}{2}\cdot f(a)) \\
&= (b-a)\cdot\tfrac{1}{2}(f(a)+f(b))
\end{aligned}$$

Approximation of functions

```
static double approx_area_under_curve(double (*f)(double),
                                       double start,
                                       double stop,
                                       unsigned long steps)
{
  double total = 0.0, x1 = start, fx1 = f(start);
  const double dx = (stop - start)/steps;
  unsigned long idx;

  for (idx = 0; idx < steps; idx++)
  {
    double fx2 = f(x1 + dx);

    total += dx * (fx1 + fx2)/2.0;
    fx1 = fx2;
    x1 += dx;
  }
  return total;
}
```


Approximation of functions

```
static double wave_fun(double x)
{
  return sin(5.0 * x) + cos(10.0 * x) + (x * x)/10.0;
}

int main(int argc, char *argv[])
{
  long steps, maxsteps;
  if (argc != 2 || sscanf(argv[1],"%ld",&maxsteps) != 1 ||
      maxsteps < 1)
  {
    fprintf(stderr,"Usage: %s <maxsteps>\n",argv[0]);
    exit(EXIT_FAILURE);
  }
  for (steps = 1; steps <= maxsteps; steps *= 10)
  {
    printf("steps = %7ld, line length %.6f, area = %.6f\n",
           steps,
           approx_line_length(wave_fun,0.0,10.0,steps),
           approx_area_under_curve(wave_fun,0.0,10.0,steps));
  }
```

Approximation of functions

```
$ linelen_area.x 1000000
steps =       1, line length 13.862140, area = 57.999720
steps =      10, line length 21.444593, area = 33.551446
steps =     100, line length 67.042603, area = 33.295515
steps =    1000, line length 69.509999, area = 33.289761
steps =   10000, line length 69.534930, area = 33.289704
steps =  100000, line length 69.535179, area = 33.289704
steps = 1000000, line length 69.535182, area = 33.289704
```


Part 4: Binomial coefficients

- for some integer n and some non-negative integer k, $\binom{n}{k}$ is the binomial-coefficient defined by

$$\binom{n}{k} = \prod_{j=1}^{k} \frac{n-(k-j)}{j} \tag{1}$$

- for $n \geq k \geq 0$ it gives the number of k-elements subsets of a set of size n
- example: $\binom{49}{6} = 13983816$ is the number of 6-elements subsets of size 49 (6 from 49)

$\Rightarrow$ $\frac{1}{13983816} = 7.15112e-08$ is the probability for six numbers in the lottery

Properties of Binomial coefficients

Property 1:

For all natural number k, n, $n \geq k \geq 0$, $\binom{n}{k}$ is the coefficient in the binomial formula:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

for any $x, y \in \mathbb{R}$.

Property 2:

For $n \geq k \geq 0$ the following holds:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

The latter property leads to a simple program to compute $\binom{n}{k}$:

Brute force computation of binomial coefficients

```
typedef unsigned long Ulong;

Ulong faculty(Ulong n)
{
  Ulong prod;

  for (prod = 1; n > 0; n--)
    prod *= n;
  return prod;
}

Ulong evalbinom1(Ulong n, Ulong k)
{
  return faculty(n)/(faculty(k) * faculty(n-k));
}
```

- `evalbinom1(49,6)` delivers 1. What happened?
- overflow in integer multiplication

Safe addition (with check for overflow)

- overflow in addition of two variables `unsigned long a, b;` occurs if `a + b > ULONG_MAX` where `ULONG_MAX` is defined in `limits.h`
- `a + b > ULONG_MAX` $\Leftrightarrow$ `a > ULONG_MAX - b`
- the subtraction is safe, as `b <= ULONG_MAX` is required
- applying this technique gives the following function for safe addition:

```
Ulong safeadd(Ulong a,Ulong b)
{
  if (a == 0) return b;
  if (b == 0) return a;
  assert(b <= ULONG_MAX);
  if (a > ULONG_MAX - b)
  {
    fprintf(stderr,"%s: overflow (%lu + %lu)\n",__func__,a,b);
    exit(EXIT_FAILURE);
  }
  return a + b;
}
```


Safe multiplication (with check for overflow)

- Apply the same technique for safe multiplication
- use division instead of subtraction

```
Ulong safemult(Ulong a, Ulong b)
{
  if (a == 0 || b == 0)
    return 0;
  if (a > ULONG_MAX/b)
  {
    fprintf(stderr,"%s: overflow (%lu * %lu)\n",
            __func__,a,b);
    exit(EXIT_FAILURE);
  }
  return a * b;
}
```

- `__func__` is preprocessor constant: name of function statement or expression appears in

Safe evaluation of binomial coefficient

```
Ulong safefaculty(Ulong n)
{
  Ulong prod;
  for (prod = 1; n > 0; n--)
  {
    prod = safemult(prod,n);
  }
  return prod;
}

Ulong safeevalbinom1(Ulong n, Ulong k)
{
  return safefaculty(n)/safemult(safefaculty(k),
                                 safefaculty(n-k));
}
```

`evalbinom1(49,6)` leads to program exit with error message

`safemult:  overflow for 11630125423268352000 * 38`

A solution with interweaved multiplication and division

- compute $\binom{n}{k}$ directly according to the definition: $\prod_{j=1}^{k} \frac{n-(k-j)}{j}$

```
Ulong evalbinom2(Ulong n, Ulong k)
{
  Ulong j, result;

  if (k == 0)
    return 1UL;
  if (n == 0)
    return 0;
  for (result = 1UL, j=1; j<=k; j++)
    result = safemult(result,n - k + j)/j;
  return result;
}
```

`evalbinom2(49,6)` delivers correct result but

`evalbinom2(63,34)` leads to overflow

A solution without division and multiplication

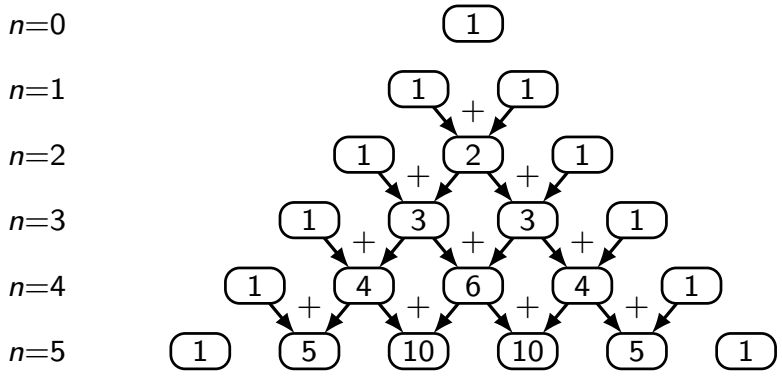
- the following recurrences hold:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ for all } n,k > 0$$

$$\binom{n}{0} = 1 \text{ for all } n \geq 0$$

$$\binom{0}{k} = 0 \text{ for all } k > 0$$

- this recurrence evaluates Pascal's Triangle:



A solution without division and multiplication

```
Ulong evalbinom3(Ulong n, Ulong k)
{
  if (k == 0)
    return 1UL;
  if (n == 0)
    return 0;
  return safeadd(evalbinom3(n-1,k-1),evalbinom3(n-1,k));
}
```

- solution wastes time as many results are computed again and again
- e.g. evaluating `evalbinom3(49,6)` leads to 32244451 function calls

A solution using dynamic programming

- use dynamic programming to store results in table when they are computed for the first time
- as the values in the table are initially not defined, we introduce a structure to keep track of whether they are defined or not:

```
typedef struct
{
  bool defined;
  unsigned long value;
} Definedvalue;
```


A solution using dynamic programming

```
Ulong evalbinom4memo(Definedvalue **tab, Ulong n, Ulong k)
{
  if (k == 0) return 1UL;
  if (n == 0) return 0;
  if (!tab[n-1][k-1].defined)
  {
    tab[n-1][k-1].value = evalbinom4memo(tab,n-1,k-1);
    tab[n-1][k-1].defined = true;
  }
  if (!tab[n-1][k].defined)
  {
    tab[n-1][k].value = evalbinom4memo(tab,n-1,k);
    tab[n-1][k].defined = true;
  }
  return safeadd(tab[n-1][k-1].value,tab[n-1][k].value);
}
```

A solution without division and multiplication

```
Ulong evalbinom4(Ulong n, Ulong k)
{
  Ulong ret, i, j;
  Definedvalue **tab;

  array2dim_malloc(tab, n+1,k+1);
  for (i=0; i<=n; i++)
    for (j=0; j<=k; j++)
      tab[i][j].defined = false;
  ret = evalbinom4memo(tab,n,k);
  array2dim_delete(tab);
  return ret;
}
```

- macros `array2dim_malloc` and `array2dim_delete` have been introduced before (see C-programming course)
- note that table entries are first set to be undefined
- whenever a value is to be computed, it is first checked if it is already stored in table
- if not stored $\Rightarrow$ compute it; otherwise $\Rightarrow$ take it

A solution using logarithms

- for all positive integers p the function Γ is defined by $\Gamma(p) = (p-1)!$
- Γ is spelled as *Gamma*
- the following holds:

$$
\begin{aligned}
\binom{n}{k} &= \frac{n!}{k!(n-k)!} \\
&= \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)!} \\
&= \exp\left(\ln \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)}\right) \\
&= \exp\left(\ln\Gamma(n+1) - (\ln(\Gamma(k+1)\Gamma(n-k+1)))\right) \\
&= \exp\left(\ln\Gamma(n+1) - (\ln\Gamma(k+1) + \ln\Gamma(n-k+1))\right) \\
&= \exp\left(\ln\Gamma(n+1) - \ln\Gamma(k+1) - \ln\Gamma(n-k+1)\right)
\end{aligned}
$$

A solution using logarithms

- the last right-hand side can be computed using the *C*-function `lgamma` from the math library: `double lgamma(double x);`
- `lgamma(p)` computes $\ln \Gamma(p)$ for any positive integer p

```
Ulong evalbinom5(Ulong n, Ulong k)
{
  assert (k<=n);
  return (Ulong) exp(lgamma(n+1) - lgamma(k+1)
                                 - lgamma(n-k+1));
}
```

- use of logarithms and exponential functions leads to small rounding errors and thus incorrect results:

$$\begin{array}{rcll} \binom{49}{6} & = & 13983816 \text{ but} & \\ \texttt{evalbinom5(49,6)} & = & 13983815 & \text{last digit wrong} \\ \binom{63}{20} & = & 13488561475572645 \text{ but} & \\ \texttt{evalbinom5(63,20)} & = & 13488561475572828 & \text{last three digits wrong} \end{array}$$

Part 5: Minimizing unimodal functions

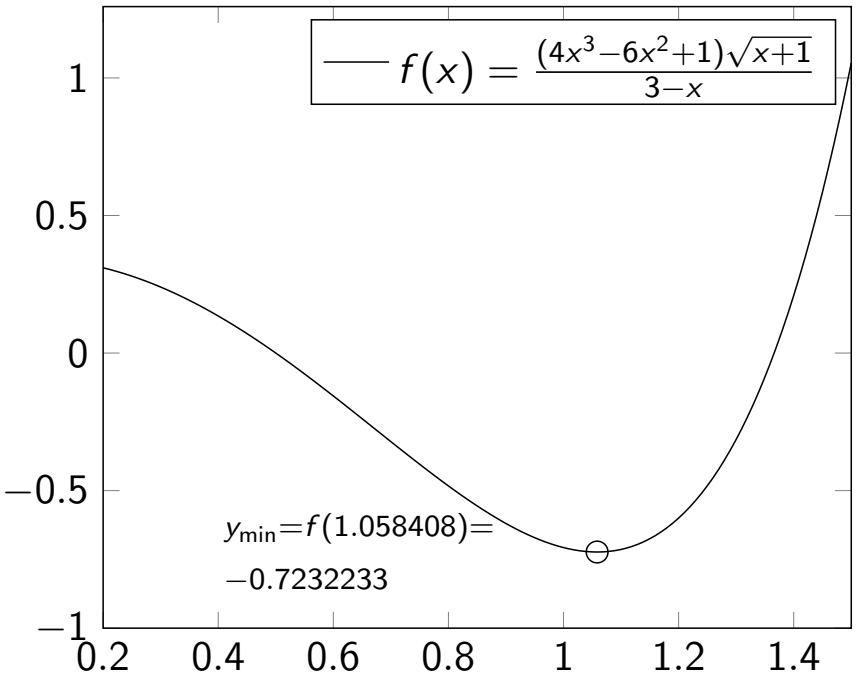
- consider a function $f : \mathbb{R} \to \mathbb{R}$ and an interval $[\ell, r]$, $\ell, r \in \mathbb{R}$, $\ell \leq r$
- a function $f : \mathbb{R} \to \mathbb{R}$ is unimodal with minimum $x_{\min}$ in $[\ell, r]$, if $\ell \leq x_{\min} \leq r$ and the following holds:
  - for all x_1, x_2, $\ell \leq x_1 < x_2 < x_{\min}$ we have $f(x_1) > f(x_2) > f(x_{\min})$
  - for all x_3, x_4, $x_{\min} < x_3 < x_4 \leq r$ we have $f(x_{\min}) < f(x_3) < f(x_4)$
- f is unimodal in $[\ell, r]$ if there is some $x_{\min}$ such that f is unimodal with minimum $x_{\min}$ in $[\ell, r]$
- f is monotone decreasing for values $< x_{\min}$ and
- f is monotone increasing for values $> x_{\min}$

$\Rightarrow$ f has only one minimum in the given interval

- problem: given unimodal function f, find $x_{\min}$

[4] Material for this part taken from: W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in C: The Art of Scientific Computing, (second edition), 1992, Cambridge Univ Press (Golden Section search is explained in Section 10.1). Figure on page 43 from S.S. Rao, S.S. Rao, "Engineering optimization: theory and practice", page 253, John Wiley and sons, 2004

