

Basics of C

Lecture 1

Summer 2018
Innopolis University
Eugene Zouev

Introductory Remarks

Organizational Aspects

Who Is This Guy? 😊

- Eugene Zouev
- Have been working at Moscow Univ., Swiss Fed Inst of Technology (ETH Zürich), EPFL (Lausanne); PhD (1999, Moscow Univ).
- Prof. interests: compiler construction, language design, PL semantics.
- The author of the 1st Russian C++ compiler (Interstron Ltd., 1999-2000).
- Zonnon language implementation for .NET & Visual Studio (ETHZ, 2005).
- Swift prototype compiler for Tizen & Android (Samsung R&D Center, 2015)
- «Редкая профессия», ДМК Пресс, Москва 2014.

Why the Course?

- C is the most popular language for developing system software
 - See monthly TIOBE index of popularity.
- C – its semantics (and syntax) – lies in the foundation of the current software world.
 - Most of modern languages (re)use C syntax 😊.
- Each software that requires extreme efficiency with strong memory limitations is typically written in C.
 - Examples are Windows, Linux, Android, Tizen.
- The course **Operating Systems** in the forthcoming fall semester is completely based on C – and its prerequisite assumes you know the basics of C.

C & C++



The Structure of the Course

- | | | |
|------|----------|---|
| 1 | May, 24 | 10:40-12:10 Lecture, rm 321
12:20-13:50 Lab., rms 321, 101 |
| 2 | May, 28 | 10:40-12:10 Lecture, rm 321
12:20-13:50 Lab., rms 321, 101 |
| 3 | June, 1 | 10:40-12:10 Lecture, rm 321
12:20-13:50 Lab., rms 321, 101 |
| 4 | June, 5 | 10:40-12:10 Lecture, rm 321
12:20-13:50 Lab., rms 321, 101 |
| 5 | June, 6 | 10:40-12:10 Lecture, rm 321
12:20-13:50 Lab., rms 321, 101 |
| Exam | June, 13 | 10:40-13:40, rm 321 |

The Structure of the Course

- Each lecture will end with one/two/more assignments.
 - Solutions should be send to TAs (details will be given later) until the next lecture starts.
 - Solutions matter on the final grade
 - The grading formula will be given later
- Labs: 2 acad. hours after each lecture.
 - Discussion about the lecture contents/problems.
 - Solutions of the assignments given on the prev lecture.
 - Additional program examples.
 - Student presentations (?)
- Final examination
 - Written form
 - With no help of electronic devices
 - Lecture slides are allowed
 - Three tasks (typically to write programs in C)

The C Programming Language

C Language Authors

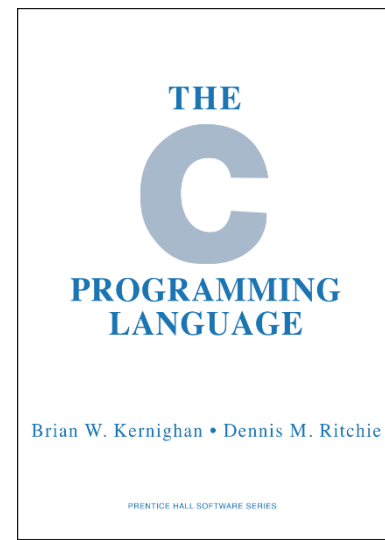


Brian Kernighan



Dennis Ritchie

References



“K&R C”

- C International Standard
ISO/IEC 9899:2011

The latest publicly-available document (n1570):

<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>

- Working group JTC1/SC22/WG14 - C
- C99 Rationale:
<http://www.open-std.org/jtc1/sc22/wg14/www/docs/C99RationaleV5.10.pdf>
- Kernighan, Brian W.; Ritchie, Dennis M. (February 1978). The C Programming Language (1st ed.). Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110163-3.
- Any modern book in C ☺.
- Online resources (many of them...)

