Programming Software Systems

Introduction to Programming
for the Computer Engineering Track

Tutorial 1

Eugene Zouev
Fall Semester 2020
Innopolis University

Outline

- Some low-level topics:
  - software & hardware layers
  - number representation
  - values & addresses
- How C programs are built: an extended view
- More on program stack
- C entities and declarations
- C type system; predefined and user-defined types.
- The first "real" C program.

Program Execution Layers

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Program Execution Layers

High Level Language Program (e.g., C)

↓ *Compiler*

Assembly Language Program (e.g., MIPS)

↓ *Assembler*

Machine Language Program (MIPS)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Program Execution Layers

High Level Language Program (e.g., C)

↓ *Compiler*

Assembly Language Program (e.g., MIPS)

↓ *Assembler*

Machine Language Program (MIPS)

↓ *Machine Interpretation*

Hardware Architecture Description (e.g., block diagrams)

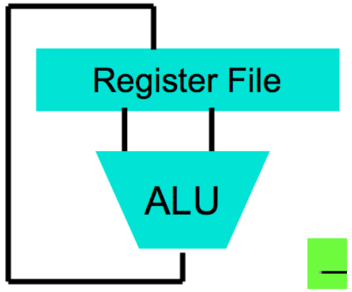
↓ *Architecture Implementation*

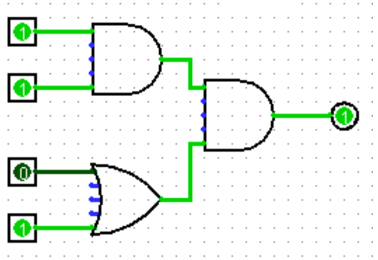
Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```





Program Execution Layers

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

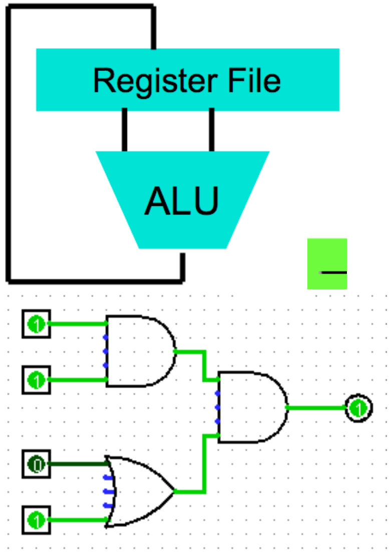
Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

We are here now

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



The architecture of a software program is **much more complicated** than hardware which it runs on.

Some Key Points: Numbers

- Inside computers, **everything is a number.**

  Program instructions, data, …

- Numbers are represented in a **binary code** with a fixed size

  8-bit bytes, 16-bit half words, 32-bit words,
  64-bit double words, …

- Positive and "unsigned" numbers are represented in the **direct code.**

- Negative numbers are represented in the **two's complement code.**

Some Key Points: Numbers

- Inside computers, **everything is a number**.

  Program instructions, data, …

- Numbers are represented in a **binary code** with a fixed size

  See next slides

  8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …

- Positive and "unsigned" numbers are represented in the **direct code**.

  For direct & two's complement code refer to the **Computer Architecture** course or Wikipedia ☺

- Negative numbers are represented in the **two's complement code**.

  See next slides for positive, unsigned and negative numbers

Number Representation

- **Positional notation:**
  Value of i-th digit is ***d* × *Base*^{i}** where i starts at 0 and increases from right to left

- **Mostly used bases**:
  Binary (base 2), Octal (base 8),
  Hexadecimal (base 16), Decimal (base 10)