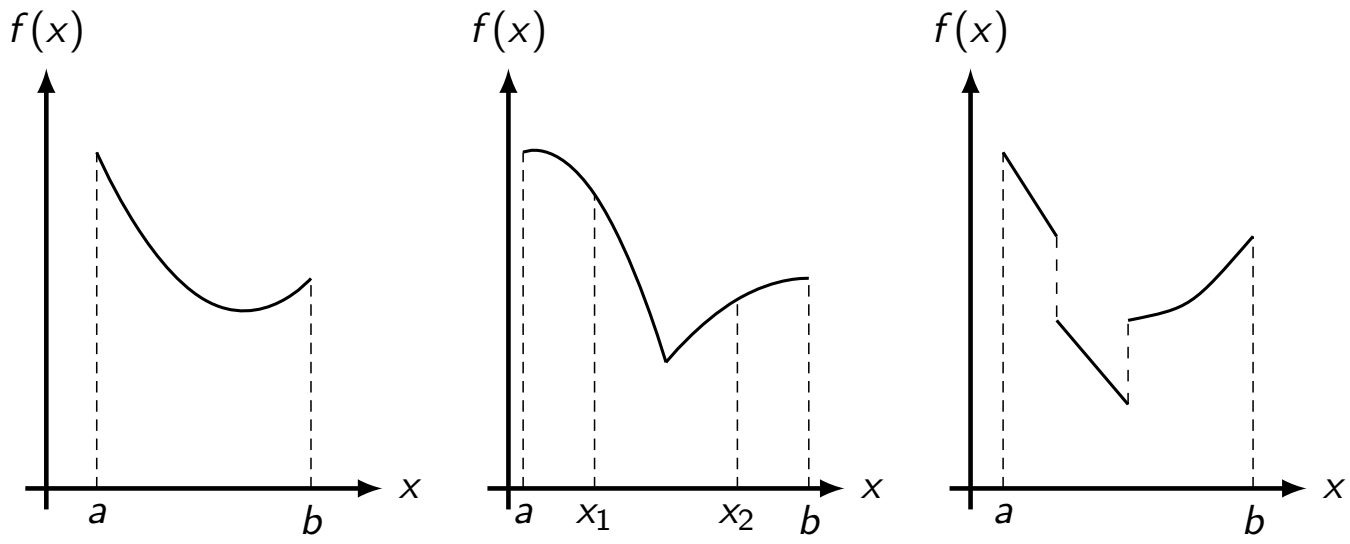
Example of unimodal function



f is unimodal in [0.2..1.5] and $x_{\min}$ =1.058408

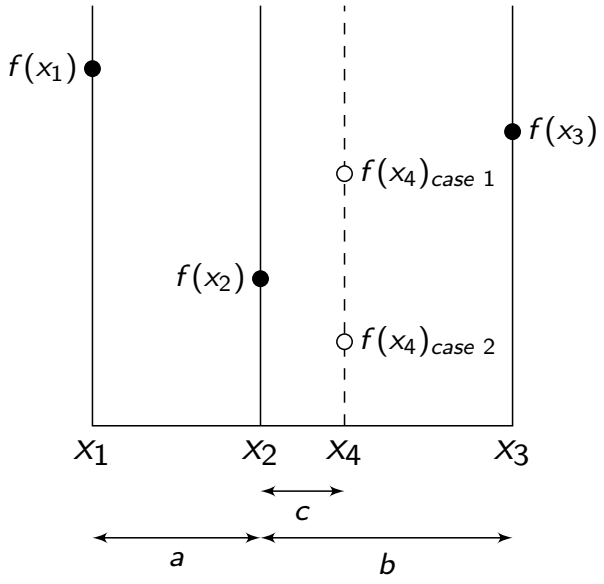
A method for minimizing unimodal functions

- use *golden section search* which iteratively narrows an interval containing $x_{\min}$
- method does not require function to be continuous (stetig)
- works (among others) for functions with the following shape:



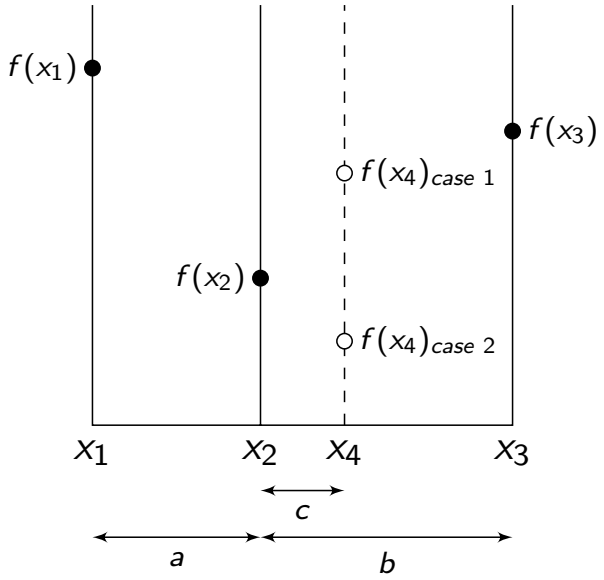
A method for minimizing unimodal functions

- let f be a function f which is unimodal in $[x_1, x_3]$
- let $x_1 \leq x_2 \leq x_3$ and $|x_2 - x_1| < |x_3 - x_2|$ and suppose $f(x_1) > f(x_2) < f(x_3)$, as shown here:



- by unimodality and choice of triple $(x_1, x_2, x_3) \Rightarrow$ minimum between x_1 and x_3
- choose some x_4 either between x_1 and x_2 or between x_2 and x_3
- figure shows latter choice

A method for minimizing unimodal functions



- $f(x_4)$, case 1: $f(x_2) < f(x_4) \Rightarrow$ minimum is between x_1 and x_4

$\Rightarrow$ continue with triple (x_1, x_2, x_4)

- $f(x_4)$, case 2: $f(x_2) > f(x_4) \Rightarrow$ minimum is between x_2 and x_3

$\Rightarrow$ continue with triple (x_2, x_4, x_3)

- case that left part from x_1 to x_2 is larger then right part from x_2 to x_3 is handled by exchanging the first and third value of the triple.

$\Rightarrow$ in case 1 continue with (x_4, x_2, x_1).

A method for minimizing unimodal functions: sketch of algorithm

- general rule: middle point x_2 of each triple (x_1, x_2, x_3) has the smalltest function value seen so far in the iteration
- either $(x_1 \leq x_2 \leq x_3$ and $x_2 - x_1 < x_3 - x_2)$ or
  $(x_3 \leq x_2 \leq x_1$ and $x_1 - x_2 < x_2 - x_3)$
- process is called *bracketing*
- each bracketing step delivers triple representing a narrower search interval that is guaranteed to contain the function's minimum in the original interval
- continue bracketing until distance between the two outer values of the triple is sufficiently small

Function `gsearch` on next page implements this idea


```
static double gsearch(double (*f)(double),
                      double x1,double x2,double x3)
{
  unsigned long iter;
  for (iter = 0; !sufficiently_small(x1,x3); iter++)
  {
    double x4;

    /* x1<=x2<=x3 && x2-x1<x3-x2 || x3<=x2<=x1 && x1-x2<x2-x3 */
    x4 = x2 + det_offset(x3 - x2); /* use offset c */
    if (f(x2) <= f(x4))
    { /* f(x_4), case 1 */
      x3 = x1; x1 = x4;
      /* now x3 <= x2 <= x1 and x1 - x2 < x2 - x3 */
    } else
    { /* f(x_4), case 2 */
      x1 = x2; x2 = x4;
      /* now x1 <= x2 <= x3 and x2 - x1 < x3 - x2 */
    }
  }
  printf("iterations=%lu\n",iter);
  return (x1 + x3)/2.0; /* xmin is halfway between x1 and x3*/
}
```

- implement the function $f(x) = \frac{(4x^3-6x^2+1)\sqrt{x+1}}{3-x}$:

```
static double samplefun(double x)
{
  return ((4.0 * pow(x,3.0) - 6.0 * pow(x,2.0) + 1) *
          sqrt(x+1.0))/(3.0 - x);
}

int main(int argc,char *argv[])
{
  double xmin = gsearch(samplefun,0.0,0.2,1.5);
  printf("f(xmin=%.20f)=%.20f\n",xmin,samplefun(xmin));

  return EXIT_SUCCESS;
}
```

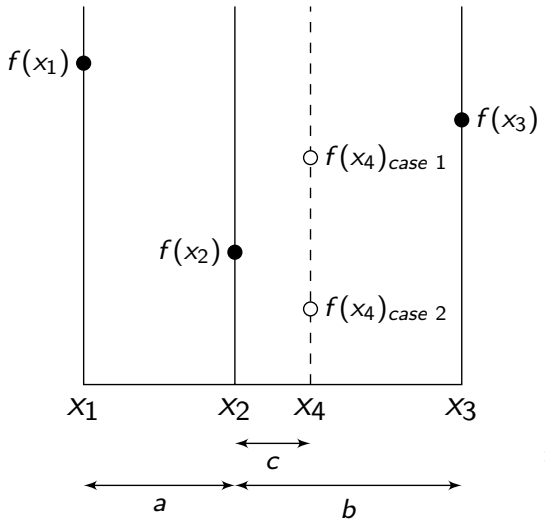
program output

```
iterations=40 f(xmin=1.0584076389981169 6191)=-0.72322333714960007622
```

Remaining task

define functions `det_offset` and `sufficiently_small`

Determining offset for x_4 via `det_offset`



- suppose x_4 is determined
- new search interval either
  - between x_1 and x_4 ($f(x_4)_{\text{case 1}}$) of length $a + c$ or
  - between x_2 and x_3 ($f(x_4)_{\text{case 2}}$) of length b
- to optimize rate of convergence, in both cases the interval size should be equal

$\Rightarrow$ $b = a + c$

- choose a constant proportion $\frac{a}{b}$ of spacing between the triple x_1, x_2 and x_3 during all iterations
- if (x_1, x_2, x_4) is the new triple, we want $\frac{c}{a} = \frac{a}{b}$
- if (x_2, x_4, x_3) is the new triple, we want $\frac{c}{b-c} = \frac{a}{b}$

Determining offset for x_4 via `det_offset`

To combine these equations we solve for c:

$$\frac{c}{a} = \frac{a}{b} \quad \Leftrightarrow \quad c = \frac{a^2}{b}$$

$$\begin{aligned}
\frac{c}{b-c} = \frac{a}{b} \quad &\Leftrightarrow \quad c = \frac{a}{b}(b-c) \\
&\Leftrightarrow \quad c = \frac{a}{b}b - \frac{a}{b}c \\
&\Leftrightarrow \quad \frac{a}{b}c + c = a \\
&\Leftrightarrow \quad \frac{ca}{b} + \frac{cb}{b} = a \\
&\Leftrightarrow \quad \frac{ca+cb}{b} = a \\
&\Leftrightarrow \quad c(a+b) = ab \\
&\Leftrightarrow \quad c = \frac{ab}{a+b}
\end{aligned}$$

Determining offset for x_4 via `det_offset`

As c appears on left side of both equations, right hand sides must be equal:

$$\begin{aligned}
\frac{a^2}{b} = \frac{ab}{a+b} \quad &\Leftrightarrow \quad a = \frac{b^2}{b+a} \\
&\Leftrightarrow \quad a(b+a) = b^2 \\
&\Leftrightarrow \quad ab + a^2 = b^2 \\
&\Leftrightarrow \quad \frac{ab}{b^2} + \frac{a^2}{b^2} = \frac{b^2}{b^2} \\
&\Leftrightarrow \quad \frac{a}{b} + \left(\frac{a}{b}\right)^2 = 1 \\
&\Leftrightarrow \quad \left(\frac{a}{b}\right)^2 + \frac{a}{b} - 1 = 0
\end{aligned}$$

Determining offset for x_4 via `det_offset`

Solving the previous equation:

- the equation

$$\left(\frac{a}{b}\right)^2 + \frac{a}{b} - 1 = 0$$

  can be written as a quadratic equation of the form

$$nx^2 + mx + \ell = 0$$

  with $x = \frac{a}{b}$, $n = m = 1$ and $\ell = -1$
- the quadratic equation has the solution

$$x = \frac{-m + \sqrt{m^2 - 4n\ell}}{2n} = \frac{-1 + \sqrt{1 - 4 \cdot 1 \cdot (-1)}}{2} = \frac{\sqrt{5} - 1}{2}$$

$\Rightarrow$ $\varphi = \frac{\sqrt{5}-1}{2}$ (the golden ratio) is the choice for $\frac{a}{b} = \frac{c}{a} = \frac{c}{b-c}$

Determining offset for x_4 via `det_offset`

- from the previous we conclude

$$c = b - a = b - \frac{a}{b}b = b - \varphi b = (1 - \varphi)b$$

- which implies

$$x_4 = x_2 + c = x_2 + (1 - \varphi)b = x_2 + (1 - \varphi)(x_3 - x_2)$$

- In the program we therefore write

```
x4 = x2 + det_offset(x3 - x2);
```

- and implement `det_offset` as follows:

```
#define GOLD  ((sqrt(5.0) - 1.0)/2.0) /* 0.6180339887 */
static double det_offset(double dist)  /* x3 - x2 */
{
  return (1.0 - GOLD) * dist;
}
```

- Note that `dist` may become < 0 after $f(x_4)$ case 1, because $x_3 \leq x_2$.

Termination via `sufficiently_small`

- recall: bracketing is continued until distance between the two outer points of the triple is "sufficiently small"
- we use the criterion

$$|x_3 - x_1| < \tau$$

  where τ is a tolerance parameter of the algorithm
- choose $\tau = \sqrt{\epsilon}$ where ϵ is the machine accuracy
- implement `sufficiently_small` as follows:

```
#define TOL sqrt(1.110223e-16)   /* argument is accuracy */
bool sufficiently_small(double x1,double x3)
{
  /* fabs(x3 - x1) is the distance between x3 and x1 */
  return fabs(x3 - x1) < TOL ? true : false;
}
```


Machine Accuracy ϵ

- ϵ is the largest floating point number, which when added to the floating point number `1.0` gives the result `1.0`
- for all larger values $z > \epsilon$ we get $1.0 + z > 1.0$.
- for some floating point type `T`, ϵ is computed by the following code-fragment:

```
T eps;
for (eps = (T) 1.0; (T) 1.0 + eps > (T) 1.0; eps /= (T) 2.0)
   /* Nothing */;
```

T	ϵ	iterations	const ϵ' from `float.h`
`float`	$5.960464e-08$	24	`FLT_EPSILON`
`double`	$1.110223e-16$	53	`DBL_EPSILON`
`long double`	$5.421011e-20$	64	`LDBL_EPSILON`

- $\epsilon' = \min\{f \mid 1.0 + f > 1.0\} \Rightarrow$ one more iteration $\Rightarrow \epsilon = \frac{\epsilon'}{2}$

Part 6: Cartesian Products

- suppose we have k sets $S_0, S_1, \ldots, S_{k-1}$
- the cartesian product $S_0 \times S_1 \times \cdots \times S_{k-1}$ is the set of k-tuples which can be build from the elements of the given sets
- formally $S_0 \times S_1 \times \cdots \times S_{k-1}$ is the set

$$\{(a_0, a_1, \ldots, a_{k-1}) \mid a_0 \in S_0, a_1 \in S_1, \ldots, a_{k-1} \in S_{k-1}\}$$

- $|S_0 \times S_1 \times \cdots \times S_{k-1}| = \prod_{i=0}^{k-1} |S_i|$ is the size of the cartesian product
- cartesian products have many applications for k-dimensional problems, e.g.
  - enumerating all sequences of length k over a given alphabet
  - combining matches from k genomes
  - enumerating all neighbors in a k-dimensional vector space

Enumerating cartesian products

- for a small fixed value of k one can easily write cascades of `for`-loops to enumerate the cartesian product
- we first need a datatype for sets (of `unsigned long`-Elements in the example)

```
typedef unsigned long Ulong;
typedef struct Set Set; /* only know the name of the type */
Ulong set_size(const Set *); /* deliver size of set */
Ulong set_get(const Set *set,Ulong i); /* ith element */

void cartproduct2(const Set *s0,const Set *s1)
{
  Ulong i0, i1;

  for (i0=0; i0<set_size(s0); i0++)
    for (i1=0; i1<set_size(s1); i1++)
      printf("%lu␣%lu\n",set_get(s0,i0),set_get(s1,i1));
}
```

Enumerating cartesian products

for $k = 3$ we need the following loops:

```
void cartproduct3(const Set *s0,const Set *s1,const Set *s2)
{
  Ulong i0, i1, i2;

  for (i0=0; i0<set_size(s0); i0++)
    for (i1=0; i1<set_size(s1); i1++)
      for (i2=0; i2<set_size(s2); i2++)
        printf("%lu␣%lu␣%lu\n",set_get(s0,i0),set_get(s1,i1),
                               set_get(s2,i2));
}
```


