Welcome to LABC

```
hello class
```

```c
#include <stdio.h>
int main()
{
      printf ("hello class\n");
      return 0;
}
```

Today’s Lecture

- Administration
- Basics
- The preprocessor

Course Staff

Teacher: Eran Marom

TA: Eva Kimel

Course exercises coordinator: Ohad Cohen

Course's email: labc@cs

Course's site: www.cs.huji.ac.il/~labc

Don't send emails to our personal mailboxes!

It will just get lost.

Time Schedule

6 classes + TAs of C
+ 1 summary and Q&A before the exam
+ 3 **time consuming** exercises + tons of FUN ☺

Lessons:
Tue: 12:00-14:45
Thu: 15:00-15:45

Tirgul (3 groups):
Wed: 11:00, 18:00
Thu: 14:00

Additional HELP:

Reception hour:
Thursday 16:00,
Rothberg B404

Eva, Ohad
– see course site

Forums

Administration

Read and follow course guidelines at course web site:

- Course guidelines
  - Submission policy
  - Communication
  - Grading policy
  - etc.
- Coding guidelines

Course Grading

1. 3-4 theoretical exercises
2. 3 programming exercises
3. 1 exam

Final grade:

- 50% exercises and 50% exam
- You must have a pass grade both in the exercises average, and in the exam
- You must have a grade >= 20 in each exercise

Don’t wait for the last minute

Plagiarism policy

- You may discuss your exercise with friends
- You must not exchange any written material (or code) between students
- You are not allowed to copy code from external resources.

It is very easy to detect copying...

How 2 get a 100 for the hw? ☺

- **Be honest!**
- Organize your time = start as early as possible
- Read very carefully the instructions
- Ask in the forum if something is unclear
- Pass the pre-submission script!
- Follow the coding standards guidelines
- Check each part of your code, including edge cases
- We will **not** publish all of our tests prior to submission
- However, you are welcome to share your testers with other students!
- Learn & explore by yourself!

Course Objectives

Learn the unique features of C:

1. C as a procedural programming language
2. **Pointers** [even to functions and to pointers!]
3. Memory management

Practice of programming:

- Style
- Modularity
- Testing & Debugging
- Efficiency & Portability

Books & Other resource

Books

1. “The C Programming Language” , 2nd Edition, Brian W. Kernighan & Dennis M.Ritchie
2. “C in a Nutshell”, 1st edition, Peter Prinz & Tony Crawford, O'reilly 2005.
3. “The Practice of Programming”, Brian W. Kernighan & Rob Pike
4. “Programming Pearls” 2nd Edition, Jon Bentley

Web

1. Wikipedia the article C_(programming_language) to begin with is a nice reading.
2. MSDN
3. http://www.cplusplus.com/, http://en.cppreference.com/w/, and http://stackoverflow.com/

Working environments

Your exercises must compile and run on the school machines (aquarium)

Writing C code:

- IDE (Integrated development environment): emacs, eclipse, code::blocks, qtcreator etc.
- Any text editor + SHELL commands

Working from home:

- http://wiki.cs.huji.ac.il/wiki/Connecting_from_outside
- SSH to CS's "river" server (**Only river**)
- Install Linux on your machine (don't be afraid – windows will remain) or on Virtual Machine (Virtual Box) or use cygwin.
- Visual Studio (You can get professional edition from the "aguda" for free, or use the free express edition or https://www.dreamspark.com/)
- CodeBlocks with some free compiler – http://www.codeblocks.org
- If you wish to practice code snippets, this is a good web site to do so: http://compileonline.com/

History

C	C++	Java
70's – Development of UNIX. (Richie+Kernigham – Bell labs)	80's – Large efficient code. (Stroustrup – Bell labs)	90's – Language for the web. (Sun Microsystems)

Simple to convert to machine code.	⟷	**VM**
Fast and Efficient	⟷	**Secure and Safe**
Welcome to labC	⟷	**Easy to avoid bugs Easy to Debug**

History – C:

- First "standard"
- Default int

- enums
- return struct
- fun. prototypes
- void

- //
- VLA
- variadic macros
- inline

- multithreading
- Anonymous structs
- _Generic

K&R 1st ed. ('78) → ANSI C C89 C90 ('89) → C99 ('99) → C11 ('11)