Decimal digits

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Hexadecimal digits

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
```

Number Representation: Examples

Decimal representation

$$
\begin{aligned}
123_{10} &= 1_{10} \times 10_{10}^{2} + 2_{10} \times 10_{10}^{1} + 3_{10} \times 10_{10}^{0} \\
&= 1\text{x}100_{10} + 2\text{x}10_{10} + 3\text{x}1_{10} \\
&= 100_{10} + 20_{10} + 3_{10} \\
&= 123_{10}
\end{aligned}
$$

Hexadecimal representation

$$
\begin{aligned}
\text{FFF}_{hex} &= 15_{ten}\text{x } 16_{ten}^{2} + 15_{ten}\text{x } 16_{ten}^{1} + 15_{ten}\text{x } 16_{ten}^{0} \\
&= 3840_{ten} + 240_{ten} + 15_{ten} \\
&= 4095_{ten}
\end{aligned}
$$

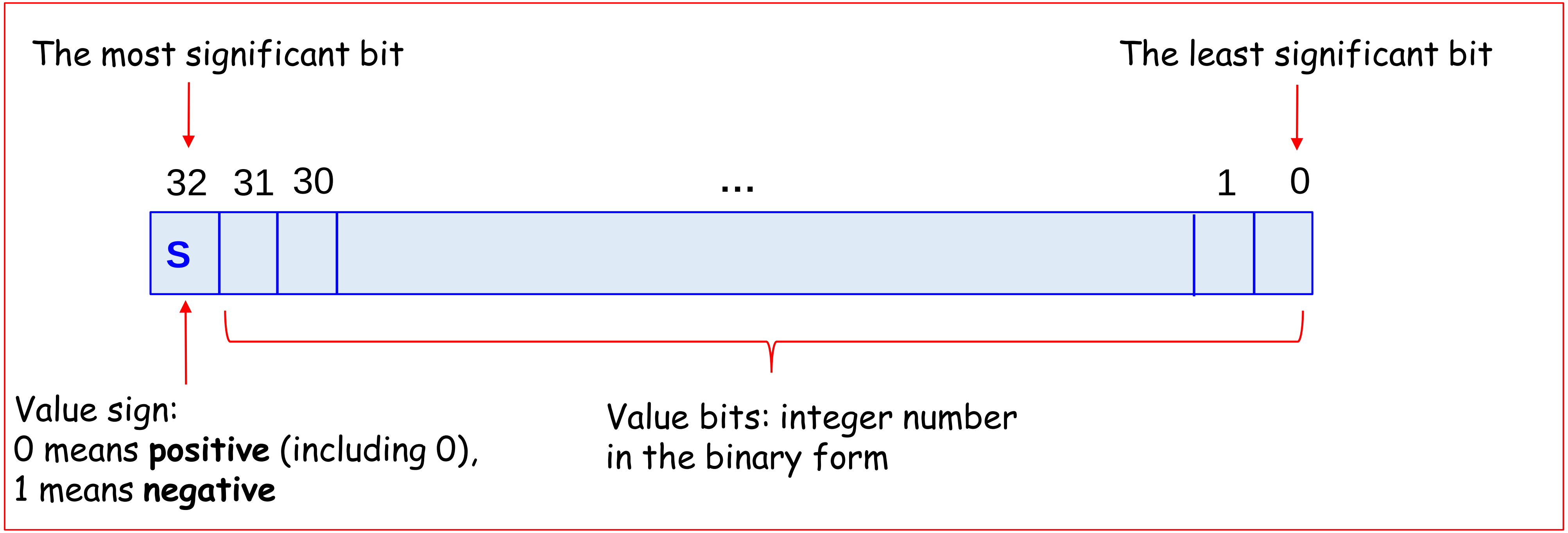
Binary representation

$$1111\ 1111\ 1111_{two} = \text{FFF}_{hex} = 4095_{ten}$$

Numbers: Signed & Unsigned

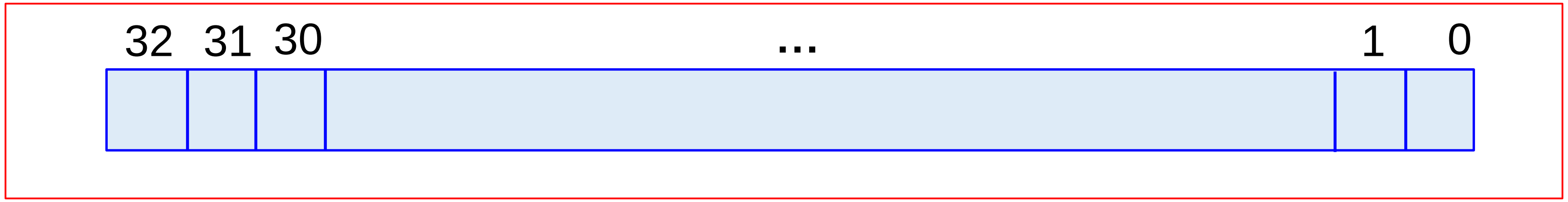
- C, C++, Java etc. have *signed integers*, e.g., +7, -255

32-bit word can represent 2^{32} binary numbers



- C, C++ also have *unsigned integers*, which are used e.g. for representing *addresses*

Unsigned integers in 32 bit word represent 0 to 2^{32}-1 (4'294'967'295)



Memory, Addresses & Values

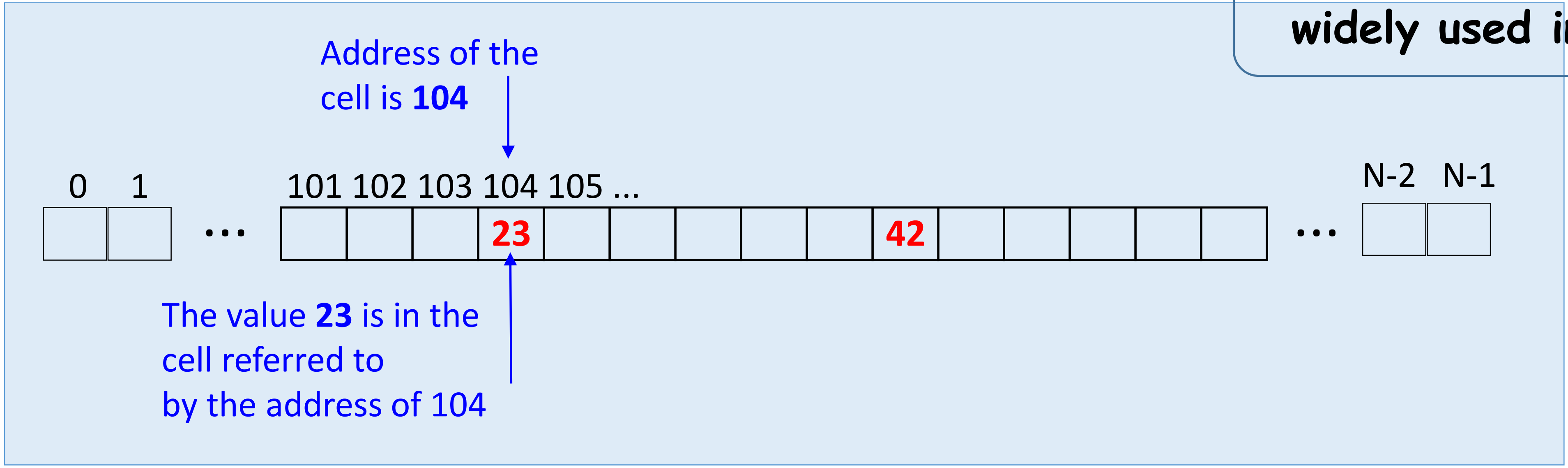
- Memory can be considered as a huge single array (or, a big sequence) of cells
  - **Each cell has an address** associated with it