Enumerating cartesian products

- in many applications k is not fixed and not known at compile time
- need a more flexible way to enumerate cartesian products
- we provide an iterative solution based on the concept of k turning wheels
- for each $i, 0 \leq i \leq k-1$, the ith wheel can take a value in the range from 0 to $|S_i| - 1$
- the state of the wheels describes the k-tuple

$$(S_0[w_0], S_1[w_1], \ldots, S_{k-1}[w_{k-1}])$$

where

- w_i is the value taken by the ith wheel
- $S_i[w_i]$ denotes the element with index w_i in S_i

Example of turning wheels

- Let $k = 2$ and $S_0 = S_1 = \{a, b\}$
- each of the two wheels can take two values 0 and 1 and a state of the wheels describes a pair which here is interpreted as a string of length 2 over the alphabet $\{a, b\}$

w_0	w_1	$(S_0[w_0], S_1[w_1])$	string
0	0	(a, a)	aa
0	1	(a, b)	ab
1	0	(b, a)	ba
1	1	(b, b)	bb

Turning 2 wheels

- to turn the wheels, one transforms the cascade of loops into a single loop which, in each iteration, computes the next state of the wheels

```
void cartproduct2singleloop(const Set **sets) /* 2 sets */
{
  Ulong w[2] = {0}, loopid = 1UL; /* 0=outer, 1=inner */

  printf("%lu␣%lu\n",set_get(sets[0],0),set_get(sets[1],0));
  for(;;)
  {
    w[loopid]++; /* turn wheel loopid */
    if (w[loopid] == set_size(sets[loopid])) /* end of loop */
    {
      w[loopid] = 0; /* init for next iteration of this loop */
      if (loopid == 0) /* at end of outer loop: done */
        break;
      loopid--; /* now turn to outer loop */
    } else
    {
      printf("%lu␣%lu\n",set_get(sets[0],w[0]),
                         set_get(sets[0],w[1]));
      if (loopid < 1UL) loopid = 1UL; /* return to inner loop */
    }
  }
}
```

Turning 3 wheels

```
void cartproduct3singleloop(const Set **sets)
{
  Ulong w[3] = {0},
        loopid = 2UL; /* 0=outer, 1=middle, 2=inner */

  /* output */
  for(;;)
  {
    w[loopid]++; /* turn wheel loopid */
    if (w[loopid] == set_size(sets[loopid])) /* end of loop */
    {
      w[loopid] = 0; /* init for next iteration of this loop */
      if (loopid == 0) /* already at end of outermost loop: done */
        break;
      loopid--; /* now turn to surrounding loop */
    } else
    {
      /* output */
      if (loopid < 2UL) loopid = 2UL; /* return to inner loop */
    }
  }
}
```


Turning k wheels

- `cartproduct2singleloop` and `cartproduct3singleloop` only differ in the size of the array `w` and the setting of the variable `loopid` for the inner loop

$k = 2$	$k = 3$
`w[2] = 0`	`w[3] = 0`
`loopid = 1UL`	`loopid = 2UL`

- for an arbitrary k use
  - an array `w` of k elements initialized to 0
    `Ulong *w = calloc(k,sizeof (*w))`
  - and set `loopid` to $k - 1$ if necessary
    `if (loopid < k-1) loopid = k-1`
- for output of a set of sets for a given wheel state we use the function

```
void cardproduct_show(const Set **sets,Ulong k,const Ulong *w);
```

Turning k wheels

```
void cartproductsingleloop(const Set **sets,Ulong k)
{
  Ulong *w = calloc(k,sizeof (*w)),
        loopid = k-1; /* 0=outer, ..., k-1 = inner*/

  cardproduct_show(sets,k,w);
  for(;;)
  {
    w[loopid]++; /* turn wheel loopid */
    if (w[loopid] == set_size(sets[loopid])) /* end of loop */
    {
      w[loopid] = 0; /* init for next iteration of this loop */
      if (loopid == 0) /* already at end of outermost loop: done */
        break;
      loopid--; /* now turn to surrounding loop */
    } else
    {
      cardproduct_show(sets,k,w);
      if (loopid < k-1) loopid = k-1; /* return to inner loop */
    }
  }
  free(w);
}
```


Turning k wheels: implementation of Set-type

```
struct Set {
  Ulong *elements, /* store the elements here */
        size, /* number of elements store */
        allocated; /* number of entries allocated */
};

Set *set_new(void) /* constructor for new empty set */
{
  Set *s = malloc(sizeof (*s));
  assert(s != NULL);
  s->elements = NULL;
  s->size = 0;
  s->allocated = 0;
  return s;
}

void set_delete(Set *set) /* destructor */
{
  if (set != NULL)
  {
    free(set->elements);
    free(set);
  }
}
```

Turning k wheels: implementation of Set-type

```
bool set_member(const Set *set,Ulong elem) /* elem \in set? */
{
  Ulong idx;

  for (idx = 0; idx < set->size; idx++)
    if (set->elements[idx] == elem)
      return true;
  return false;
}

void set_add(Set *set,Ulong elem) /* add elem to set */
{
  if (!set_member(set,elem))
  {
    if (set->size == set->allocated)
    {
      const Ulong add = 32UL;
      set->elements = realloc(set->elements,set->allocated + add);
      set->allocated += add;
    }
    set->elements[set->size++] = elem;
  }
}
```


Turning k wheels: implementation of Set-type

```
Ulong set_size(const Set *set) /* size of set */
{
  return set->size;
}

Ulong set_get(const Set *set,Ulong i) /* get ith element from set */
{
  assert(i < set->size);
  return set->elements[i];
}

void cardproduct_show(const Set **sets,Ulong k,const Ulong *w)
{ /* show cartesian product for given wheel state w */
  Ulong idx;

  for (idx=0; idx<k; idx++)
  {
    printf("%lu%s",set_get(sets[idx],w[idx]),
                   idx < k-1 ? "␣" : "\n");
  }
}
```

Turning k wheels: reading Sets from argv

```
Set **sets_read(const char **argv,Ulong k)
{ /* parse sets of strings from argv: "1 2" "5 7" are valid args */
  unsigned long idx; long readlong;
  const char *ptr;
  Set **sets;

  sets = malloc(sizeof (*sets) * k);
  assert(sets != NULL);
  for (idx=0; idx<k; idx++)
  {
    sets[idx] = set_new();
    for (ptr = argv[idx+1];
         sscanf(ptr,"%ld",&readlong) == 1;
         ptr++)
    {
      set_add(sets[idx],(Ulong) readlong);
      ptr = strchr(ptr,(int) ' ');
      if (ptr == NULL)
        break;
    }
  }
  return sets;
}
```


Turning k wheels: reading Sets from argv

```
void sets_delete(Set **sets,Ulong k) /* delete set of sets */
{
  unsigned long idx;

  for (idx=0; idx<k; idx++)
    set_delete(sets[idx]);
  free(sets);
}

int main(int argc,const char **argv)
{
  if (argc >= 2)
  {
    Ulong k = (Ulong) argc-1;
    Set **sets = sets_read(argv,k);

    cartproductsingleloop((const Set **) sets,k);
    sets_delete(sets,k);
  }
  return EXIT_SUCCESS;
}
```

Turning k wheels: example output

```
cart2-3.x "1 2 3" "4 5" "6 7"
1 4 6
1 4 7
1 5 6
1 5 7
2 4 6
2 4 7
2 5 6
2 5 7
3 4 6
3 4 7
3 5 6
3 5 7
```

Shared Memory Multithreading with Phreads

Stefan Kurtz

Research Group for Genome Informatics
Center for Bioinformatics Hamburg
University of Hamburg

June 4, 2015

Index I

Index II

References

- POSIX Threads Programming, Tutorial, Lawrence Livermore National Laboratory, `https://computing.llnl.gov/tutorials/pthreads/`
- ECE 1747 Parallel Programming, slides from `http://www.eecg.toronto.edu/`
- James Demmel and Kathy Yelick `http://www.cs.berkeley.edu/~demmel/cs267_Spr11/`

Threads: the basics

- thread is independent stream of instructions that can be scheduled to run as such by the operating system
- viewpoint of programmer: thread can be seen as a function that runs independently from its main program
- consider a main program that contains a number of functions.
- if all of these function are able to be scheduled to run simultaneously and/or independently by the operating system, we have a "multi-threaded" program

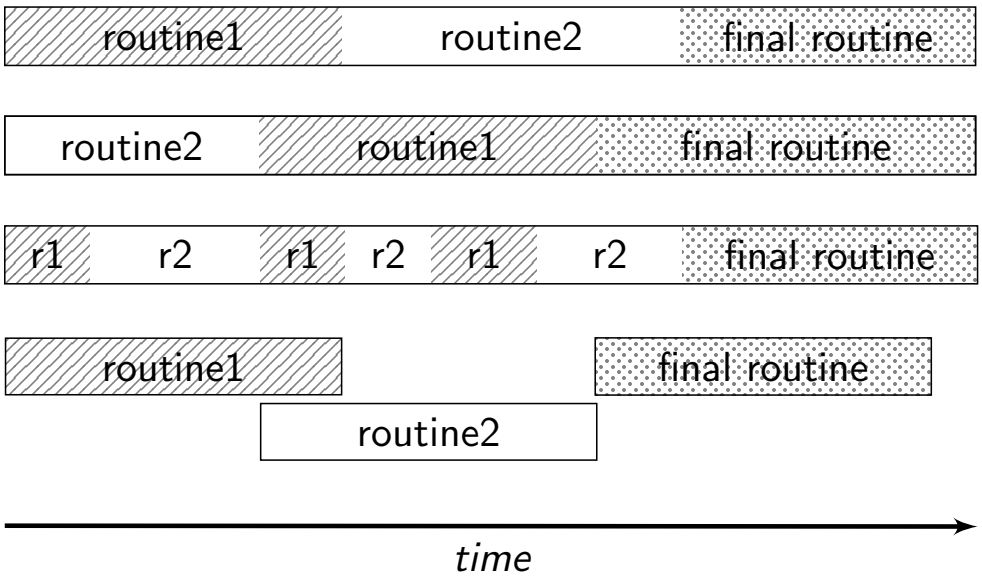
$\Rightarrow$ restructure programs such that parallel parts form separate functions

- task of software developer:
  - decide how to decompose the computation into parallel parts
  - create and destroy threads to support that decomposition
  - add synchronization to make sure that dependences are covered

Applicability of Pthreads

Programs with one of the following characteristics may be suited for pthreads:

1. Data that can be operated on by multiple tasks simultaneously
2. Use many CPU cycles in some places but not others
3. Some work is more important than other work



Models for threaded programs

Manager/worker

- single thread (manager) assigns work to other threads (workers)
- typically, the manager handles all input and schedules work for worker threads
- static worker pool: #worker threads is determined initially
- dynamic worker pool: #worker threads adapts to the task

Pipeline:

a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread.

Peer:

similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

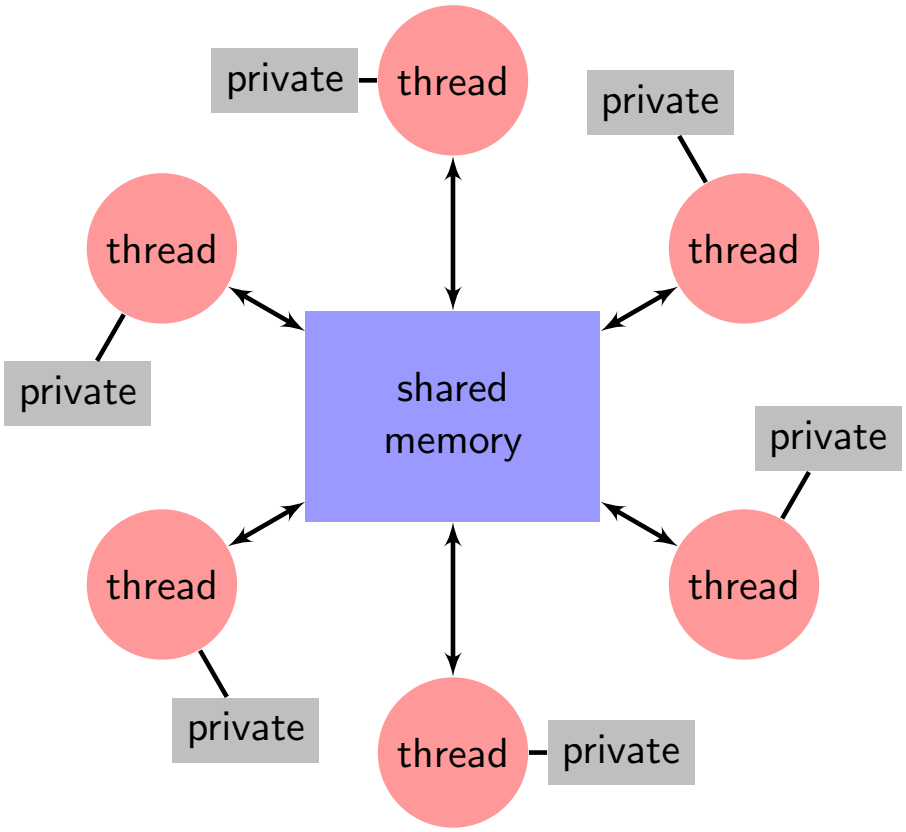
Threads and the memory

shared memory

- all threads have access to the same shared memory space (mostly for reading)
- additionally own private data

Software developer is

responsible for synchronizing access to shared data



Overview of POSIX threads

- POSIX: Portable Operating System Interface for UNIX
- interface to operating system utilities
- Pthreads: POSIX standard shared-memory multithreading interface
- system calls for thread management and synchronization
- should be relatively uniform across UNIX-like OS platforms
- pthreads contain support for
  - creating parallelism
  - synchronizing
- no explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread
- overhead of creating threads is cheap compared to creating processes
- motivation for using Pthreads: realize potential program performance gains on multi-core machines

Forking Posix Threads

Signature:

```
int pthread_create(pthread_t *,const pthread_attr_t *,
                   void * (*)(void *), void *);
```

- `thread_id` is the thread id or handle (used to halt, etc.)
- `thread_attribute` various attributes
- Standard default values obtained by passing a NULL pointer
- Sample attribute: minimum stack size
- `thread_fun` the function to be run (takes and returns `void*`)
- `fun_arg` an argument can be passed to `thread_fun` when it starts
- `errorcode` will be set nonzero if the create operation fails

Example call:

```
errcode = pthread_create(&thread_id, &thread_attribute,
                         thread_fun, &fun_arg);
```