This course

C – Design Decisions

"Bare bones" – the language leaves maximal flexibility to the programmer

- Efficient code (operating systems)
- More control on memory & CPU usage

High-level

- Type checking
- High-level constructs

Portable

- Standard language definition
- Standard library

C – beware:

No run-time checks

- Array boundary overruns
- Illegal pointers

No memory management

- Programmer has to manage memory
  - Get memory from OS
  - Return it when done using it

C++ - OO extension of C

Classes & methods

- OO design of classes

Generic programming

- Template allow for code reuse

Stricter type system

Some run-time checks & memory control

First Program in C

```c
// This line is a comment,
/* and those lines also.
Next line includes standard I/O library*/
#include <stdio.h>

// main – program entry point. Execution starts here
int main()
{ // {…} define a block
   printf("Hello class!\n");
   return 0;
}
```

Compiling & Running…

```
> gcc –Wextra hello.c –o hello
> hello
Hello class!
>
```

The basic syntax

Coding guidelines on website.

Variables

Statically typed – each variable has a type. Declare by: <type> <name>.

```
int x;
int x,y;
```

```
int x=0;
```

Optionally initialize (otherwise, for local variables, it is **undefined!**)

Variables

Statically typed – each variable has a type. Declare by: <type> <name>.

```
int x;
int x,y;


int x=0;
```

Important word in C. anything may happen. Probably won't format your disk, but may give a hacker a way to do it.

Optionally initialize (otherwise, for local variables, it is **undefined!**)

Variables

Where to declare?

1. Inside a block (C89 - block beginning), will be visible only in block.
2. Outside all blocks – global - will be visible everywhere.

```
int x=0; // global
int main()
{
   int x=1; //local hides global
   {
      int x=2; //local hides outer scope
      //x is 2
   }
   //x is 1 again!
}
```

Scopes

- **Code block** defined with **“{”** and **“}”.**
- Only declarations inside current or outer scopes are visible.
- Declarations in inner scope **hide** declarations in outer scopes:
- Outmost scope (global) had no brackets.
- Keep in mind that a function is also a scope.

```
int y=5,x=0;
{
   int x=y;
   //x is 5
   {
      int y;
   }
}
// x is 0
```

Statements - conditional

```
if (expression)
//single statement or block
else if (expression)
//single statement or block
else (expression)

//single statement or block
switch (integer value)
later...
```

Statements - loops

The usual suspects:

```
int i,j; //in ANSI C you cannot declare inside the for!

// for( initial ; test condition ; update step )
for (i=0,j=0; (i<10 && j<5); i++,j+=2)
//statement or block

while (condition)
//statement or block

do
//statement or block
while (condition);
```

Second Program

```
#include <stdio.h>
int main()
{
   int i;      // declares i as an integer
   int j = 0; // declares j as an integer,
              // and initializes it to 0

   for( i = 0; i < 10; i++ )
   {
      j += i; // shorthand for j = j + i
      printf("%d %d %d\n", i, j, (i*(i+1))/2);
   }
   return 0;
}
```

Running…

```
> gcc –Wextra loop.c –o loop
> loop
```

```
0 0 0
1 1 1
2 3 3
3 6 6
4 10 10
5 15 15
6 21 21
7 28 28
8 36 36
9 45 45
```

Character Input/Output

```
#include <stdio.h>
int main()
{
    int c;
    while( (c = getchar()) != EOF )
    {
        putchar(c);
    }
    return 0;
}
```

#define macro

```
#include <stdio.h>
#define NUM_OF_LINES 10
int main()
{
   int n = 0;
   int c;
   while(((c=getchar())!=EOF) && (n< NUM_OF_LINES) )
   {
      putchar(c);
      if( c == '\n' )
          n++;
   }
   return 0;
}
```

General Input/Output

&&&

```
#include <stdio.h>
int main()
{
   int n;
   float q;
   double w;
   printf("Please enter an int, a float and a double\n");
   scanf("%d %f %lf",&n,&q,&w);
   printf("ok, I got: n=%d, q=%f, w=%lf",n,q,w);
   return 0;
}
```

Functions

C allows to define functions

Syntax:

Return type

Parameter declaration

```
int power( int a, int b )
{
…
  return 7;
}
```

Return statement

Procedures

Functions that return **`void`**

```
void power( int a, int b )
{
…
  return;
}
```

Return w/o value (optional)