  - Each cell also stores some value
- Typically, “cell” contains 8 bit
- Don’t confuse the **address** referring to a memory location with the **value** stored there.

These notions are widely used in C

Memory, Addresses & Values

- Memory can be considered as a huge single array (or, a big sequence) of cells
  - **Each cell has an address** associated with it
  - Each cell also stores some value
- Typically, "cell" contains 8 bit
- Don't confuse the **address** referring to a memory location with the **value** stored there.

These notions are widely used in C



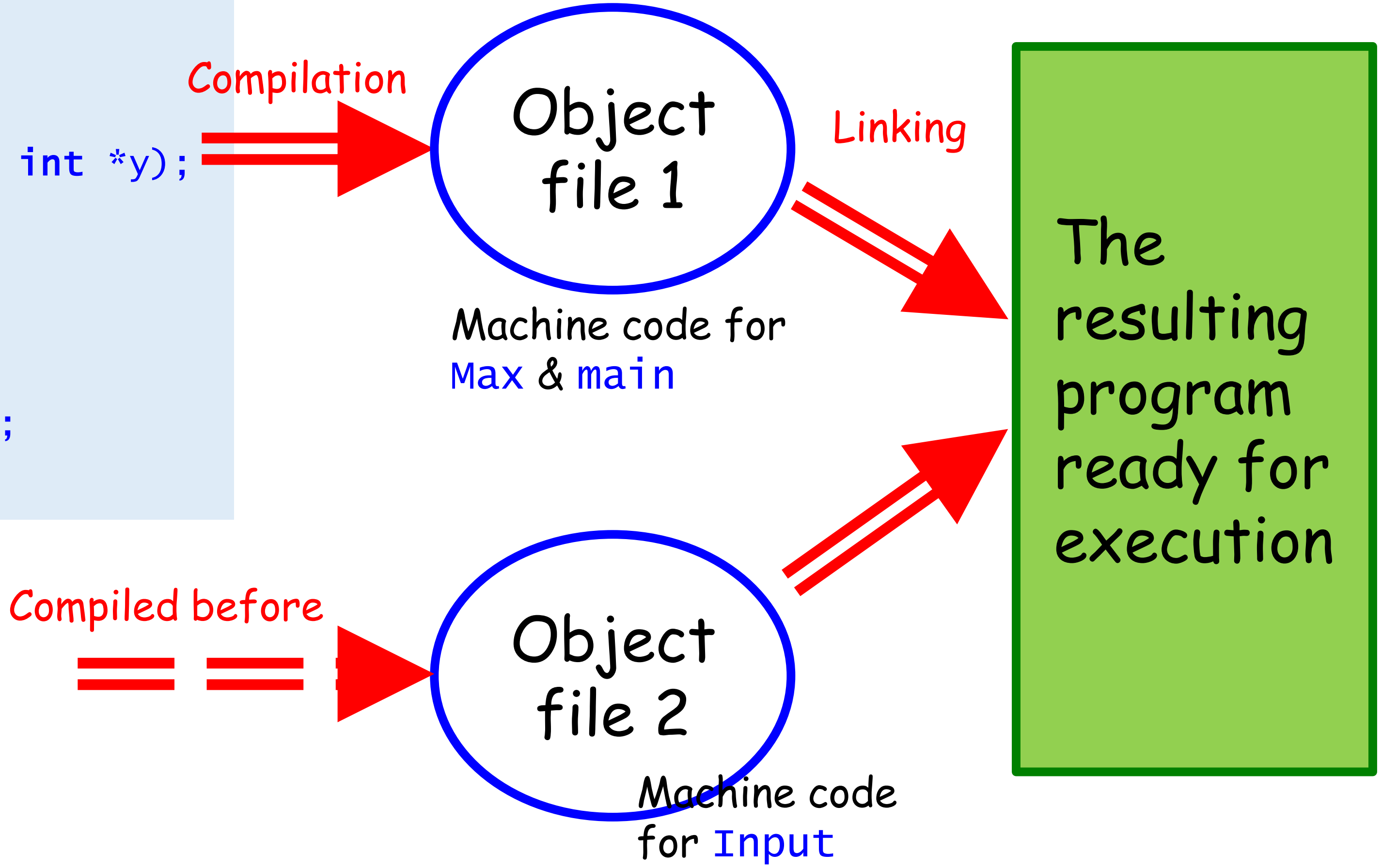
How C Programs are Built

Source & object files, compilation & linking