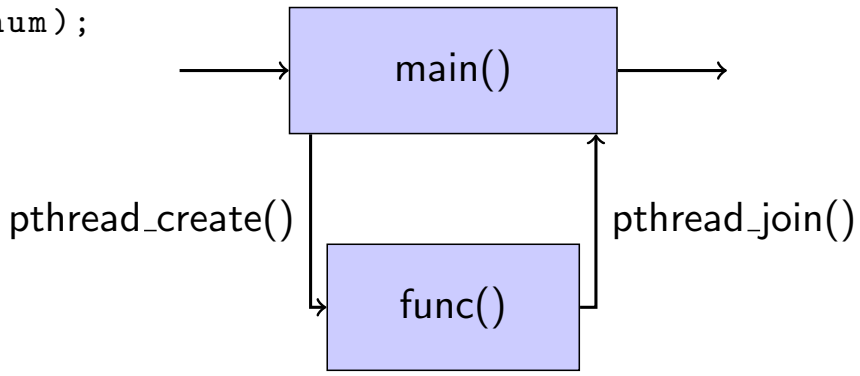

General Structure of Threaded Programs

```
int pthread_join(pthread_t thread, void **result);
```

wait for the thread to exit and, if `result` is not `NULL`, place the exit status of the target thread into the memory location referred to by `result`.

```
#include <pthread.h>
void *func(void *arg) { /* execute in parallel */ }
pthread_t id; int threadnum = 0;
pthread_create(&id,NULL,
               func,
               &threadnum);
...
pthread_join(id,NULL);
```

- to compile add option `-lpthread` to loader: `LDFLAGS+=-lpthread`



A minimal working programming with threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static void *print_msg(void *ptr)
{
  printf("%s\n", (char *) ptr);
  return NULL;
}

int main(void)
{
  const char *msg1 = "Thread␣1";
  const char *msg2 = "Thread␣2";
  pthread_t tid1, tid2;

  /* Create independent threads each of which will
     execute function */
  pthread_create(&tid1, NULL, print_msg, (void *) msg1);
  pthread_create(&tid2, NULL, print_msg, (void *) msg2);
```


A minimal working programming with threads

```
  /* Wait until threads are complete before main continues.
     Unless we wait we have the risk of executing an exit
     which will terminate the process and all threads
     before the threads have completed. */

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("Thread␣1␣finished\n");
  printf("Thread␣2␣finished\n");
  exit(EXIT_SUCCESS);
}
```

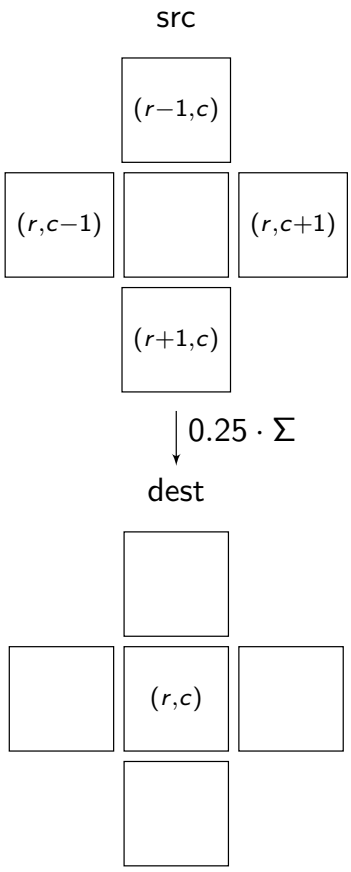
```
Thread 1
Thread 2
Thread 1 finished
Thread 2 finished
```

A useful example: compute ranges of rows of matrix

- goal: for a given $(m+1) \times (n+1)$-matrix *src* compute new matrix

$$dest_{r,c} = 0.25 \cdot (src_{r,c-1} + src_{r,c+1} + src_{r-1,c} + src_{r+1,c})$$

for $1 \leq r \leq m-1$ and $1 \leq c \leq n-1$.



Compute ranges of rows of matrix: matrix computation

- restrict computation to rows *r* between `rowstart` and `rowend-1`:

```
static void matrix_eval(double **dest,double **src,
                        unsigned long rowstart,
                        unsigned long rowend,
                        unsigned long columns)
{
  unsigned long r, c;
  for (r = rowstart; r < rowend; r++)
    for (c = 1UL; c < columns; c++)
      dest[r][c] = 0.25 * (src[r][c-1] +
                           src[r][c+1] +
                           src[r-1][c] +
                           src[r+1][c]);
}
```

Compute ranges of rows of matrix: matrix copying

- add a function to copy the matrix

```
static void matrix_copy(double **dest,
                        double **src,
                        unsigned long rowstart,
                        unsigned long rowend,
                        unsigned long columns)
{
  unsigned long row, col;

  for (row = rowstart; row < rowend; row++)
  {
    for (col = 1UL; col < columns; col++)
    {
      dest[row][col] = src[row][col];
    }
  }
}
```


Compute ranges of rows of matrix: iteration

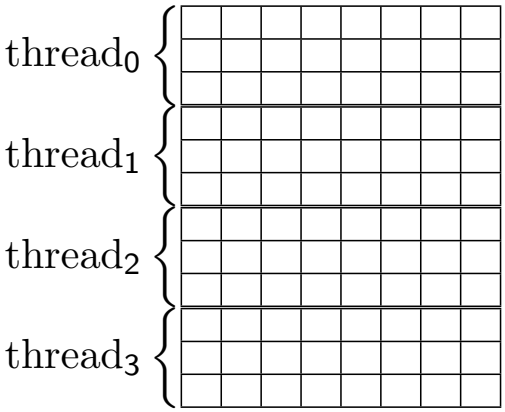
```
static void matrix_eval_seq_iter(unsigned long iterations,
                                 double **matrix,
                                 unsigned long rows,
                                 unsigned long columns)
{
  unsigned long iter;
  double **temp;

  array2dim_malloc(temp, rows+1, columns+1);
  for (iter = 0; iter < iterations; iter++)
  {
    matrix_eval(temp,matrix,1UL,rows,columns);
    matrix_copy(matrix,temp,1UL,rows,columns);
  }
  array2dim_delete(temp);
}
```

Threading the matrix computations

- divide the rows into parts of (nearly) the same size
- each thread has access to the entire src-matrix and dest-matrix (shared data) but only processes its own part (private data)

$\Rightarrow$ put the values referring to the matrices into own structure and refer to it in the private part

- pass private part (including row range and threadid) to each thread



Compute ranges of rows of matrix: shared and private data

- put the data shared between the threads into a structure:

```
typedef struct
{
  unsigned long columns;
  double **matrix, **temp;
} Matrixresources;
```

- and add a structure for the private data describing the part of the computation done by the thread:

```
typedef struct
{
  unsigned long rowstart, rowend; /* range of rows */
                                  /* processed by thread */
  pthread_t ident;                /* with this identifier */
  Matrixresources *mrptr;         /* ref. to shared data */
} Partinfo;
```

Compute ranges of rows of matrix: thread functions

- put matrix computations into thread functions of appropriate type:

```
static void *matrix_eval_threadfun(void *vpartinfo)
{
  Partinfo *partinfo = (Partinfo *) vpartinfo;

  matrix_eval(partinfo->mrptr->temp,partinfo->mrptr->matrix,
              partinfo->rowstart,partinfo->rowend,
              partinfo->mrptr->columns);
  return NULL;
}

static void *matrix_copy_threadfun(void *vpartinfo)
{
  Partinfo *partinfo = (Partinfo *) vpartinfo;

  matrix_copy(partinfo->mrptr->matrix,partinfo->mrptr->temp,
              partinfo->rowstart,partinfo->rowend,
              partinfo->mrptr->columns);
  return NULL;
}
```


Compute ranges of rows of matrix: initialize the partinfo-table

```
#define MIN(X,Y)  ((X) < (Y) ? (X) : (Y))
static Partinfo *partinfo_new(unsigned long numthreads,
                              Matrixresources *mrptr,
                              unsigned long rows)
{
  unsigned long t, width = rows/numthreads;
  Partinfo *partinfo_tab;

  partinfo_tab = malloc(sizeof *partinfo_tab * numthreads);
  assert(numthreads > 1UL && partinfo_tab != NULL);
  for (t = 0; t < numthreads; t++)
  {
    partinfo_tab[t].mrptr = mrptr; /* shared resource */
    partinfo_tab[t].rowstart = 1UL + t * width;
    partinfo_tab[t].rowend = MIN(1UL + (t+1) * width,rows);
  }
  return partinfo_tab;
}
```

Compute ranges of rows of matrix: thread creation/joining

```
static void matrix_eval_step(Partinfo *partinfo_tab,
                             unsigned long numthreads)
{
  unsigned long t;

  for (t = 0; t < numthreads; t++)
    pthread_create(&partinfo_tab[t].ident,NULL,
                   matrix_eval_threadfun,partinfo_tab + t);
  for (t = 0; t < numthreads; t++)
    pthread_join(partinfo_tab[t].ident,NULL);
  /* Now all rows of the matrix are computed and program runs
     with a single thread, before it is threaded again. */
  for (t = 0; t < numthreads; t++)
    pthread_create(&partinfo_tab[t].ident,NULL,
                   matrix_copy_threadfun,partinfo_tab + t);
  for (t = 0; t < numthreads; t++)
    pthread_join(partinfo_tab[t].ident,NULL);
}
```


Compute ranges of rows of matrix: iteration with threaded computation of matrices

```
static void matrix_eval_threaded_iter(unsigned long numthreads,
                                      unsigned long iterations,
                                      double **matrix,
                                      unsigned long rows,
                                      unsigned long columns)
{
  Matrixresources mr;
  Partinfo *partinfo_tab;
  unsigned long iter;

  mr.matrix = matrix;
  mr.columns = columns;
  array2dim_malloc(mr.temp, rows+1, columns+1);
  partinfo_tab = partinfo_new(numthreads, &mr, rows);
  for (iter = 0; iter < iterations; iter++)
    matrix_eval_step(partinfo_tab,numthreads);
  free(partinfo_tab);
  array2dim_delete(mr.temp);
}
```

Compute ranges of rows of matrix: the main function

```
int main(int argc, char *argv[])
{
  unsigned long numthreads, iterations, rows, columns;
  double **matrix;

  if (argc != 5) usage();
  numthreads = parse_arg(argv[1]); /* numthreads <- argv[1] */
  iterations = parse_arg(argv[2]); /* iterations <- argv[2] */
  rows = parse_arg(argv[3]);       /* rows <- argv[3] */
  columns = parse_arg(argv[4]);    /* columns <- argv[4] */
  srand48(366292341);           /* init random num generator */
  matrix = matrix_new_random(rows,columns);
  if (numthreads == 1UL || numthreads >= rows)
    matrix_eval_seq_iter(iterations, matrix, rows, columns);
  else
    matrix_eval_threaded_iter(numthreads,iterations, matrix,
                              rows, columns);
  matrix_show(matrix,rows, columns);
  array2dim_delete(matrix);
  return EXIT_SUCCESS;
}
```


Compute ranges of rows of matrix: evaluation

- run matrix program with 1000 iterations for a $(10000+1) \times (100+1)$ matrix and varying number of threads
- determine running time (for Intel I7 with 4 cores) depending on number of threads

# threads	running time (s)
1	2.464
2	2.236
3	2.161
4	2.135

- each of the n iterations for $t > 1$ threads means to create and join $2nt$ threads
- overhead is too large to obtain a good speed-up

The problem of data races

- sometimes different threads access the same shared variable

```
int s = 0, *a = array_new(n,...);
```

thread 1	thread 2
`int i;`	`int i;`
`for (i = 0; i <= n/2-1; i++)`	`for (i = n/2; i<n; i++)`
`  s += f(a[i])`	`  s += f(a[i])`

- problem: a race condition on variable `s`
- a race condition or data race occurs when:
  - two threads access the same variable, and at least one does a write
  - the accesses are concurrent (not synchronized) so they could happen simultaneously

Mutexes as basic type for Synchronization

- Pthreads provides mutexes to synchronize threads accessing shared data
- mutexes – mutual exclusion (also called locks)
- to create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
/* or pthread_mutex_init(&amutex, NULL); */
```

- to use a mutex

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- to deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

A simple example with Mutexes

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

typedef struct
{
  pthread_mutex_t mutex1;
  unsigned long counter;
} Sharedvars;

static void *increment(void *data)
{
  Sharedvars *sv = (Sharedvars *) data;

  pthread_mutex_lock(&sv->mutex1);
  sv->counter++;
  pthread_mutex_unlock(&sv->mutex1);
  return NULL;
}
```


A simple example with Mutexes

```
int main(void)
{
  const unsigned long numthreads = 10UL;
  unsigned long t;
  pthread_t threadid_tab[numthreads];
  Sharedvars sv = {PTHREAD_MUTEX_INITIALIZER,0};

  /* Create independent threads each of which
     will execute increment */

  for (t = 0; t < numthreads; t++)
    pthread_create(&threadid_tab[t], NULL, increment,
                   (void *) &sv);

  for (t = 0; t < numthreads; t++)
    pthread_join(threadid_tab[t], NULL);
  printf("final value = %lu\n",sv.counter);
  exit(EXIT_SUCCESS);
}
```

Computing the dot-product using mutexes

- The dot product of two vectors $a = (a_1, a_2, \ldots, a_n) \in \mathbb{R}^n$ and $b = (b_1, b_2, \ldots, b_n) \in \mathbb{R}^n$ is defined by

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

- compute the dot product using threads each of which sums up the products from a part of a and b.
- the sum of each part (of length `veclen`) is stored in private variable.
- before the thread finishes it adds value of this variable to overall sum: this requires a mutex
- declarations for the shared data and the info provided to threads:

```
typedef struct
{
  double *a, *b, sum;
  unsigned long veclen;
  pthread_mutex_t mutexsum;
} Shareddata;
```

```
typedef struct
{
  Shareddata *sd_ptr;
  unsigned long threadnum;
  pthread_t thread_id;
} Threadinfo;
```


Computing the dot-product using mutexes

```
static void *dotprod(void *vthreadinfo)
{
  Threadinfo *threadinfo = (Threadinfo *) vthreadinfo;
  unsigned long idx, start, end;
  double threadsum = 0.0;

  start = threadinfo->threadnum * threadinfo->sd_ptr->veclen;
  end = start + threadinfo->sd_ptr->veclen;
  for (idx = start; idx<end ; idx++)
    threadsum += (threadinfo->sd_ptr->a[idx] *
                  threadinfo->sd_ptr->b[idx]);
  pthread_mutex_lock (&threadinfo->sd_ptr->mutexsum);
  threadinfo->sd_ptr->sum += threadsum;
  printf("thread␣%lu␣[%7lu,%7lu]:␣threadsum=%.2f␣sum=%.2f\n",
         threadinfo->threadnum,start,end,threadsum,
         threadinfo->sd_ptr->sum);
  pthread_mutex_unlock (&threadinfo->sd_ptr->mutexsum);
  return NULL;
}
```

Computing the dot-product using mutexes

```
static Shareddata *shareddata_new(unsigned long numthreads,
                                  unsigned long veclen)
{
  unsigned long idx;
  Shareddata *sd = malloc(sizeof *sd);

  assert(sd != NULL);
  sd->a = malloc (numthreads * veclen * sizeof *sd->a);
  sd->b = malloc (numthreads * veclen * sizeof *sd->b);
  assert(sd->a != NULL && sd->b != NULL);
  srand48(366292341);
  for (idx = 0; idx < veclen * numthreads; idx++)
  {
    sd->a[idx] = drand48();
    sd->b[idx] = drand48();
  }
  sd->veclen = veclen;
  sd->sum = 0;
  pthread_mutex_init(&sd->mutexsum, NULL);
  return sd;
}
```


Computing the dot-product using mutexes