Example – printing powers

```c
#include <stdio.h>

int power( int base, int n )
{
   int i, p;

   p = 1;
   for( i = 0; i < n; i++ )
   {
      p = p * base;
   }
   return p;
}
```

```c
int main()
{
   int i;

   for( i = 0; i < 10; i++ )
   {
      printf("%d %d %d\n",
         i,
         power(2,i),
         power(-3,i) );
   }
   return 0;
}
```

Functions Declaration

```
void funcA()
{
    ...
}
void funcB()
{
    funcA();
}
void funcC()
{
    funcB();
    funcA();
    funcB();
}
```

```
void funcA()
{
    ...
}
void funcB()
{
    funcC();
}
void funcC()
{
    funcB();
}
```

"Rule 1": A function "knows" only functions which were declared above it.

Error:
funcC is not known yet.

Functions Declaration

Amendment to “Rule 1” : Use forward declarations.

```
void funcC(int param);
void funcA()
{

}
void funcB()
{
   funcC(7);
}
void funcC(int param)
{
}
```

Function declaration

- A declaration tells the compiler the function **name and return type.**

```
//the following 3 declarations are legit:
  int foo(int a);
  int foo(int);
  int foo();
  int main()
  {
      foo(5);
      return 0;
  }
  int foo(int a) {return a;}
```

NO function overloading

- A function may have several declarations, but only one definition.
- The following code will not compile:

```
int foo(int a) {return a;}
int foo(int a, int b) {return a*b;}

int main()
{
  foo(5);
  return 0;
}
```

Primitive types & sizeof operator

```
int main()
{
  //Basic primitive types
  printf("sizeof(char)   = %lu\n",sizeof(char));
  printf("sizeof(int)    = %lu\n",sizeof(int));
  printf("sizeof(float)  = %lu\n",sizeof(float));
  printf("sizeof(double) = %lu\n",sizeof(double));

  //Other types:
  printf("sizeof(void*)  = %lu\n",sizeof(void*));
  printf("sizeof(long double)= %lu\n",sizeof(long double));
  return 0;
}
```

Boolean types

Boolean type **doesn't exist in C**!

Use *char/int instead* *(there is also a possibility to work on bits)*

zero = false

non-zero = true

Examples:

```
while (1)
{
}
```

```
if (-1974)
{
}
```

```
i = (3==4);
```

Boolean types

Boolean type **doesn't exist in C**!

Use *char/int instead (there is also a possibility to work on bits)*

zero = false

non-zero = true

Examples:

```
#define TRUE
1
while (TRUE)
{
}

(infinite loop)
```

```
while (1)
{
}

(infinite loop)
```

```
if (-1974)
{
}

(true statement)
```

```
i = (3==4);

(i equals zero)
```

Boolean variables – example

```
int main()
{
    int a = 5;
    while(1)
    {
        if(!(a-3))
        {
            printf("** a=3 **\n");
            break;
        }
        printf("a=%d\n",a--);
    }
    return 0;
}
```

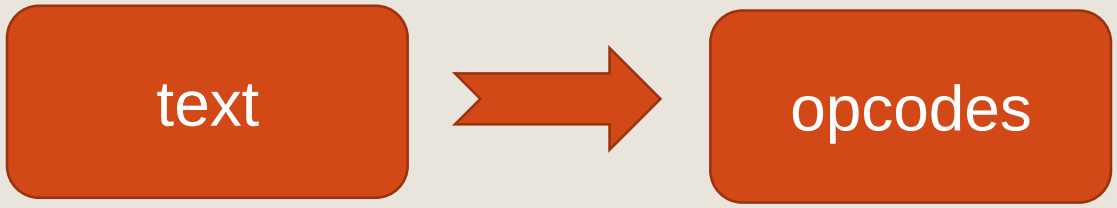
Why has the same TRUE conditions as in (a==3)?