```c
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);
}
```



How C Programs are Built

Translation units and separate compilation

Max.c

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

Main.c

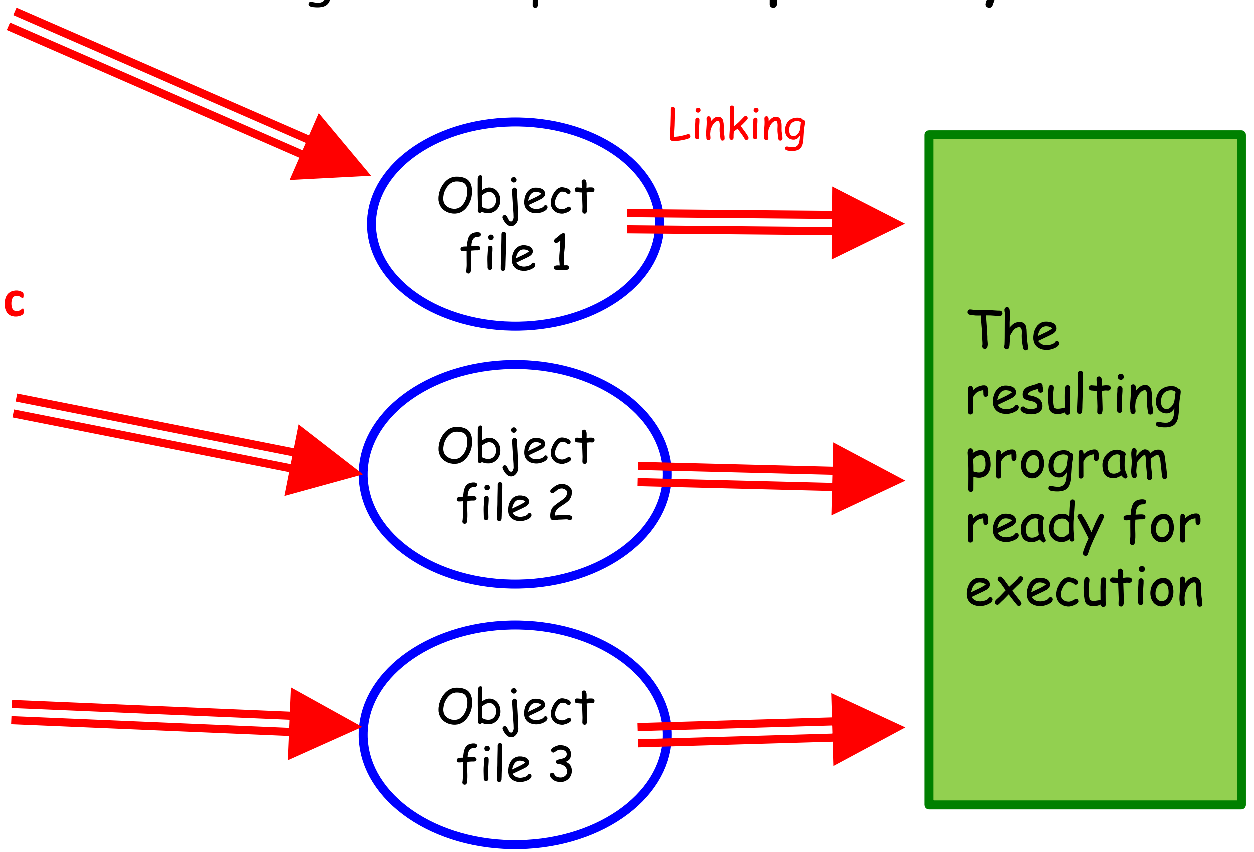
```
void Input(int* x, int *y);
int Max(int a, int b);