```
static void shareddata_delete(Shareddata *sd)
{
  if (sd != NULL)
  {
    free(sd->a);
    free(sd->b);
    pthread_mutex_destroy(&sd->mutexsum);
    free(sd);
  }
}
```

- threads have attributes, which can be changed by the user
- to change the stack size for a thread to 8192 (before calling `pthread_create`), do this:
  `pthread_attr_setstacksize(&attr, (size_t) 8192);`
- to guarantee that a thread is created as joinable, call
  `pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);`
  and supply `&attr` to `pthread_create`

Computing the dot-product using mutexes

```
unsigned long idx;
const unsigned long numthreads = 4, veclen = 100000UL;
Threadinfo threadinfo[numthreads];
pthread_attr_t attr;
Shareddata *shareddata;

shareddata = shareddata_new(numthreads,veclen);
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
for (idx=0; idx<numthreads; idx++)
{
  threadinfo[idx].threadnum = idx;
  threadinfo[idx].sd_ptr = shareddata;
  pthread_create(&threadinfo[idx].thread_id, &attr, dotprod,
                 (void *) &threadinfo[idx]);
}
pthread_attr_destroy(&attr);
for (idx = 0; idx < numthreads; idx++)
  pthread_join(threadinfo[idx].thread_id, NULL);
printf ("sum = %.2f \n", shareddata->sum);
shareddata_delete(shareddata);
```


Computing the dot-product using mutexes

```
$ thread-dotprod.x
thread 1 [ 100000, 200000]: threadsum=25005.63 sum=25005.63
thread 0 [      0, 100000]: threadsum=24931.58 sum=49937.21
thread 2 [ 200000, 300000]: threadsum=25074.60 sum=75011.81
thread 3 [ 300000, 400000]: threadsum=24909.43 sum=99921.25
sum = 99921.25
$ thread-dotprod.x
thread 3 [ 300000, 400000]: threadsum=24909.43 sum=24909.43
thread 2 [ 200000, 300000]: threadsum=25074.60 sum=49984.03
thread 1 [ 100000, 200000]: threadsum=25005.63 sum=74989.66
thread 0 [      0, 100000]: threadsum=24931.58 sum=99921.25
sum = 99921.25
$ thread-dotprod.x
thread 0 [      0, 100000]: threadsum=24931.58 sum=24931.58
thread 1 [ 100000, 200000]: threadsum=25005.63 sum=49937.21
thread 2 [ 200000, 300000]: threadsum=25074.60 sum=75011.81
thread 3 [ 300000, 400000]: threadsum=24909.43 sum=99921.25
sum = 99921.25
```

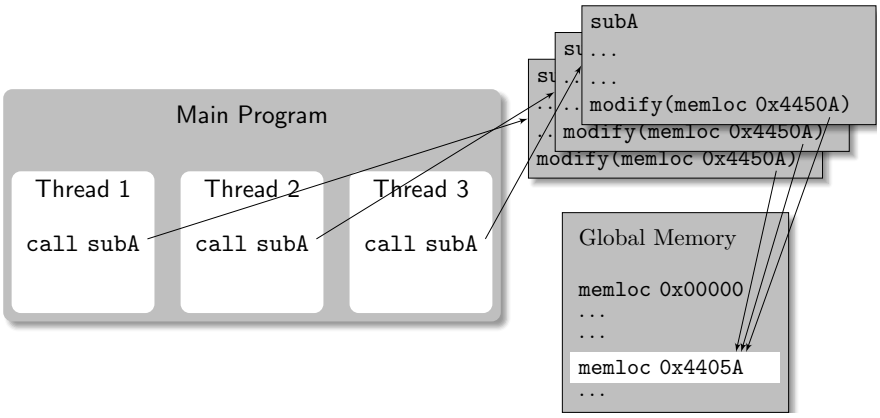
- note that the order in which the parts are summed differs in each of the three executions

Threads and software libraries

- many widely used libraries have parts which work with global variables, e.g. the hash table implementation of the standard C-lib:

```
#include <search.h>
int hcreate(size_t nel);
void hdestroy(void);
ENTRY *hsearch(ENTRY item, ACTION action);
```

- functions have no pointer to hash table ⇒ hash table is global variable
- such implementations are not thread-safe: cannot be used with threads



Recommendation

use an external library routine only if you are 100% certain that it is thread-safe

Summary of Programming with Threads

- POSIX Threads are based on OS features
- user has explicit control over thread
- good: control can be used for performance benefit
- bad: user has to deal with it
- threads can be used not only in C, but from multiple languages
- ability to access shared data is convenient
- pitfalls
  - data race bugs are very difficult to find because they occur non-deterministically
- use libraries only if they are thread-safe

C++ for Java Programmers, a tutorial

Stefan Kurtz

Center for Bioinformatics Hamburg
University of Hamburg

sections 1-4 and 8 are adapted from original slides of Uwe Rastofer, University of Erlangen

June 4, 2015

Index I

Index II

A Short History of C++ (Self Study)

- 1980: Dennis Ritchie extends C to C with Classes
- 1983: Bjarne Stroustrup introduces C++ V1.0
- 1986: Bjarne Stroustrup publishes *The C++ Programming Language* (1st Edition)
- 1989: ANSI approves Standard C with elements from C++
- 1989: ANSI committee X3J16 begins standardization of C++ (V2.0)
- 1991:The Annotated C++ Reference Manual defines C++ V3.0 including *Templates* and *Exceptions*
- 1993: C++ V3.1 includes *Namespaces* and *Run-Time Type Identification*
- 1997: ISO WG21 and ANSI X3J16 adopt C++ and the Standard Template Library (STL) as standard ISO/IEC FDIS 14882

What is C++?

- super-set of C with
  - strong typing
  - prototypes
  - overloading of functions and standard operators
- extends C to include object-oriented concepts
  - Objects
  - Classes
  - Inheritance
  - Polymorphism
- BUT: C++ does not enforce an object-oriented style of programming (in contrast to Java)

- Version used in exercises: C++98.
- see man page for gcc on ZBH, newer standards are not fully supported.

Literature (Self Study)

- Bjarne Stroustrup: *Die C++ Programmiersprache*. 4. Auflage, Addison-Wesley, Reading MA, 2010.
- *ANSI C++ Public Comment Draft*, December 1996. See tutorial web page
- Scott Meyers: *Effective C++*, Third Edition, Addison-Wesley, Reading MA, 2005.
- Scott Meyers: *More Effective C++*, Addison-Wesley, Reading MA, 1995.
- Harvey M. Deitel, Paul J. Deitel. *C++ – How to program*. 8th Edition, Prentice-Hall, 2011.

Input and output

- input from and output to streams via operators

`cin`	input stream (global)
`cout, cerr, (clog)`	output stream (global)
`>>`	input operator
`<<`	output operator

```
#include <iostream>
int main(void)
{
  int readint; // I/O variable
  std::cin >> readint;
  std::cout << "readint=" << readint << "\n";
  return 0;
}
```

- Problem: program does not recognize invalid input:

```
printf "0"| main-mini.x
readint=0
```

```
printf "abracra"| main-mini.x
readint=0
```

Scope operator

- new operator :: for accessing scopes ("Geltungsbereiche")
- mainly used with classes and namespaces
- intention: if a variable is declared and there is a variable with the same name in an outer context, then the latter is hidden
- prepending the ::-operator makes it visible in the inner context
- *here*: access hidden variables with the same identifier in other scopes

```
#include <iostream>

int test = 4711;    // global variable

int main(void)
{
  int test = 1234; // local variable
  std::cout << "value␣of␣global␣var.:␣" << ::test << "\n";
  std::cout << "value␣of␣local␣var.:␣" << test << "\n";
}
```


Namespaces I

- new keyword namespace:
  namespace *namespace_name* {
  declarations/definitions
  }
- opens a new namespace for identifiers
- intention: different namespaces can contain identical identifiers
- using the identifiers with their name space prevents name clashes
- namespaces can be nested
- access via scope operator ::
- like package in Java

```
namespace Date {
  typedef struct {
    int hour,
        minute;
  } Time;
}
```

```
int main() {
  Date::Time today;
  today.hour = 14;
  today.minute = 15;
  today.minute++;
  return EXIT_SUCCESS;
}
```

Namespaces II

- import identifiers from other name spaces via `using`:
  `using` *namespace_name*`::`*identifier*`;`
  - Like `import package.`*identifier*`;` in Java
- import of complete name spaces:
  `using namespace` *namespace_name*`;`
  - Like `import package.*;` in Java

```
namespace Date {
  typedef struct {
    int hour,
        minute;
  } Time;
}
```

```
using namespace Date;

int main() {
  Time today;
  today.hour = 14;
  today.minute = 15;
  today.minute++;
  return EXIT_SUCCESS;
}
```


Memory management I

- recall: C-memory management with functions `malloc`, `calloc`, `realloc` and `free`
- C++: two operators `new` and `delete` for memory management
  - memory allocation with `new`
    ```
    type *pointer_to_type; /*declaration of variable */
    pointer_to_type = new type; /*allocation */
    ```
    - if allocation fails a `std::bad_alloc` exception is thrown and a `NULL` pointer is returned (see example below)
  - memory deallocation with `delete`
    ```
    delete pointer_to_type;
    ```
    - programmer is responsible for deallocation (no garbage collection in C++)
    - pointer is still accessible after deallocation

    $\Rightarrow$ common source of programming errors
    - `delete` for a `NULL` pointer is allowed

Memory management II

```
/* from http://www.cplusplus.com/reference/new/bad_alloc/ */
#include <iostream>      // std::cout
#include <new>           // std::bad_alloc

int main (__attribute__ ((unused)) int argc,char **argv)
{
  try {
    const unsigned long len = 1UL << 34; /* 2**34 */
    int *myarray = new int[len];
    for (unsigned long idx = 0; idx < len; idx++)
      myarray[idx] = 0;
    delete[] myarray;
  }
  catch (std::bad_alloc) {
    std::cerr << argv[0] << ": bad_alloc exception caught\n";
  }
  return 0;
}
```


Memory management III

- no double deletions allowed:

```
int *x = NULL;    // okay
delete x;         // okay, NULL-ptr can be deleted
x = new int;      // okay
delete x;         // okay
delete x;         // error, double deletion
```

- special syntax for arrays:

```
const unsigned long entries = 7;
unsigned long *squares = new unsigned long[entries];

for (unsigned long i=0; i<entries; i++)
  squares[i] = i * i;
delete[] squares;
```

- never ever mix `malloc`/`free` with `new`/`delete`
  that means: do not use `delete` on pointers referring to memory delivered by `malloc`, `calloc` or `realloc`; no `free` for `new`ed memory.

⇒ caution: some functions (like `strdup`) implicitly call `malloc`.

Example: Allocation with new/delete and malloc/free

```
int main(int argc,char **argv)
{
  char **argvcopy = new char * [argc];

  for (int idx = 0; idx < argc; idx++)
  {
    argvcopy[idx] = strdup(argv[idx]);
  }
  for (int idx = 0; idx < argc; idx++)
  {
    std::cout << "argv[" << idx << "]=" << argv[idx] << "\n";
  }
  for (int idx = 0; idx < argc; idx++)
  {
    free(argvcopy[idx]);
  }
  delete[] argvcopy;
  return EXIT_SUCCESS;
}
```


Function overloading

- same function name for different implementations
- works for pure C functions and C++ methods
- overloaded functions are distinguished by:
  - number of parameters
  - type of parameters
  - order of parameter types
  - *not*: return type of function (as return value may be ignored)

```
void print(); //okay
void print(int, char *); //okay
void print(int d, const char *s); //okay
int print(float); //okay
int print(); //error: not distinguishable from first
int print(int, char *); //error: not distinguishable from second
```

Reference variables I

- address operator `&` in variable declaration

  `type &reference_variable = variable_of_type;`
- reference variable
  - no real variables
  - alias for another variable
  - must be initialized during declaration (with *lvalue* – an expression that can be on the left side of an assignment, i.e. it can take a value)

```
int x = 5; //variable
int &rx = x; //reference to x
x = 6; //x==6 and rx==6
rx++; //x==7 and rx==7
```

- operations on reference variables affect the referenced variables
- similar to pointers with implicit dereferencing but less flexible

Reference variables II

- reference parameters
  - allow implicit call-by-reference semantics
  - no pointers necessary
  - caller writes call with normal syntax
  - disadvantage: syntax of call does not show semantics

```
#include <iostream>

void increment(int &x)
{
  x++;
}

int main(void)
{
  int x = 5;
  increment(x); // call by reference not visible
  std::cout << "x=" << x << "\n"; // x==6
}
```

Reference variables III

- returning references is also possible

```
int global = 0;
int &func(void) {
  return global; // returns reference to global
}

int main(void) {
  int x = func() + 1; // x = global + 1
  func() = 3;       // global = 3
  std::cout << "x=" << x << ", global=" << global << "\n";
}
```

- function returns a variable (lvalue), but not a value
- returning references to local variables is forbidden

```
int &func() {
  int x = 0;
  int &rx = x;
  return rx;     // forbidden
}
```


Default parameters

- function parameters may contain default values
- will be used when the actual parameter in a call is missing
  ⇒ only at the end of the parameter list, no gaps allowed

```
void print(char *string, int nl = 1);
print("Test",0);
print("Test");    // is equal to print("Test",1)
print();          // wrong, char * parameter is missing
```

- caution: function overloading and default parameters may generate ambiguities

```
void print(char *string)

{...}

void print(char *string,
           int nl = 1)

{...}

print("Test");
```

generates compiler error:

```
default-ambig.cpp:18:1: error: call to 'print' is ambiguous
print("Test");
^~~~~
default-ambig.cpp:4:6: note: candidate function
void print(char *string)
     ^
default-ambig.cpp:9:6: note: candidate function
void print(char *string,
     ^
1 error generated.
```

Constants

- reserved word `const` modifies declaration
  - const variables are read-only (final in Java)
  - initialization during declaration

```
const int k = 42;
char *const s1 = "Test1";
const char *s2 = "Test2";
const char *const *s3 = "Test3";

k = 4;              // error: k ist const
s1 = "New test";    // error: pointer is const
*s1 = 'P';          // okay, characters are not const
s2 = "New test";    // okay, pointer is not const
*s2 = 'P';          // error: characters are const
```

- should be preferred to `#define`, because managed by the compiler
  - definition of local constants
  - pointer to constants possible (like pointers to variables)

Classes

- class declaration in C++ with reserved word `class`:
  `class` *class_name*
  {
  declaration of member variables and functions
  };
- contains declaration of data and methods (in C++ called *members*)
- In C++, sending a message means accessing a member

```
class Person
{
  char *name;
  int age;
  void setName(char *);
  void setAge(int);
};
```

Visibility

- different visibility for parts of an object:
  - `private` member can be accessed only from within its class
  - `public` member can be accessed from anywhere
  - `protected` like private, but subclasses have access
- parts can be declared in any order and can be repeated
- `public` parts are the interface for other objects
- default visibility is `private`

```
class Person
{
  private: char *name;
           int age;
  public: void setName(char *);
          void setAge(int);
};
```


Object creation

- syntax is the same like declaring a variable
- static creation:

```
Person peter;
Person john;
```

  - object deleted when identifier goes out of scope
- dynamic creation:

```
Person *peter;
peter = new Person; //object is created now
```

  - object explicitly deleted

```
delete peter; //object is deleted now
```

Object access

- access from outside the object
  - private member variables and functions are not accessible
  - public member variables and functions are accessible
- access operators
  - as in `structs` with the dot operator `.`
  - with pointers to objects use the arrow operator `->`