Booleans in C99

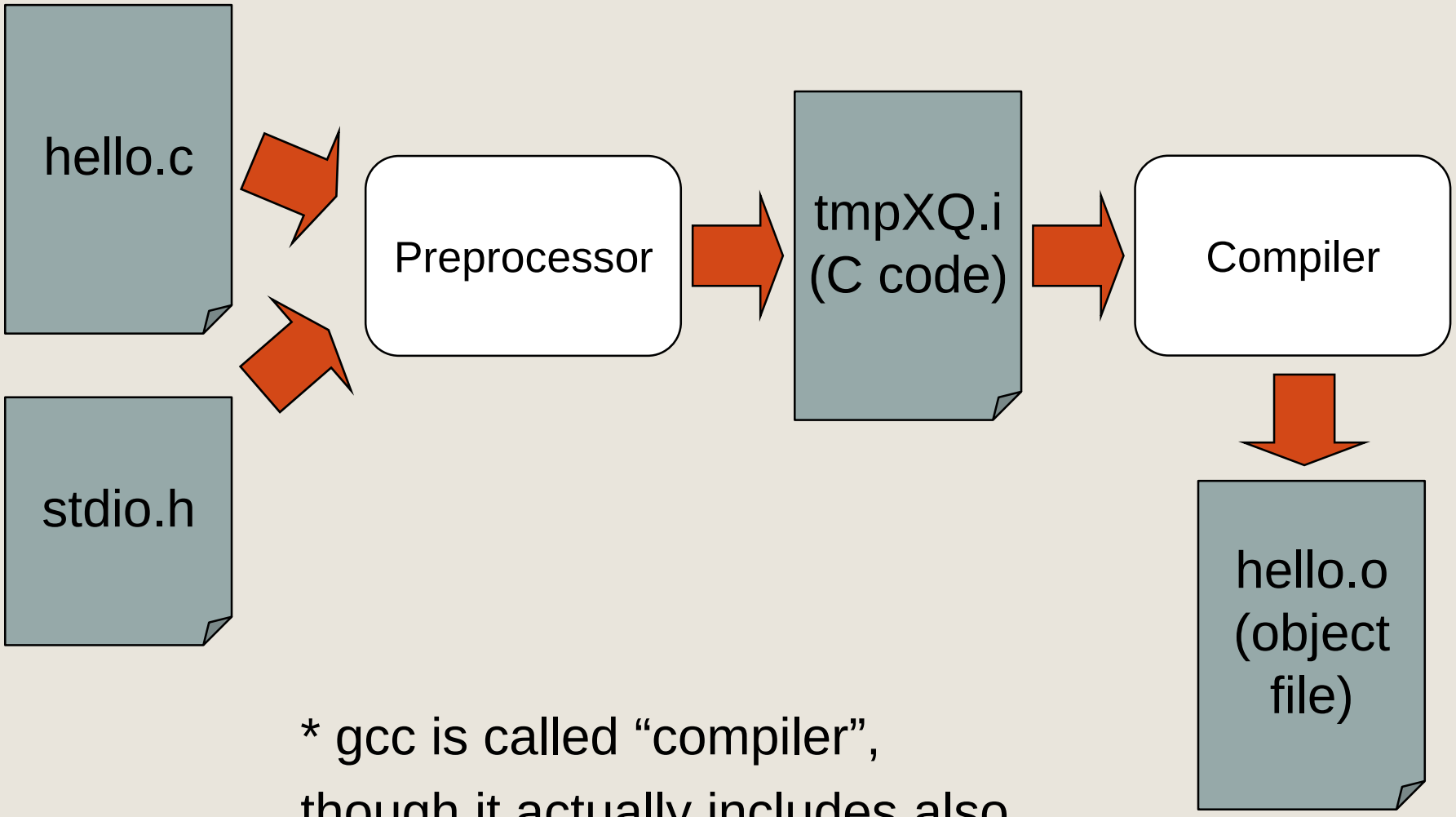
C99 added the _Bool type. You can use it as follows:

```
#include <stdbool.h>
int main()
{
   bool t = true;
   bool f = false;
   if (t != f)
   {
      printf(“t=%d, f=%d\n”, t, f);
      printf(“It is %s that 3 is greater than 4.\n”,
              (3>4) ? ”true” : ”false”);
   }
   return 0;
}
```

COMPILATION AND LINKAGE



Compilation in C



* gcc is called “compiler”, though it actually includes also preprocessor and linker

The Preprocessor

Preprocessor

A single-pass program that:

1. Includes header files
2. Expands macros
3. Controls conditional compilation
4. Removes comments

Outputs – a code ready for the compiler to work on.

Preprocessor

We can test what the preprocessor does:

```
> gcc -E hello.c
```

will print the C code after running the preprocessing

#include directive

```
#include  "foo.h"
```

Includes the file ‘foo.h’, start searching from the same directory as the file that contains the #include

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

Includes the file ‘stdio.h’, start searching from the **standard library directory** (part of gcc installation)

Header files

Header file contains

1. Definitions of data types (`typedef`, `structs`)
2. Declarations of functions & constants that are shared by multiple modules.