The C Programming Language

- C is very simple & compact language. (Oh, really? 😊)
 - However, C programs can be extremely complicated and might look cryptic.
- C is complete & very powerful language.
- C is "middle-level" language.
 - No constructs with complicated semantics; no built-in system support like memory management.
- C was designed to be as close to hardware as possible.
 - Each C language construct is typically mapped to a clear machine code (or even to a single machine instruction).
- The C core language is completely independent from its standard library (see next slides).
- The C language is old.
 - It doesn't support modern programming patterns & idioms.
 - Its programming paradigm is conservative & archaic.

The C Programming Language

- C is typed language (but not strongly typed).
 - Each C object is characterized by its type;
 - No way to change object's type during program execution;
 - There are a lot of ways, however, to **convert** types.
- C assumes compilation.
 - C programs should be **compiled** into a sequence of machine instructions before running;
 - Typically, C program should also be **linked** with some other programs (libraries) before running.

The Source & Machine Code Example

Software: from C to processor instructions

C:

```
int f (int a, int b)
{
    int s = 0;

    while (s < a)
        s += b;

    return s;
}
```

Assembly:

```
                .set noreorder
sum:
    blez        $4, exit
    move        $2, $0

    addu        $2, $2, $5

loop:
    slt         $3, $2, $4
    bnel        $3, $0, loop
    addu        $2, $2, $5

exit:
    jr          $31
    nop
```