```
Person peter;
Person *john = new Person;

peter.setName("Peter␣Smith"); // okay, public
std::cout << peter.name;      // error, private
john->setAge(35);             // okay, public
std::cout << john->age;       // error, private
delete john;
```


Member functions (methods) I

- definition *within* the `class` declaration:
  - function body comes directly after the declaration (as in Java)
  - function becomes automatically `inline`
  - usually used in header files (`.hpp`, `.H` or `.hh`)
  - we use the suffix `.hpp`
- definition *outside* the class:
  - within the class only declaration of the function prototype
  - during definition first name the class
  - this is followed by the function name separated by scope operator `::`
  - usually used in implementation files (`.C`, `.cc` or `.cpp`)
  - we use the suffix `.cpp`

Member functions (methods) II

- header (person.hpp)

```
class Person {
  private:
    char *name;
    int age;

  public:
    void setName(char *n) { // inline
            this->name = n;
    }
    void setAge(int);
};
```

- implementation (person.cpp)

```
#include "person.hpp"

void Person::setAge(int i) { this->age = i; }
```


Constant Objects (Self Study) I

- variable declared `const`
  - initialized when declared
  - cannot be changed afterwards
- silly example:

```
const Person nobody;
```

- only operations that do not alter the object may be executed
  - easy for member variable access
  - methods that do not alter members
- how does the compiler know?
  - it does not!
  - needs a hint from the programmer

Constant Objects (Self Study) II

- methods may be declared `const`
- const methods do not change the object they are called at

```
class Person
{
  private:
    char *name;
    int age;

  public:
    int getAge(void) const
    {
      return this->age;
    }
};
```


Constructors I

- constructors are class methods
- name for constructor is the name of the class
- constructor has *no* return type (not even void)
- obtain different constructors via overloading
- declaration usually in the `public` part of the class
- purpose: new object is automatically initialized after creation
  $\Rightarrow$ constructor has to put object in a consistent state
- compiler creates a minimal default constructor (no arguments) if not declared in class

Constructors II

- the constructor is called during:
  - creation of an object via the operator `new`
  - creation of a static object
- minimal default constructor (created by the compiler):

```
Person::Person() {}
```

- constructor with minimal initialisation (replaces minimal constructor):

```
Person::Person()
{
  this->name = NULL;
  this->age = 0;
}
```

- other constructor with different signature:

```
Person::Person(char *n, int i = 0)
{
  this->name = n;
  this->age = i;
}
```


Destructors

- destructors are similar to `finalize` in Java
- a destructor is a class method
- name of destructor is name of the class with ~ (tilde) in front
- *no* return type (not even void) and *no* parameters
- only one destructor possible, no overloading
- declaration usually in the `public` part of the class
- purpose: cleaning up before deleting the object
- minimal default destructor (created by the compiler):

```
Person::~Person() {}
```

- destructor is called
  - when an object is deleted via the operator `delete`
  - when leaving the scope of a static object

Member objects I

- objects of other classes as members within a class

```
class Workplace
{
  Person worker; ...
};
```

- access via operators . and -> as usual
- during initialization of the class:
  - constructors of the member objects are called.
  - sometimes it is necessary to add initializations of the member-object

```
Workplace::Workplace(char *name)
{
  this->worker.setName(name);
}
```


Copy constructor

- when copying objects we need copy constructors, e.g. to initialize an object with an existing object

```
Person peter(john); //call the copy constructor declared below
```

```
Person::Person(const Person &p)
{
  this->name = p.name;
  this->age = p.age + 1;
}
```

- important: use reference operator `&` for parameter
- default copy constructor (created by the compiler) copies bit-by-bit

Arrays of objects I

in arrays of objects, the elements are initialized by constructors:

- static arrays
  - without initialization ⇒ for all elements the standard constructor is called

```
Person test[4]; //calls Person::Person() 4 times
```

- with initialization

```
Person test[4] =
{
  Person("Peter"), Person("John")
};
```

  - for the first two elements, initialization expressions are used:
    `test[0]` and `test[1]`: `Person::Person(char *)`
  - for the rest, the standard constructor is called:
    `test[2]` and `test[3]`: `Person::Person()`

Arrays of objects II

- for dynamically allocated arrays always the default constructor is called

```
Person *table;
table = new Person[4];
```

- here `Person::Person()` is called 4 times
- access to objects in array as usual via operator `[]`

```
Person table[4];
table[0].setName("Peter");
```

- destruction of arrays
  - for all elements the destructor is called
  - dynamically allocated arrays have to be deleted via `delete[]`

Example: Representing points in two dimensions

- point.hpp:

```
class Point {
  private: long xcoord, ycoord;
  public: void setX(long);
          void setY(long);
          double distanceTo(Point);
};
```

- point.cpp:

```
#include "point.hpp"
void Point::setX(long x) { xcoord = x; }
void Point::setY(long y) { ycoord = y; }
double Point::distanceTo(Point p)
{
  return sqrt(pow(xcoord - p.xcoord,2.0) +
              pow(ycoord - p.ycoord,2.0));
}
```


Example: Representing lines between two points I

```
class Line
{
  public: Point start, end;
          double length;
};
```

- this implementation violates the DRY (don't repeat yourself) principle, as `length` can be derived from `start` and `end`
- better compute the line-length from the points (and do not store redundant information):

```
class Line
{
  public: Point start, end;
          double length() {
            return start.distanceTo(end);
          }
};
```

Example: Representing lines between two points II

- for efficiency reasons, one sometimes sacrifices the DRY principle:

```
class Line {
  private: bool changed; /* has point been changed? */
           Point start, end;
           double length; /* update after change */
  public: void setStart(Point p) { this->start = p;
                                   this->changed = true;}
    void setEnd(Point p) { this->end = p;
                           this->changed = true;}
    Point getStart(void) { return this->start; }
    Point getEnd(void) { return this->end; }
    double getLength(void) {
      if (this->changed) {
        this->length = this->start.distanceTo(this->end);
        this->changed = false;
      }
      return this->length;
    }
};
```


Recall: computation of machine accuracy (in C)

```
float accuracy_float(void) {
  float eps;
  for (eps = (float) 1.0;
       (float) 1.0 + eps > (float) 1.0;
       eps /= (float) 2.0)
   /* Nothing */;
  return eps;
}

double accuracy_double(void) {
  double eps;
  for (eps = (double) 1.0;
       (double) 1.0 + eps > (double) 1.0;
       eps /= (double) 2.0)
   /* Nothing */;
  return eps;
}

long double accuracy_long_double(void) {
  long double eps;
  for (eps = (long double) 1.0;
       (long double) 1.0 + eps > (long double) 1.0;
       eps /= (long double) 2.0)
   /* Nothing */;
  return eps;
}
```

```
int main(void)
{
  float f = accuracy_float();
  double d = accuracy_double();
  long double ld = accuracy_long_double();
  printf("accuracy float=%e\n",f);
  printf("accuracy double=%e\n",d);
  printf("accuracy long double=%Le\n",ld);
  return EXIT_SUCCESS;
}
```

```
accuracy float=5.960464e-08
accuracy double=1.110223e-16
accuracy long double=5.421011e-20
```

Computation of machine accuracy (in C++)

```
template <typename T>
T accuracy(void)
{
  T eps;
  for (eps = (T) 1.0; (T) 1.0+eps > (T) 1.0; eps /= (T) 2.0)
   /* Nothing */;
  return eps;
}

int main(void)
{
  float f = accuracy<float>();
  double d = accuracy<double>();
  long double ld = accuracy<long double>();
  std::cout << "accuracy float=" << f << "\n";
  std::cout << "accuracy double=" << d << "\n";
  std::cout << "accuracy long double=" << ld << "\n";
}
```


The concept of templates I

Function templates

- are special functions that can operate with generic types
- functionality can be adapted to more than one type or class
- no repetition of code for each type
- effect is achieved via template parameters:
  - a special kind of parameter that can be used to pass a type as argument
  - parameters can be used as if they were any regular type
- C++-compiler replaces template parameters by concrete types at compile time

⇒ program can be highly optimized by compiler

⇒ templates provide a mechanism for type-abstraction without increasing the runtime of the program

section taken from `http://www.cplusplus.com/doc/tutorial/templates/`

The concept of templates II

- format:

```
template <typename identifier> function_declaration
```

- example: template function with mytype as template parameter

```
template <typename mytype>
mytype getmax (mytype a, mytype b) {
  return a > b ? a : b;
}
```


Use of template functions I

- format for using the function template:

```
function_name <type> (parameters);
```

- example: use `getmax`

```
int main(void)
{
  int i = 5, j = 6, k;
  long l = 10, m = 5, n;
  k = getmax<int> (i,j);
  /* <int> is optional */
  n = getmax<long> (l,m);
  /* <long> is optional */
  cout << k << "\n";
  cout << n << "\n";
  return 0;
}
```

- example: cannot mix types

```
int i;
long l;
k = getmax (i,l);
// wrong
```

A generic matrix implementation with templates I

Recall matrix implementation in C:

```
int **int_matrix_new(unsigned long rows,
                     unsigned long cols)
{
  unsigned long idx;
  /* row pointers: */
  int **matrix = malloc(sizeof *matrix * rows);
  /* values: */
  matrix[0] = malloc(sizeof **matrix * rows * cols);
  for (idx = 1UL; idx < rows; idx++)
    matrix[idx] = matrix[idx-1] + cols;
  return matrix;
}

void int_matrix_delete(int **matrix)
{
  if (matrix != NULL)
  {
    free(matrix[0]); /* free array of values */
    free(matrix);    /* free array of row pointers */
  }
}
```

- for other types:
  - either copy functions and replace type or
  - use macros

A generic matrix implementation with templates II

```
template <typename T>
T **matrix_new(unsigned long rows,unsigned long cols)
{
  T **matrix = new T* [rows];
  matrix[0] = new T [rows * cols];
  for (unsigned long idx = 1; idx < rows; idx++)
  {
    matrix[idx] = matrix[idx-1] + cols;
  }
  return matrix;
}

template <typename T>
void matrix_delete(T **matrix)
{
  if (matrix != NULL) {
    delete[] matrix[0];
    delete[] matrix;
  }
}
```

A generic matrix implementation with templates III

```
void matrix_example(unsigned long m,unsigned long n)
{
  unsigned long row, col;
  double **matrix;

  /* create a m x n double array */
  matrix = matrix_new<double>(m, n);

  /* use matrix in conventional way via matrix[row][col] */
  for (row = 0; row < m; row++) {
    for (col = 0; col < n; col++)
      matrix[row][col] = 3.14 * row * col;
  }
  matrix_delete(matrix);
}
```


Class templates I

- class can have members that use template parameters as types
- example: class storing two elements of any valid type

```
template <class T>
class mypair {
    T values[2]; /* array of 2 values of type template type T */
  public:
    mypair (T first, T second) {
      values[0] = first; values[1] = second;
    }
};
```

- example: declare object of this class storing two `int` values

```
mypair<int> myobject (115, 36);
```

- example: use same class storing two `float` values

```
mypair<double> myfloats (3.0, 2.18);
```

Class templates II

- previous class template `mypair` contains only one member function `mypair`
- the latter has been defined inline within the class declaration
- for each function member declared outside the declaration of the class template, precede that definition with the `template <...>`-prefix, see example on next page


```
template <class T>
class mypair {
  T vala, valb; // this time use two scalar vars for values
  public: mypair (T first, T second) { // inline constructor
            vala = first;
            valb = second;
          }
          T getmax ();                 // forward declaration
};

template <class T>
T mypair<T>::getmax () { // member function of template class
  return vala > valb ? vala : valb;
}
```

Templates in the file structure

when generating codes for templates

- compiler does not treat templates as normal functions or classes
- they are compiled on demand
- the code of a template function is compiled once an instantiation with specific template arguments is used (but not earlier)

⇒ compiler generates a function specifically for those arguments from the template

file structures:

- template class and template functions must be declared and implemented in the same file

⇒ no separation of interface and implementation

- declaration and implementation must appear in header file
- include this in any file that uses the template

Template specialization I

- sometimes we want to define a different implementation for a template when a specific type is passed as template parameter

⇒ declare a specialization of that template

- example: suppose a class called `mycontainer`
  - to store one value of any type
  - with just one member function called `increase` to increment the value
- when it stores values of type `char`, we want to have a completely different implementation with a function member `uppercase`
- declare a class template specialization for that type:

```
template <class T>
class mycontainer {
  T value;
  public: mycontainer (T arg) { value = arg; }
          T increase () { return ++value; }
};
```

Template specialization II

```
// class template specialization for T = char
template <>
class mycontainer <char> {
  char value;
  public: mycontainer (char arg) { value = arg; }
          char uppercase () {
            if (value >='a' && value <= 'z')
              return value - ('a' - 'A');
            return value;
          }
};

int main (void) {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  std::cout << myint.increase() << "\n";
  std::cout << mychar.uppercase() << "\n";
  return 0;
}
```


Template specialization I

- class template specialization uses an empty template parameter list <> and the specialization parameter <char>:

```
template <> class mycontainer <char> ... ;
```

- <char> is the type for which we declare a template class specialization
- note differences between the generic class template and specialization:

```
template <class T> class mycontainer ... ;
template <> class mycontainer <char> ... ;
```

- no inheritance of members from the generic template to the specialization

$\Rightarrow$ we must define all members of the specialization (even those exactly equal to the generic template class)

Overloading operators for own types

- standard operators are usually applied to fundamental types:

```
int a, b, c;
c = a + b;
```

- in addition they can be used to name operations on own types:

```
struct {
  string product;
  float price;
} a, b, c;
a = b + c;
```

does not work yet: no definition of the behavior of our class with respect to addition operation

- here is a list of all operators that can be overloaded:

```
+     -     *     /     =     <     >     +=    -=    *=    /=
<<    >>    <<=   >>=   ==    !=    <=    >=    ++    --    %
&     ^     !     |     ~     &=    ^=    |=    &&    ||    %=
[]    ()    ,     ->*   ->    new   delete new[] delete[]
```

- the following slide shows an example of overloading +

Overloading operators for own types

```
class Mypair {
  public: int x, y;   // pair of integers to store values
          Mypair () {};  // constructor without initialization
          Mypair (int,int);  // constructor
          Mypair operator+ (Mypair); // operator function,
};          // i.e. regular function whose name is standard op

Mypair::Mypair (int a,int b) { x = a; y = b; } // impl. constr.