`#include` directive allows several modules to share the same set of definitions/declarations

Modules & Header files

square.h

```
int area     (int x1,int y1,int x2,int y2);
...
```

MyProg.c

```
#include "square.h"
int main()
{
   area (2,3,5,6);
}
```

square.c

```
#include "square.h"
#include <math.h>
// implementation
int area (int x1,int y1,int x2, int y2)
{
...
}
```

#define directive

```
#define FOO 1
```

```
int x = FOO;
```

is equivalent (after the preprocessing) to:

```
int x = 1;
```

#define with arguments - MACRO

```
#define SQUARE(x) x*x
b = SQUARE (a);
```

is the same as

```
b = a*a;
```

#define -- cautions

```
#define SQUARE(x) x*x
#define PLUS(x) x+x
b = SQUARE(a+1);
c = PLUS(a,a)*5;
```

Is it what we intended?

```
b = a+1*a+1; //Actually: b = 2*a+1;
c = a+a*5; //Actually: c = 6*a;
```

Solution:

```
#define SQUARE(x) (x)*(x)
#define PLUS(x) (x + x)
```

#define

Multi-line:
All preprocessor directive effect one **line** (not c statement).
To insert a line-break, use “\”:

BAD:

```
#define x (5 +
           5)
// x == 10 !
```

GOOD:

```
#define x (5 + \
           5)
// x == 10 !
```

Alternative to macros:

- **Constants**

```
enum { FOO = 1 };
```

or

```
const int FOO = 1;
```

- **Functions** – inline functions (C99, C++, later on)

#if directive

Allows to have conditional compilation

```
#if defined(DEBUG)
   // compiled only when DEBUG exists
   printf("X = %d\n", X);
#endif
```

Debugging - assert

assert.h

```
#include <assert.h>
// Sqrt(x) - compute square root of x
// Assumption: x  non-negative
double sqrt(double x )
{
   assert( x >= 0 );  // aborts if x < 0
…
```

If the program violates the condition, then

```
assertion "x >= 0" failed: file "Sqrt.c", line 7 <exception>
```

The exception allows to catch the event in debug mode

assert.h

- Important coding practice
- Declare implicit assumptions
- Sanity checks in code
- Check for violations during debugging/testing
- #, ##

assert.h

```
// procedure that prints error message
// to disable the printing define the macro NDEBUG

void __assert(char* file,int line,char* test);

#ifdef NDEBUG
   #define assert(e)     ((void)0)
#else
   #define assert(e)     \
      ((e) ? ((void)0) :   \
      __assert(__FILE__, __LINE__, #e))
#endif
```

Debug/Test mode vs Release mode

```
#include <assert.h>
#define MAX_INTS 100


int main()
{
        int ints[MAX_INTS];
        i = foo(<something complicated>);
        // i should be in bounds, but is it really?
        // safety assertions
        assert(i>=0);
        assert(i<MAX_INTS);
        ints[i] = 0;
```

Debug/Test mode vs Release mode

```
#define NDEBUG
#include <assert.h>
#define MAX_INTS 100

int main()
{
        int ints[MAX_INTS];
        // should be in bounds, but is it really?
        i = foo(<something complicated>);
        // safety assertions
        assert(i>=0);
        assert(i<MAX_INTS);
        ints[i] = 0;
        ...
```

General purpose debug function

```
// file: my_debug.h
// defines a general purpose debug printing function
// usage: same as printf,
// to disable the printing define the macro NDEBUG


#ifdef NDEBUG
   #define printDEBUG(format, ...) ((void)0)
#else
   #define printDEBUG(format, ...)  \
       printf(format, ##__VA_ARGS__)
#endif
```

Defining NDEBUG using the compiler

```
>> gcc my_program.c -DNDEBUG -o my_exe
```

This is equivalent for adding at the beginning of the file the definition:

```
#define NDEBUG
```

Preprocessor – summary

- **Text** processing program.
- Does not know c rules
- Operates before compilation, output passed to compiler

- Can do copy and paste / cut
  - `#include`

    - pastes the included file to current file (.h by convention)
    - usually contains forward declarations
  - `#define`

    - copy-pastes the macro body where macro name appears
    - constants, simple "functions"
  - `#if`

    - If condition is not fulfilled, cut the code
    - conditional compilation (e.g. debugging code)