Machine
code

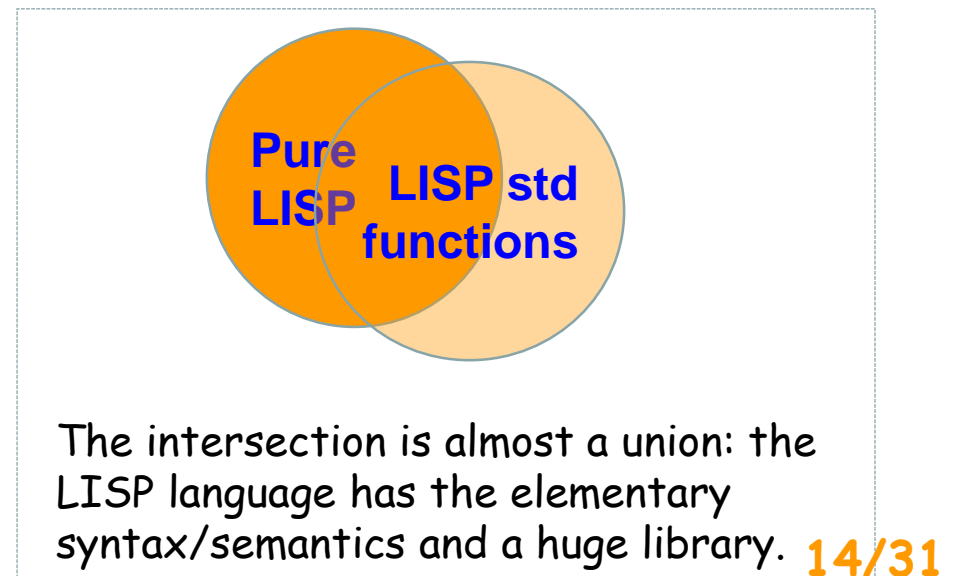
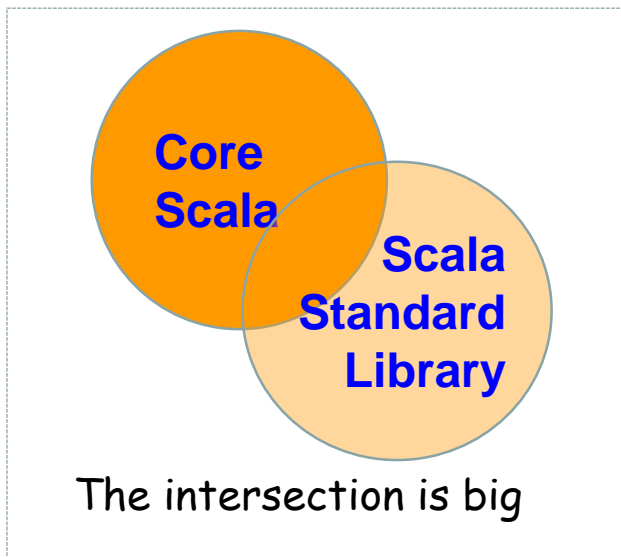
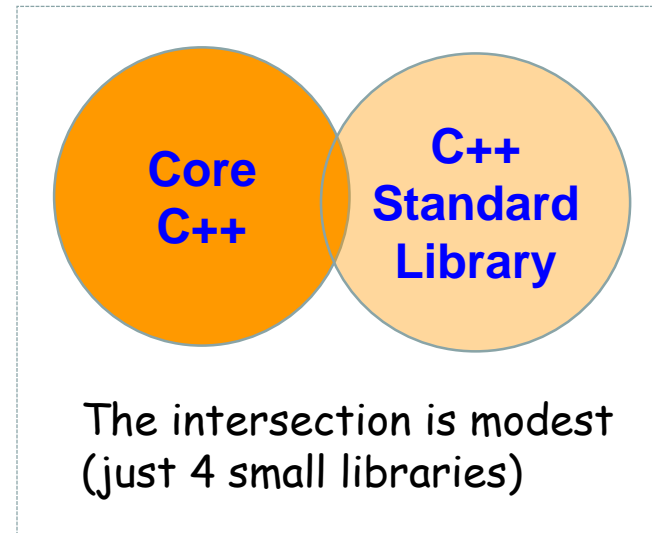
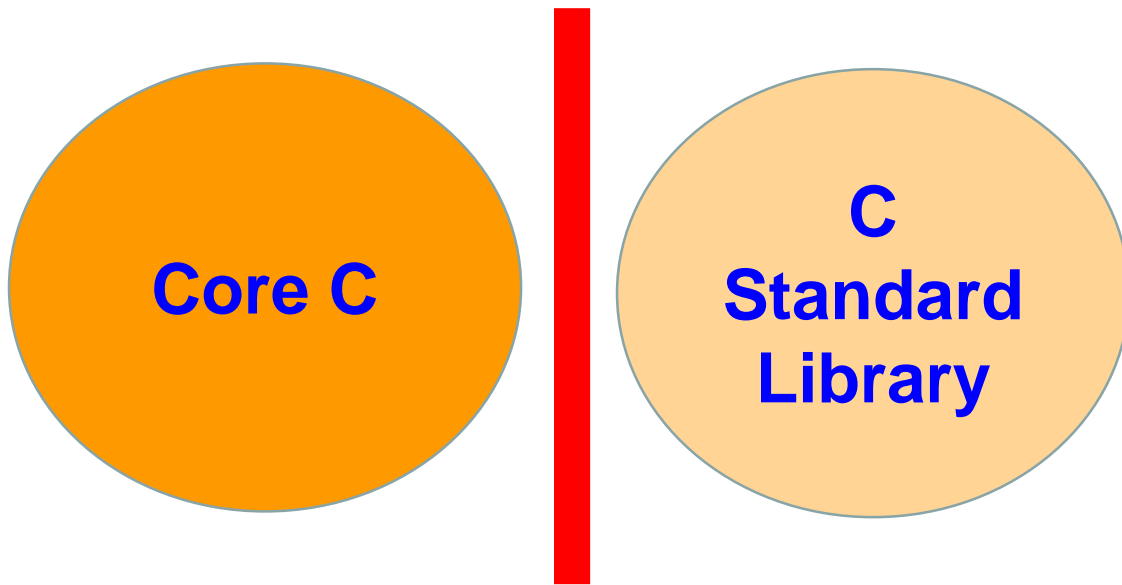
```
18800005
00001025

00451021

0044182a
5460fffe
00451021

03e00008
00000000
```