Mypair Mypair::operator+ (Mypair p) { //implement op. function
  Mypair temp;
  temp.x = x + p.x; temp.y = y + p.y;
  return temp;
}

int main () {
  Mypair a (3,1); Mypair b (1,2); Mypair c = a + b;
  // c = a + b is equivalent to c = a.operator+ (b)
  std::cout << c.x << "," << c.y << "\n"; // 4,3
  return 0;
}
```

Inheritance (Self study) I

- like in Java
- reuse of existing implementations (classes)
- new class *inherits* features from the existing class
- notation:

class that inherits:	subclass
class that is inherited from:	superclass or base class

- in C++: derivation of new classes from existing ones
- derivation/inheritance is a "is-a" relation
- one base class: single inheritance, otherwise multiple inheritance

Inheritance (Self study) II

- syntax:
  `class` subclass : [modifier] superclass1, [modifier] superclass2, ... `{`
  declaration of new member variables and new or re-implemented member functions (methods) `};`
- not inherited:
  - constructors
  - destructor
  - assignment operator

Inheritance (Self study) III

- rule in C++: everything that is not re-implemented, is inherited

```
class Person
{
  ...
  public: void print(void);
          void setName(char *);
};

class Employee : public Person { ...
    public: void print(void);
            void setSalary(float);
};
```

- behaves like

```
class Employee : public Person { ...
    public: void print();  // from Employee
            void setName(char *);   // from Person
            void setSalary(float); // from Employee
};
```


Standard Template Library: Basic notions for Containers

- *container*: holder object that stores a collection of other objects (its elements)
- containers are used to manipulate the stored objects altogether and pass them to other objects
- usually container contains elements of the same type (e.g. a set of numbers, as set of strings, a set of dates etc.)
- this type is considered a property of the container
- usually each container provides member function to insert and delete elements
- *cardinality*: number of elements in container (in most cases: no upper bound on cardinality of container)
- *duplicate*: an element that is contained more than once in a container

STL: Container-classes: Interface and Implementation

- differentiate between interface and implementation of container-classes
- important decisions for interfaces of container classes:
  - handling of duplicates:
    - duplicates are allowed and increase the cardinality
    - duplicates are not allowed and are not inserted
  - handling of order of elements:
    - not considered
    - user defined order
    - implementation defined order

	irrelevant order	user defined order	implementation defined order
duplicates	multiset	list	ordered list
no duplicates	set	sequential set	ordered set

- important decisions when implementing container-classes:
  - data structures to choose (array, linked list, binary tree)
  - efficiency

STL: Efficient implementation of container-classes

- STL: standard template library
- defines different container classes with corresponding iterators
- implemented in early 1990s by Alexander Stepanov
- since 1994 part of the C++-standard
- consider containers `set`, `vector`, `map`
- requires corresponding includes:

```
#include <set>
#include <vector>
#include <map>
```

other possible includes

```
#include <algorithm>
#include <array>
#include <bitset>
#include <deque>
#include <functional>
#include <iterator>
#include <list>
#include <memory>
#include <queue>
#include <stack>
#include <unordered_map>
#include <unordered_set>
```

STL: Operations on containers

- Operations for all containers

`size_type size();`	return cardinality
`bool empty();`	is the container empty?
`void clear();`	delete all elements from container
`iterator begin(), end();`	iterator for enumerating elements

- Operations specific for `std::set`

`std::pair<iterator, bool> insert(E e)`	insert element `e` of type `E`; if not contained, return `true`; if contained do not insert `e` and return `false` and iterator to existing element
`size_type erase(E e)`	delete element `e` of type `E`; if contained return 1, else return 0
`iterator find(E e)`	return iterator on `e` of type `E`, if it is contained

- Operations specific for `std::vector`

`void push_back(E e)`	add element `e` at the end
`E at(size_type i)`	return element at index `i`
`E erase(iterator it)`	eliminate element at iterator position

STL: Overview of containers not discussed here

`array`	fixed size arrays with no overhead
`bitset`	fixed size arrays of bits; set and retrieve certain bits
`deque`	double ended queue: expand and contract on both sides
`forward_list`	single linked list with constant time insert anywhere; only forward iteration
`list`	double linked list with constant time insert anywhere; iteration in both directions
`queue`	container with first in/first access
`stack`	container with first in/last access
`unordered_map`	map storing values with no defined order and efficient random access (hash)
`unordered_set`	set storing values with no defined order and with no duplicates
`unordered_multiset`	set storing values with no order and with possible duplicates

STL: Additional header files

`algorithm`	defines collection of functions for ranges of elements, such as `count`, `find`, `sort`
`functional`	defines template functions for standard operators, e.g. `T plus(T &x,T &y) { return x + y; }`
`memory`	utilities to manage dynamic memory
`utility`	utilities in unrelated domains; e.g. generic `swap`, `make_pair`

STL: typedef makes name of the types readable

- names of types in STL can become very long

$\Rightarrow$ use `typedef` to introduce type synonyms:

```
#include <map>
#include <string>

typedef std::map<std::string,std::string> str_str_map;
```

introduces the type name `str_str_map` as a synonym for a map from strings to strings.

STL: An example using `std::set`

```
#include <iostream>
#include <string>
#include <set>

typedef std::set<std::string> str_set;

int main(void)
{
  str_set words; /* dictionary */
  std::string word;

  while (std::getline (std::cin,word))
  {
    std::pair<str_set::iterator, bool> result;
    result = words.insert(word);
    if (result.second)
      std::cout << "new word \"" << word << "\"\n";
  }
  std::cout << "# different words: " << words.size() << "\n";

}
```

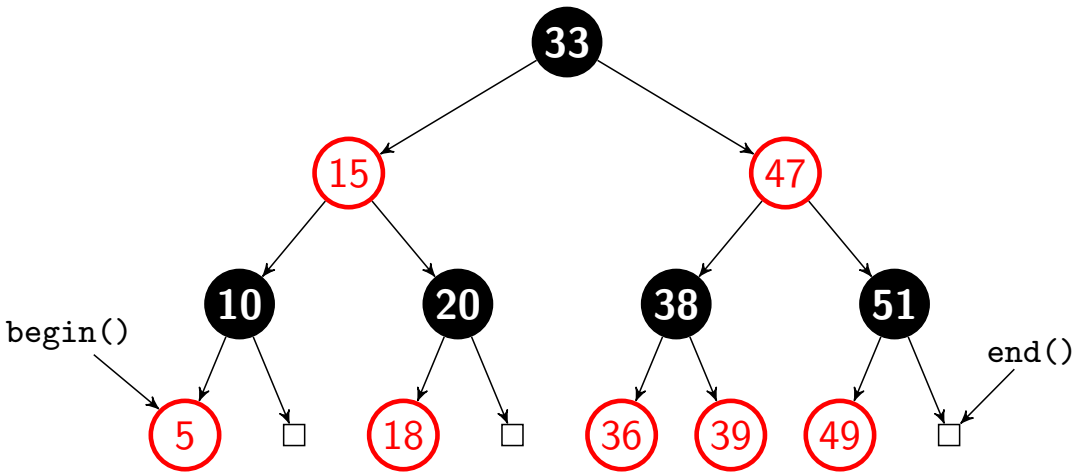

STL: order and iterators

- A set implicitely defines an order of the elements it contains:
  - order according to operator < is default
- `insert` takes care of the fact that no duplicates are inserted:
  - i.e. for two elements a and b in the set either $a < b$ or $b < a$.
- in general, for all container classes in STL, the elements are ordered and the order
  - can depend on the type of the elements (as in `std::set`),
  - or may be defined by the user of the class,
  - or is undefined (i.e. it depends on the implementation)
- for all container classes, one can enumerate the elements according to the given order
- enumeration is performed by the iterator of the STL

STL: iterators

- A pair of iterators defines a sequence by
  - `begin()`-member function (refers to smallest element with respect to order) and
  - `end()`-member function (refers to location after the largest element with respect to order)

suppose `std::set` is implemented by a red-black tree ⇒ `begin()` refers to first element; `end()` refers to first non-occupied element



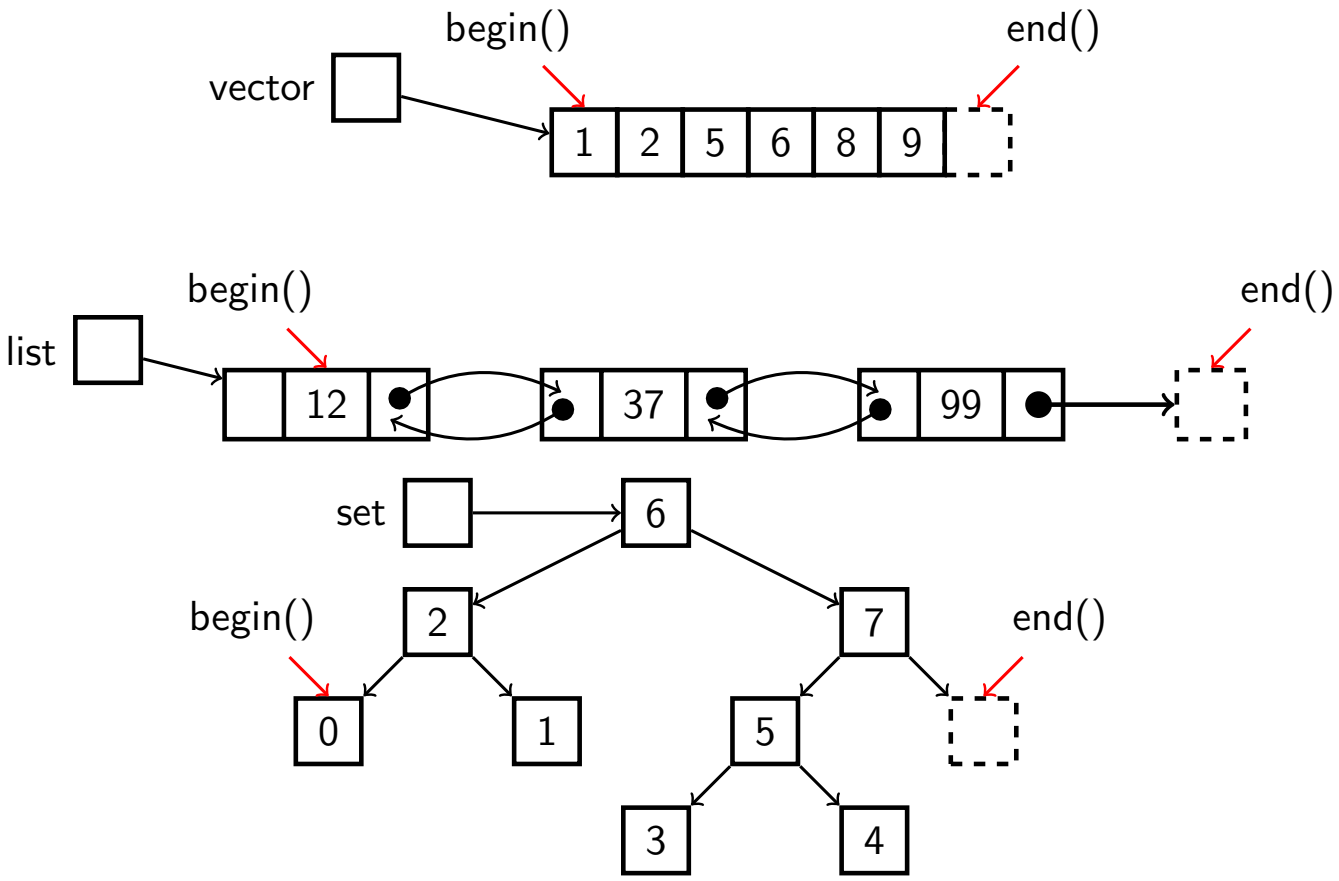
STL: iterators

- possible operations on iterators:
  - `++` move to next element
  - `*` dereference element, the iterator is currently referring to
  - `==` compare iterators, i.e. check if one iterator refers to the same element as another iterator
- example for `str_set words`:

```
for (str_set::iterator it = words.begin();
     it != words.end(); it++)
{
  std::cout << *it << "\n";
}
```

- some iterators provide additional operations like `--`, `+` and `[ ]`

STL: iterators for different containers



STL: classification of the iterators

- the different iterators can be visualized in a hierarchy: the more at the bottom, the more powerful the iterator and the less classes it can be applied to

container class	kind of iterators	possible operations

`unordered_set`
`unordered_map`
`unordered_multiset`
`unordered_multimap`
	forward	`++`

`list`
`set`
`multiset`
`map`
`multimap`
	forward
	bidirectional
		`++ --`

`vector`
`deque`
	forward
	bidirectional
	random access
		`+= -= []`
		`+ - <`
		`> ++ --`

STL: An example using std::vector

```
#include <algorithm>  // for find
#include <iostream>   // for cout
#include <vector>
#include <string>

typedef std::vector<std::string> str_vec;

int main(void)
{
  str_vec shpl; // the shiftplan
  std::cout << "# shifts = " << shpl.size() << "\n"; // 0
  shpl.push_back("Peter");
  shpl.push_back("Paul");
  shpl.push_back("Mary");
  shpl.push_back("Peter");
  shpl.push_back("John");
  std::cout << "# shifts = " << shpl.size() << "\n"; // 5
  // John should take the second shift */
  shpl.insert(shpl.begin() + 1,"John");
  // Peter has done too much => no second shift for him
  shpl.erase(std::find(shpl.begin()+1,shpl.end(),"Peter"));
```


STL: An example using std::vector

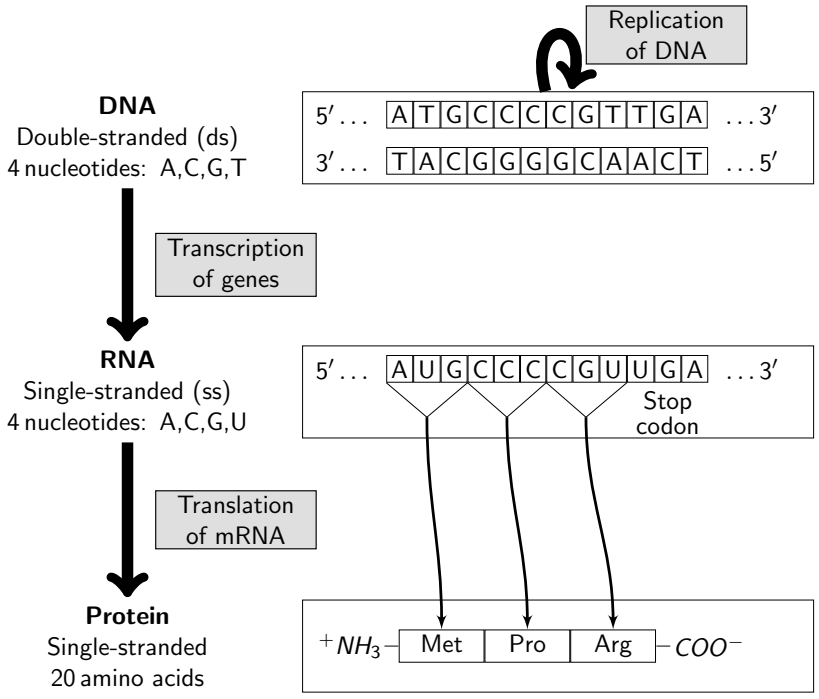
```
  for (str_vec::iterator it = shpl.begin(); it != shpl.end();
       it++)
  {
    std::cout << *it << "\n";
  }
```

Output

shifts = 0
shifts = 5
Peter
John
Paul
Mary
John

STL: An example using `std::map`: codon translation

- DNA is the carrier of genetic information
- proteins are the cellular machines which do the work
- central dogma in molecular biology:



STL: An example using `std::map`: codon translation

- each of 64 base triplets (codons) translates to an amino acid according to following table (T is used instead of U)

TTT	Phe	F	TCT	Ser	S	TAT	Tyr	Y	TGT	Cys	C
TTC	Phe	F	TCC	Ser	S	TAC	Tyr	Y	TGC	Cys	C
TTA	Leu	L	TCA	Ser	S	TAA	stop	*	TGA	stop	*
TTG	Leu	L	TCG	Ser	S	TAG	stop	*	TGG	Trp	W
CTT	Leu	L	CCT	Pro	P	CAT	His	H	CGT	Arg	R
CTC	Leu	L	CCC	Pro	P	CAC	His	H	CGC	Arg	R
CTA	Leu	L	CCA	Pro	P	CAA	Gln	Q	CGA	Arg	R
CTG	Leu	L	CCG	Pro	P	CAG	Gln	Q	CGG	Arg	R
ATT	Ile	I	ACT	Thr	T	AAT	Asn	N	AGT	Ser	S
ATC	Ile	I	ACC	Thr	T	AAC	Asn	N	AGC	Ser	S
ATA	Ile	I	ACA	Thr	T	AAA	Lys	K	AGA	Arg	R
ATG	Met	M	ACG	Thr	T	AAG	Lys	K	AGG	Arg	R
GTT	Val	V	GCT	Ala	A	GAT	Asp	D	GGT	Gly	G
GTC	Val	V	GCC	Ala	A	GAC	Asp	D	GGC	Gly	G
GTA	Val	V	GCA	Ala	A	GAA	Glu	E	GGA	Gly	G
GTG	Val	V	GCG	Ala	A	GAG	Glu	E	GGG	Gly	G

- table may vary slightly depending on organism (we use fixed table)

STL: An example using std::map: codon translation

- translation table can be implemented by a std::map which allows to store key/value pairs
- in our case: key is codon (std:string) and value is amino acid (char)

```
#include <map>
#include <string>

typedef std::map<std::string,char> str_char_map;

static char codon2aa(std::string codon)
{
  static str_char_map codonmap;

  codonmap["TCA"] = 'S';     // Serine
```

... 62 similar lines ...