Core Language & Language Library



Languages: Syntax vs Semantics

“Usual” view at a language:



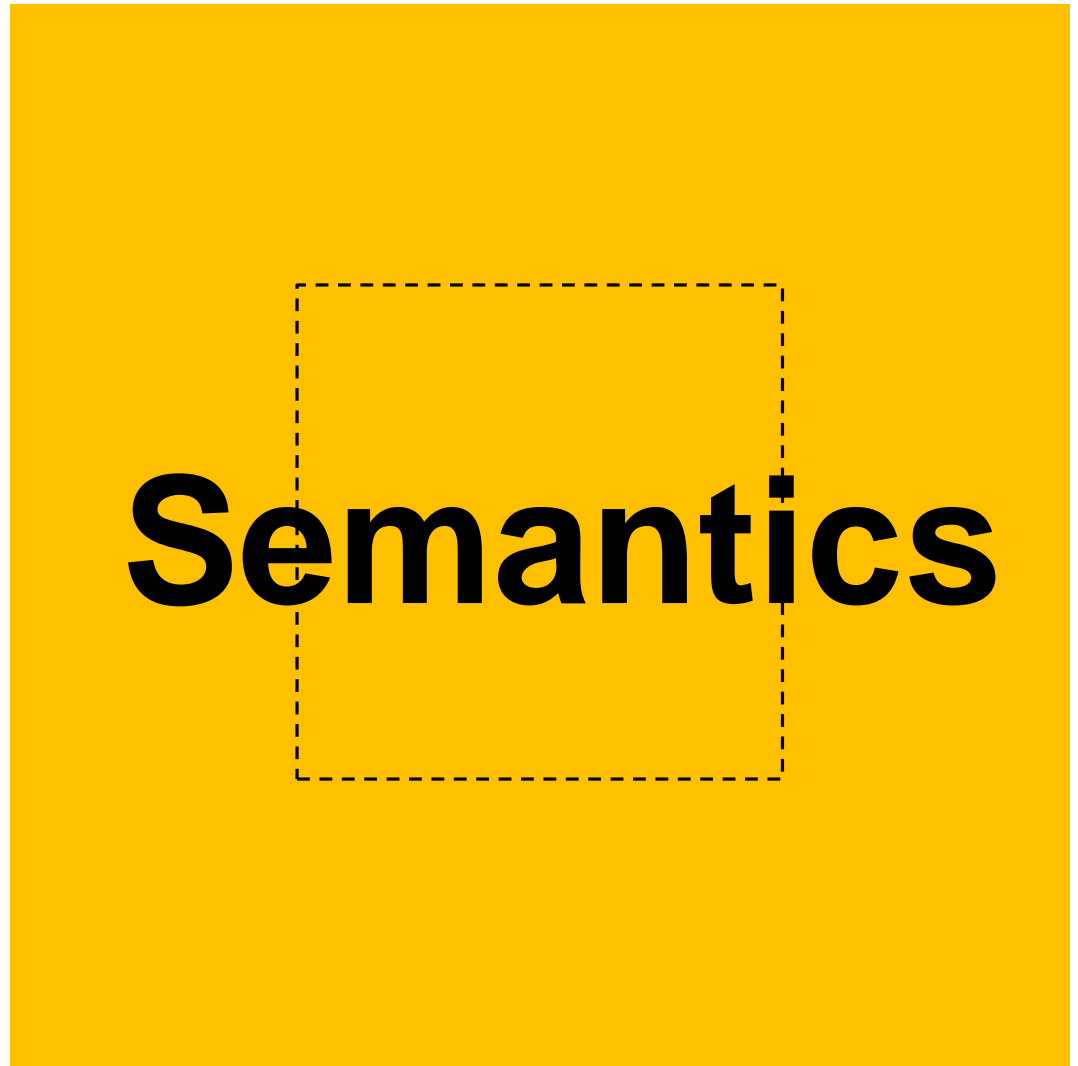
Syntax



Semantics

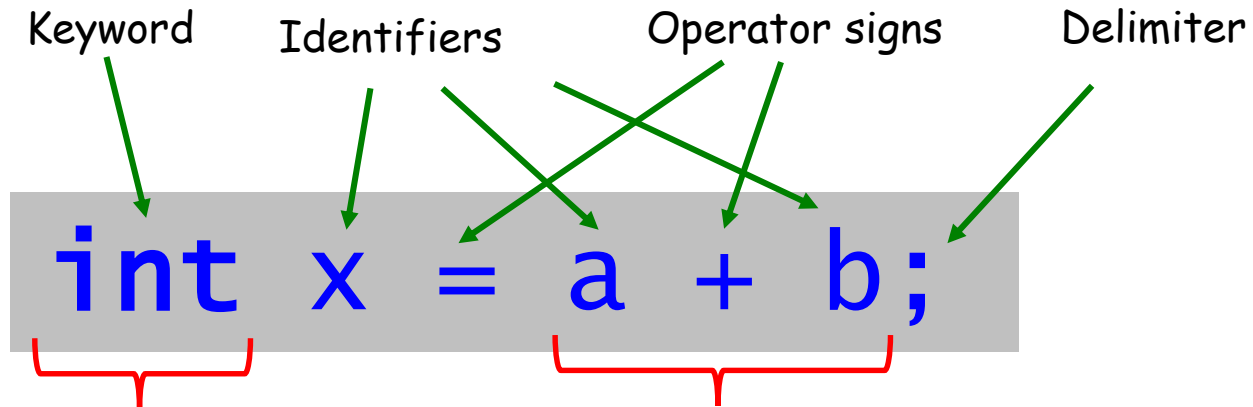
Languages: Syntax vs Semantics

Reality:

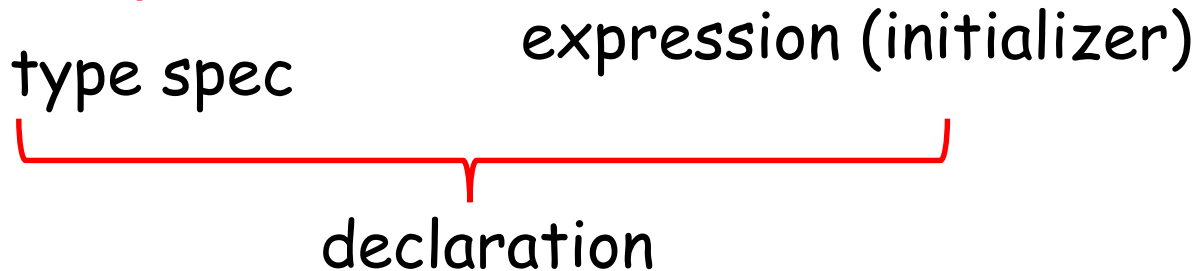


C: Syntax vs Semantics

Lexics



Syntax



Semantics

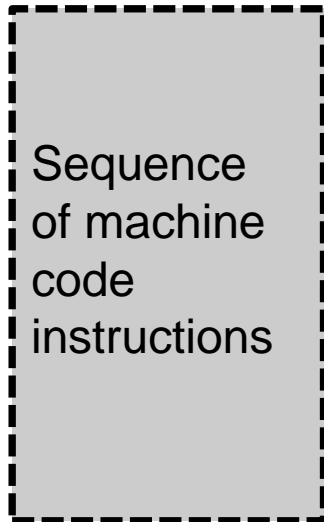
- Allocate memory for the new integer variable (in stack)
- Calculate expression from initializer
- Perform type conversion(s) to integer, if necessary
- Store the value of the expression
- Make x available in the current context

The C Memory Model

Each C program uses three kinds of memory:

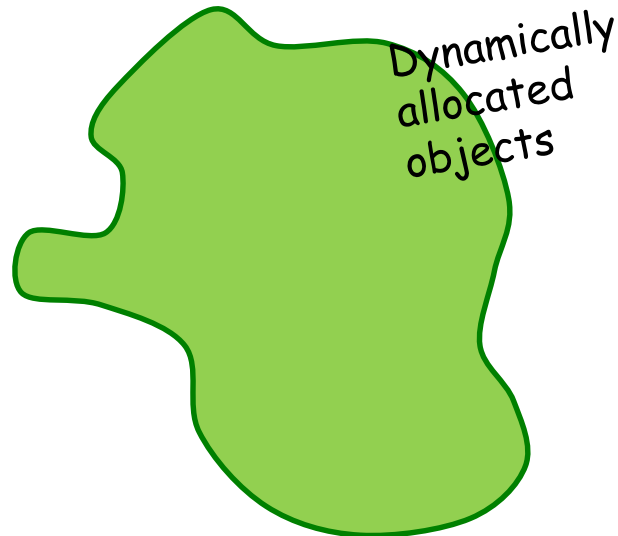
- Program
- Dynamic memory ("Heap")
- Stack

Program



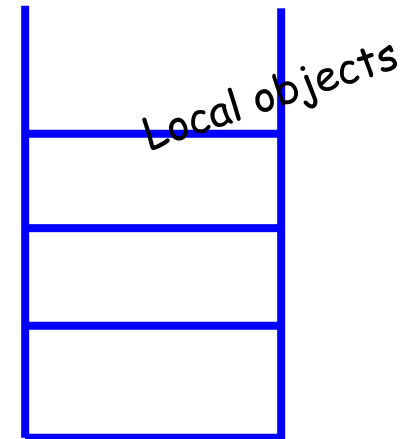
Program cannot modify this memory (self-modified programs are not allowed)

Heap



The discipline of using heap is defined by program **dynamic semantics**, i.e., at runtime (while program execution)

Stack



The discipline of using stack is defined by the (static) **program structure**

Now, starting the real
introduction to C...



The First C Program & Structure

```
int Max(int a, int b)
{
    if ( a > b )
        return a;
    else
        return b;
}

void Input(int* x,int *y);

int main()
{
    int x, y;
    input(*x,*y);
    return Max(x,y);
}
```

Some concrete observations:

- The program contains three function declarations.
- The whole program is within the single source file.
- The execution always starts from the function called `main`.

Common rules:

- The program is a sequence of declarations.
- The whole program can be splitted into several source files (and usually does).
- All program functionality is in functions.

The First C Program & Structure

1. This is the function that accepts two parameters; both should be of integer type. The result of the function should be of integer type.

2. This is the function **algorithm**: what the function actually does.

3. **return** statement specifies the **result** of the function...

4. **input** is the **preliminary** function declaration - without the algorithm. The full function definition is to be provided separately (while program linking).

5. **main** is the "entry point" of the whole program.

6. **main** has two **local declarations** and two **function calls**.

```
int Max(int a, int b) *1
{ *2
    if ( a > b )
        return a;
    else *3
        return b;
}

void Input(int* x, int *y); *4

int main() *5
{
    int x, y;
    Input(&x,&y);
    return Max(x,y); *6
}
```

How C Programs are Built

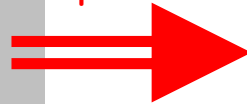
Source & object files, compilation & linking

```
int Max(int a, int b)
{
    if ( a > b )
        return a;
    else
        return b;
}

void Input(int* x, int *y);

int main()
{
    int x, y;
    Input(&x,&y);
    return Max(x,y);
}
```

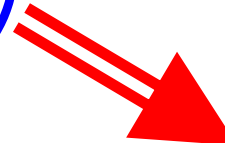
Compilation



Object
file 1

Machine code for
Max & main

Linking



The
resulting
program
ready for
execution

Compiled before



Object
file 2

Machine code
for Input

How C Programs are Built

Translation units and separate compilation

- Typically, any C program consists of several **translation units** each of which is located in a separate source file.
- The **separate compilation principle**; each TU gets compiled **independently** from others.

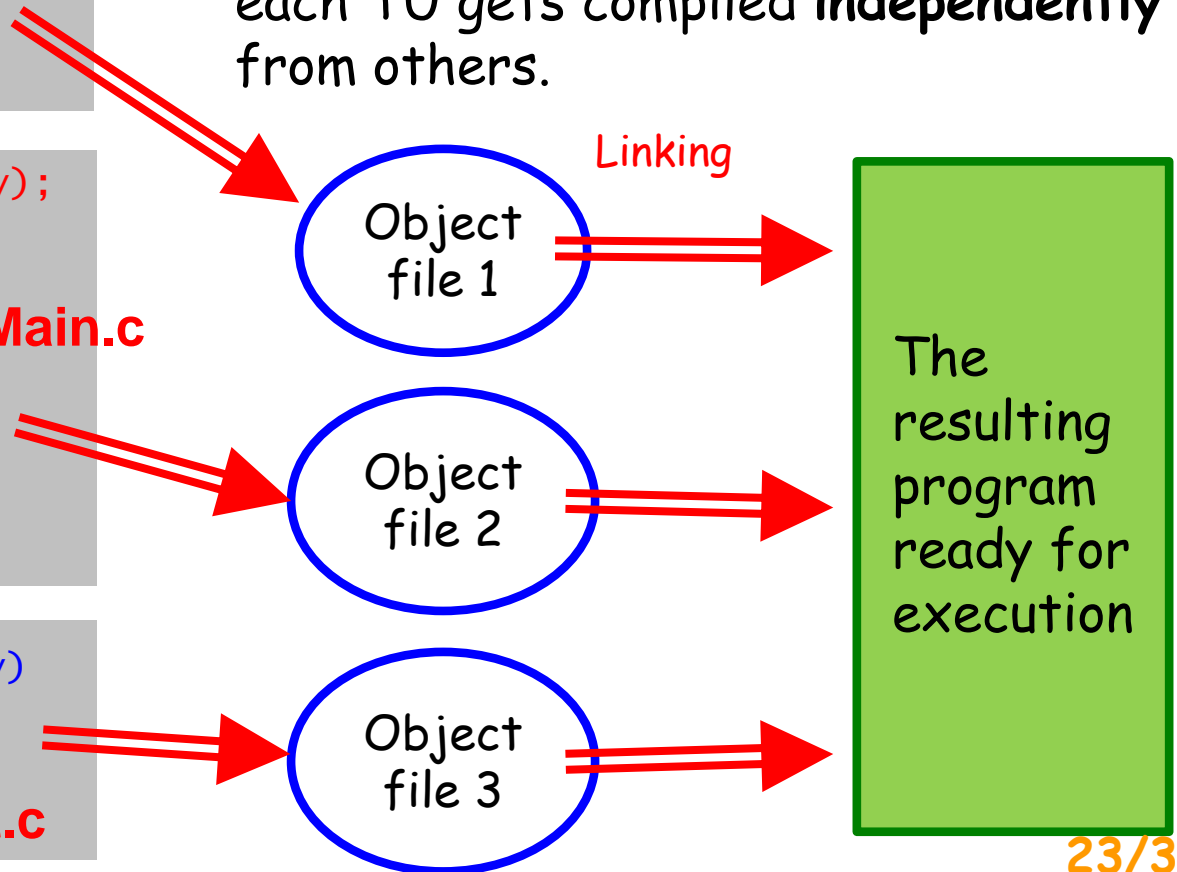
```
int Max(int a, int b) Max.c
{
    if ( a > b )
        return a;
    else
        return b;
}
```

```
void Input(int* x, int *y);
int Max(int a, int b);

int main() Main.c
{
    int x, y;
    Input(&x,&y);
    return Max(x,y);
}
```

```
void input(int* x, int *y)
{
    ...
}
```

Input.c



How C Programs are Built

Interface, implementation, and `#include` directive

What if `Max` and `Input` functions (from the prev slides) are used in many translation units?