```
  codonmap["GGT"] = 'G';     // Glycine
  if (codonmap.count(codon) == 1)
    return codonmap[codon];
  return -1;
}
```


STL: An example using std::map: codon translation

Goal:

- read a DNA sequence from a file, line by line
- concatenate the lines into some long string, the DNA sequence
- determine the codons in the DNA sequence
- perform the translation
- concatenate the resulting amino acids to obtain a protein sequence

```
int main (int argc,char **argv)
{
  std::ifstream fs;
  if (argc != 2)
  {
    std::cerr << "Usage: " << argv[0] << " <inputfile>\n";
    exit(EXIT_FAILURE);
  }
  fs.open (argv[1], std::fstream::in);
```

STL: An example using std::map: codon translation

```
if (fs.is_open())
{
  std::string line, dna_seq, protein;
  while (std::getline (fs,line)) // read line by line
  {
    dna_seq.append(line); // concatenate lines
  }
  fs.close();
  for (size_t idx = 0; idx <= dna_seq.size() - 3; idx += 3)
  {
    std::string codon = dna_seq.substr(idx,3);
    char aa = codon2aa(codon);
    if (aa == -1) {
      std::cerr << codon << " is not a valid codon\n";
      exit(EXIT_FAILURE);
    }
    protein.push_back(aa);
  }
  std::cout << protein << "\n";
}
```


STL: An example using std::map: codon translation

```
$ make stl-translate.x
g++ -g -Wall -Werror -O3 -c -o stl-translate.o stl-translate.cpp
g++ stl-translate.o -g -lm -o stl-translate.x
rm stl-translate.o
$ cat dnaseq.txt
AGCTTTTCATTCTGACTGCAACGGGCAATATGT
CTCTGTGTGGATTAAAAAAAGAGTGTCTGATAG
CAGCTTCTGAACTGGTTACCTGCCGTGAGTAAA
TTAAAATTTTATTGACTTAGGTCACTAAATACT
TTAACCAA
$ stl-translate.x dnaseq.txt
SFSF*LQRAICLCVD*KKSV**QLLNWLPAVSKLKFY*LRSLNTLT
```

STL: An example using `std::sort`: sorting integer vectors I

- goal: develop a program to sort integer vectors of different length:

unsorted	sorted
[1, 1, 2, 2, 3]	[1, 1, 2, 1, 2, 2, 1]
[4, 3, 1]	[1, 1, 2, 2, 3]
[1, 1, 2, 1, 2, 2, 1]	[2, 1, 4, 3, 2, 1, 1, 4, 1]
[2, 4, 3, 1, 1]	[2, 4, 3, 1, 1]
[2, 1, 4, 3, 2, 1, 1, 4, 1]	[4, 3, 1]

STL: An example using `std::sort`: sorting integer vectors II

```
#include <algorithm>
#include <stdio.h>
#include <iostream>
#include <vector>
#include <cassert>
#include <stdlib.h>

class Intarray
{
  int *array;
  unsigned long len;
  public:
  Intarray () {}; // constructor without initialization
  Intarray (const int *,unsigned long); // constr. w. init.
  ~Intarray (); // destructor
  bool lowerthan(const Intarray *param) const;
  void show () const;
};
```

STL: An example using `std::sort`: sorting integer vectors III

```
Intarray::Intarray (const int *arr, unsigned long l)
{
  this->array = new int [l];
  for (unsigned long idx = 0; idx < l; idx++)
    this->array[idx] = arr[idx];
  this->len = l;
}

Intarray::~Intarray ()
{
  delete[] this->array;
}

bool Intarray::lowerthan(const Intarray *param) const
{ // last const => this function does not change class vars
  const int *arr_a = this->array;
  unsigned long len_a = this->len;
  const int *arr_b = param->array;
  unsigned long len_b = param->len;
```


STL: An example using `std::sort`: sorting integer vectors IV

```
  for (unsigned long idx = 0; /* Nothing */; idx++)
  {
    if (idx < len_a)
    {
      if (idx < len_b)
      {
        if (arr_a[idx] != arr_b[idx])
          return arr_a[idx] < arr_b[idx] ? true : false;
      } else
      {
        return false; // arr_b is proper prefix of arr_a
      }
    } else
    {
      return true; // arr_a is proper prefix of arr_b
    }
  }
  return false; // arr_a == arr_b
}
```

STL: An example using std::sort: sorting integer vectors V

```
void Intarray::show () const
{
  std::cout << "vector = [";
  for (unsigned long v = 0; v < this->len; v++)
  {
    std::cout << this->array[v]
              << (v < this->len - 1 ? "," : "]\n");
  }
}

bool intarray_lowerthan(const Intarray *a,const Intarray *b)
{
  return a->lowerthan(b);
}

typedef std::vector<Intarray *> IA_tab;

int main (int argc,char *argv[])
{
  const unsigned int minlen = 3U;
```


STL: An example using std::sort: sorting integer vectors VI

```
  const unsigned int maxlen = 10U;
  const int minvalue = 1;
  const int maxvalue = 4;
  if (argc != 2)
  {
    std::cerr << "Usage: " << argv[0] << " num of vectors\n";
    exit(EXIT_FAILURE);
  }
  int numofvectors;
  if (sscanf(argv[1],"%d",&numofvectors) != 1 ||
      numofvectors <= 1)
  {
    std::cerr << "Usage: " << argv[0] <<
                 " num of vectors must be >= 2\n";
    exit(EXIT_FAILURE);
  }
  srand48(366292341);
  IA_tab ia_tab;
  int *arr = new int [maxlen+1];
```

STL: An example using `std::sort`: sorting integer vectors VII

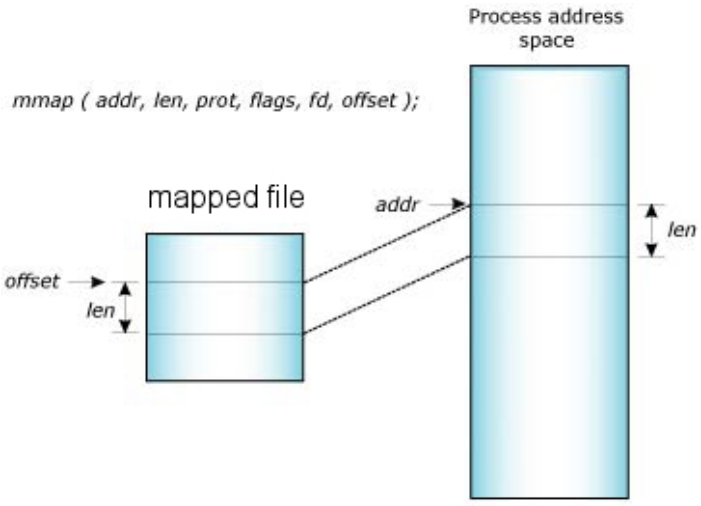
```
  for (int idx = 0; idx < numofvectors; idx++)
  {
    unsigned int len = minlen + drand48() *
                       (maxlen - minlen + 1);
    assert(minlen <= len && len <= maxlen);
    for (unsigned long v = 0; v < len; v++)
    {
      arr[v] = minvalue +
               drand48() * (maxvalue - minvalue + 1);
      assert(minvalue <= arr[v] && arr[v] <= maxvalue);
    }
    Intarray *vec = new Intarray (arr,len);
    ia_tab.push_back(vec);
  }
  delete[] arr;
  std::sort(ia_tab.begin(),ia_tab.end(),intarray_lowerthan);
```


STL: An example using `std::sort`: sorting integer vectors VIII

```
  Intarray *previous = NULL;
  for (IA_tab::iterator it = ia_tab.begin();
       it != ia_tab.end(); it++)
  {
    if (previous != NULL && !previous->lowerthan(*it))
    {
      previous->show();
      std::cerr << ">=";
      (*it)->show();
      exit(EXIT_FAILURE);
    }
    previous = *it;
    (*it)->show();
  }
  for (IA_tab::iterator it = ia_tab.begin();
       it != ia_tab.end(); it++)
    delete *it;
  return 0;
}
```

Example: Memory mapped files

- task: implement class for accessing contents of file as array of bytes
- appropriate method: memory mapping (via function `mmap`)
- this reserves a memory area of the size of the accessed files (or parts thereof)
- delivered address can be used just like a normal pointer to a `malloc`ed memory area
- only the accessed parts of the mapped file are effectively copied into RAM



`addr`	address to use or `NULL`
`len`	size of area to map
`prot`	mode to access
`flags`	shared or private mapping
`fd`	file descriptor of opened file
`off`	offset (multiple of page size)

mmap.hpp

```
class Mappedfile {
  private:
    const char *filename;
    int filedesc;
    unsigned long filesize,
                  numberoflines,
                  numberofwords;
    bool hasfilevalues;
    void *memorymap;
    void openfile(const char *filename);
    void creatememorymap(void);
    void filevalues_get(void);
  public:
    Mappedfile(const char *filename); /* contructor */
    ~Mappedfile(void);                /* destructor */
    const char *filename_get(void) const;    /* accessor */
    unsigned long filesize_get(void) const; /* accessor */
    unsigned long numberoflines_get(void);  /* accessor */
    unsigned long numberofwords_get(void);  /* accessor */
    const void *memorymap_get(void) const;  /* accessor */
};
```

mmap.cpp: required include statements

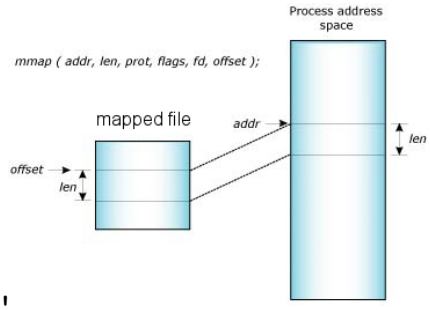
```
#include <iostream>      // for std::cerr
#include <stdlib.h>      // for EXIT_FAILURE, exit()
#include <ctype.h>       // for iswspace()
#include <sys/types.h>   // for fstat() and open()
#include <sys/stat.h>    // for fstat() and open()
#include <unistd.h>      // for fstat()
#include <fcntl.h>       // for open()
#include <sys/mman.h>    // for mmap()
#include "mmap.hpp"      // for class Mappedfile declaration
```


mmap.cpp: openfile

```
void Mappedfile::openfile(const char *filename)
{
  struct stat buf;

  this->filedesc = open(filename,O_RDONLY);
  if (this->filedesc == -1)        // check for error code
  {
    std::cerr << "openfile␣failed:␣" << filename << '\n';
    exit(EXIT_FAILURE);
  }
  if (fstat(this->filedesc,&buf) == -1) // get status of file
  {
    std::cerr << "cannot␣access␣status␣of␣file␣"
              << filename << '\n';
    exit(EXIT_FAILURE);
  }
  this->filesize = buf.st_size;  // store size of file
  this->filename = filename; // store filename
}
```

mmap.cpp: creatememorymap

```
void Mappedfile::creatememorymap(void)
{
  this->memorymap = mmap(NULL,
                         (size_t) this->filesize,
                         PROT_READ,
                         MAP_SHARED,
                         this->filedesc,
                         (off_t) 0);
  if (this->memorymap ==
      (void *) MAP_FAILED)
  {
    std::cerr << "creation of memorymap('
              << this->filename << ") failed\n";
    exit(EXIT_FAILURE);
  }
}
```



mmap.cpp: filevalues_get

```
void Mappedfile::filevalues_get(void)
{
  const unsigned char *sptr, *send;
  bool lastwasspace = false;
  this->numberoflines = this->numberofwords = 0;
  for (sptr = (const unsigned char *) this->memorymap,
       send = sptr + filesize_get();
       sptr < send; sptr++)
  {
    if (*sptr == '\n')
      this->numberoflines++;
    if (iswspace(*sptr))
    {
      if (!lastwasspace)
      {
        this->numberofwords++;
        lastwasspace = true;
      }
    } else lastwasspace = false;
  }
}
```

mmap.cpp: Constructor and Destructor

```
Mappedfile::Mappedfile(const char *filename)
{
  this->hasfilevalues = false;
  openfile(filename);
  creatememorymap();
}

Mappedfile::~Mappedfile(void)
{
  if (munmap(this->memorymap,(size_t) this->filesize) != 0)
  {
    std::cerr << "deletion␣of␣memorymap("
              << this->filename << ")␣failed\n";
    exit(EXIT_FAILURE);
  }
  close(this->filedesc);
}
```


mmap.cpp: Accessors

```
const char *Mappedfile::filename_get(void) const
{
  return this->filename;
}

unsigned long Mappedfile::filesize_get(void) const
{
  return this->filesize;
}

const void *Mappedfile::memorymap_get(void) const
{
  return this->memorymap;
}
```

mmap.cpp: More accessors

```
unsigned long Mappedfile::numberoflines_get(void)
{
  if (!this->hasfilevalues)
  {
    filevalues_get();
    this->hasfilevalues = true;
  }
  return this->numberoflines;
}

unsigned long Mappedfile::numberofwords_get(void)
{
  if (!this->hasfilevalues)
  {
    filevalues_get();
    this->hasfilevalues = true;
  }
  return this->numberofwords;
}
```


mmap-main.cpp: The main function

```
#include "mmap.hpp"

int main(int argc,char **argv)
{
  if (argc != 2)
  {
    std::cerr << "Usage: " << argv[0] << " <filename>\n";
    return EXIT_FAILURE;
  }
  Mappedfile *mpf = new Mappedfile(argv[1]);
  std::cout << mpf->filename_get() << "\n";
  std::cout << "characters=" << mpf->filesize_get() << '\n';
  std::cout << "lines=" << mpf->numberoflines_get() << '\n';
  std::cout << "words=" << mpf->numberofwords_get() << '\n';
  delete mpf;
  return EXIT_SUCCESS;
}
```