- Instead of writing forward declaration for `Max` & `Input` in each TU where they're used, the following solution is used:

Max.h

```
int Max(int a, int b);
```

```
int Max(int a, int b)
{
    if ( a > b )
        return a;
    else
        return b;
}
```

Max.c

Each translation unit is represented by two source files:

- with forward declarations ("interface");
- with full declarations ("implementation").

How C Programs are Built

Interface, implementation, and `#include` directive

...And, instead of writing forward declarations for `Max` and `Input` again and again, we write the following:

Max.h

```
int Max(int a, int b);
```

```
int Max(int a, int b)
{
    if ( a > b )
        return a;
    else
        return b;
}
```

Max.c

```
void Input(int* x, int *y);
int Max(int a, int b);
```

```
int main()
{
    int x;
    Input(&x, &y);
    return Max(x, y);
}
```

```
#include "Input.h"
#include "Max.h"
```

```
int main()
{
    int x, y;
    Input(&x, &y);
    return Max(x, y);
}
```

The semantics of `#include` directive assumes textual inclusion of the contents of the file specified to the file where the directive is written.

C Entities & Declarations

- So, a C program consists of a sequence of **declarations**.
- Each declarations introduces an **entity**.
- What is C entity?
 - **Variable** (simple variable)
 - **Array**
Informally: an indexed group of variables.
 - **Type**
A user-defined type; a synonym to other type
 - **Function**
Informally: a sequence of statements specifying the local context and some actions.

C: Variable Declarations

- `x` variable becomes available in the current context;
- The type of `x` is a default integer type;
- The initial value of `x` is not defined.

```
int x;  
int y = 0123;  
float f1 = 0.1;  
double d1, d2 = 0x555;
```

- `y` variable becomes available in the current context; its type is integer, and the initial value is 83.

- `f1` variable becomes available in the current context; its type is default float, and the initial value is 0.1.

The single declaration introduces two variables: `d1` and `d2`; their type is `double`; the initial value for `d1` is not specified, and for `d2` is 1365.0.

C: Array Declarations

- **A** is the array consisting of 100 integer values; all elements are always **of the same type**;
- The initial values of array elements are not specified;
- The memory for the array is allocated statically: before program starts.
- Array elements are indexed using integer numbers; the first element has the index of 0.

```
int A[100];  
double D[3] = { 1.2, 3.4, 5.6 };
```

- **D** is the array consisting of 3 values of type double each;
- The initial values of array elements are specified by means of the list of values within braces.

C Standard (Predefined) Types

char

_Bool

Signed integer types

signed char
short int
int
long int
long long int

Unsigned integer types

unsigned char
unsigned short int
unsigned int
unsigned long int
unsigned long long int

Floating types

float
double
long double

Complex types

float _Complex
double _Complex
long double _Complex

C Derived ("User-Defined") Types

- Array types
- Structure types
- Union types
- Function types
- Pointer types
- Atomic types

- There is no way to declare an array type independently from an array variable

```
int A[100];
```

This is a **variable** of array type
(The same is about function & pointer types)

- Structure & union types can be declared **separately** (as they are):

```
struct S {  
    int a;  
    int b;  
};
```

Having such a declaration we can use it for declaring **variables** of this type:

```
struct S s;
```

C Derived ("User-Defined") Types

Some tricks & flaws with C types and declarations

```
struct S {  
    int a, b;  
};
```

Usual declaration of a structure type...

We can use it like as follows: **struct S s;**

```
struct S {  
    int a, b;  
} s1, s2;
```

The structure type declaration **together** with variable declaration!

We can still use **S** in declarations: **struct S s3;**

```
struct {  
    int a, b;  
} s1, s2;
```

Unnamed structure type declaration **together** with variable declaration.

```
typedef struct {  
    int a, b;  
} S;
```

Here, we introduce a **synonym** to the unnamed structure type.

Later, we can use the synonym:

S s1, s2;