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

Input.c

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

- 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.



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

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

```c
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

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

Max.c

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

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

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

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

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

The semantics of #include directive assumes <u>textual inclusion</u> of the contents of the file specified to the file where the directive is written.

Stack & Activation Record 1

```
void f() {
    int i = 3;
    g();
}
void g() {
    int i = 4;
    i = i + h();
}
int h() {
    return 1 + k(5);
}
int k(int z) {
    return z+1;
}
```

Stack & Activation Record 1

Call chain

```
void f() {
    int i = 3;
    g();
}
void g() {
    int i = 4;
    i = i + h();
}
int h() {
    return 1 + k(5);
}
int k(int z) {
    return z+1;
}
```

Stack & Activation Record 1

Call chain

Return chain

```
void f() {
    int i = 3;
    g();
}
void g() {
    int i = 4;
    i = i + h();
}
int h() {
    return 1 + k(5);
}
int k(int z) {
    return z+1;
}
```

Stack & Activation Record 2

- Each time a function is called, **all the information specifically needed for the function** execution are put on the stack
- That information is collectively called the **activation record (AR)** of the function call
- This allows recursion, since for each call there will be a separate activation record on the stack
- When the call is completed (the function "returns") the corresponding AR is destroyed ("popped out" of the stack)
- Activation records are organized from bottom to top in memory diagram
- *All this machinery is controlled by runtime support and (often, partially) by hardware*

Stack & Activation Record 3

The information stored in the AR (also known as Stack Frame) for one call are the following:

- Information to restart the execution at the end of the call, i.e. after the function "returns"; these usually are:
  - **Return address: where to pass the control after exiting from the function**
  - **Pointer to the Stack frame of the calling function**
  - **The value to be returned to the calling function** (if any)
- Information needed to perform the computation (usually the **actual arguments** passed to the function in the call – if any)
- **Local variables** (if any)

Stack & Activation Record 3

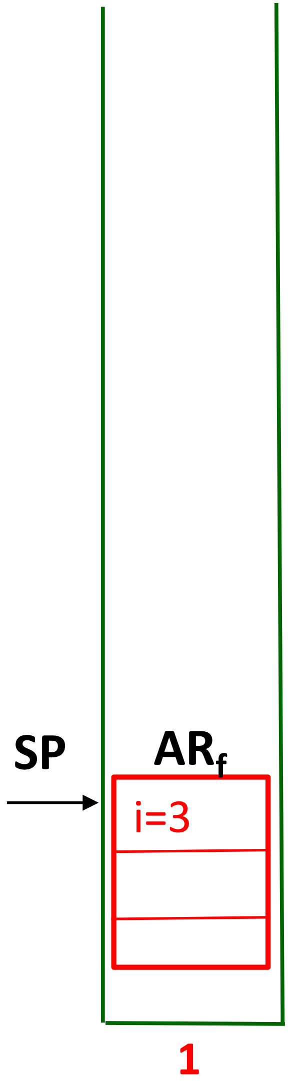
```
1   void f() {
        int i = 3;
        g();            (*)
    }
2   void g() {
        int i = 4;
        i = i + h();  (**)
    }
3   int h() {
        return 1 + k(5); (***)
    }
4   int k(int z) {
        return z+1;
    }
```

AR – Activation Record
SP – Stack Pointer
RA – Return Address
RV – Return Value
DL – Dynamic Link

Stack & Activation Record 3

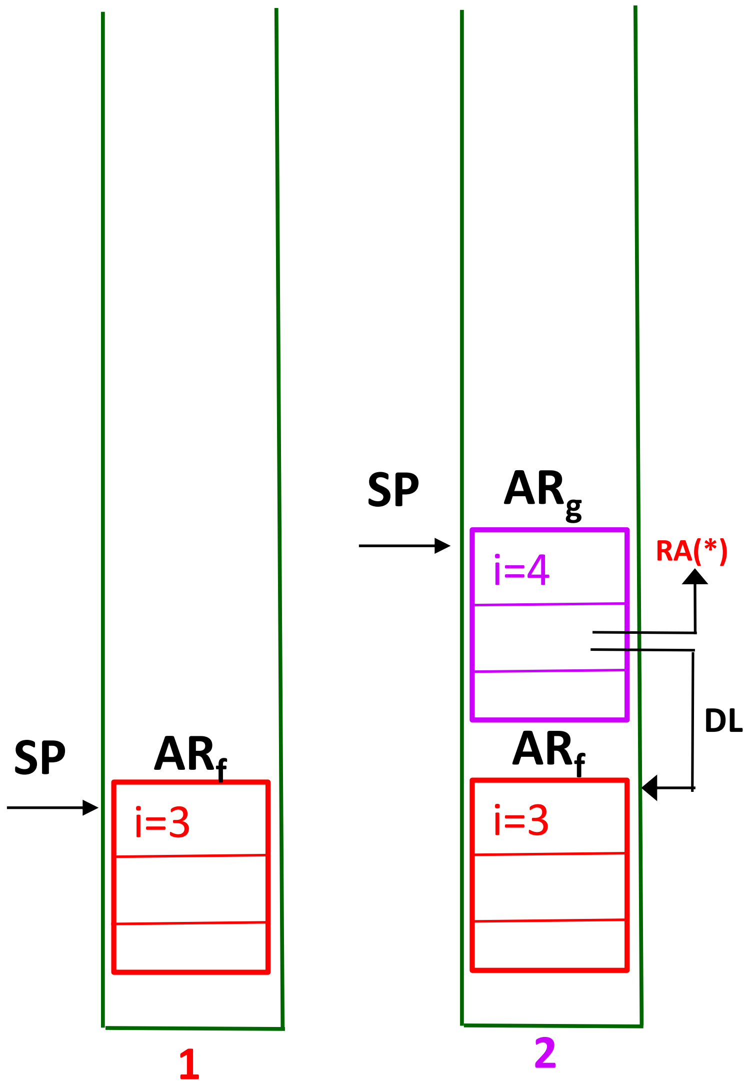
```
1  void f() {
       int i = 3;
       g();              (*)
   }
2  void g() {
       int i = 4;
       i = i + h();   (**)
   }
3  int h() {
       return 1 + k(5);  (***)
   }
4  int k(int z) {
       return z+1;
   }
```

AR – Activation Record
SP – Stack Pointer
RA – Return Address
RV – Return Value
DL – Dynamic Link



Stack & Activation Record 3

1
```
void f() {
    int i = 3;
    g();            (*)
}
```
2
```
void g() {
    int i = 4;
    i = i + h();  (**)
}
```
3
```
int h() {
    return 1 + k(5);  (***)
}
```
4
```
int k(int z) {
    return z+1;
}
```

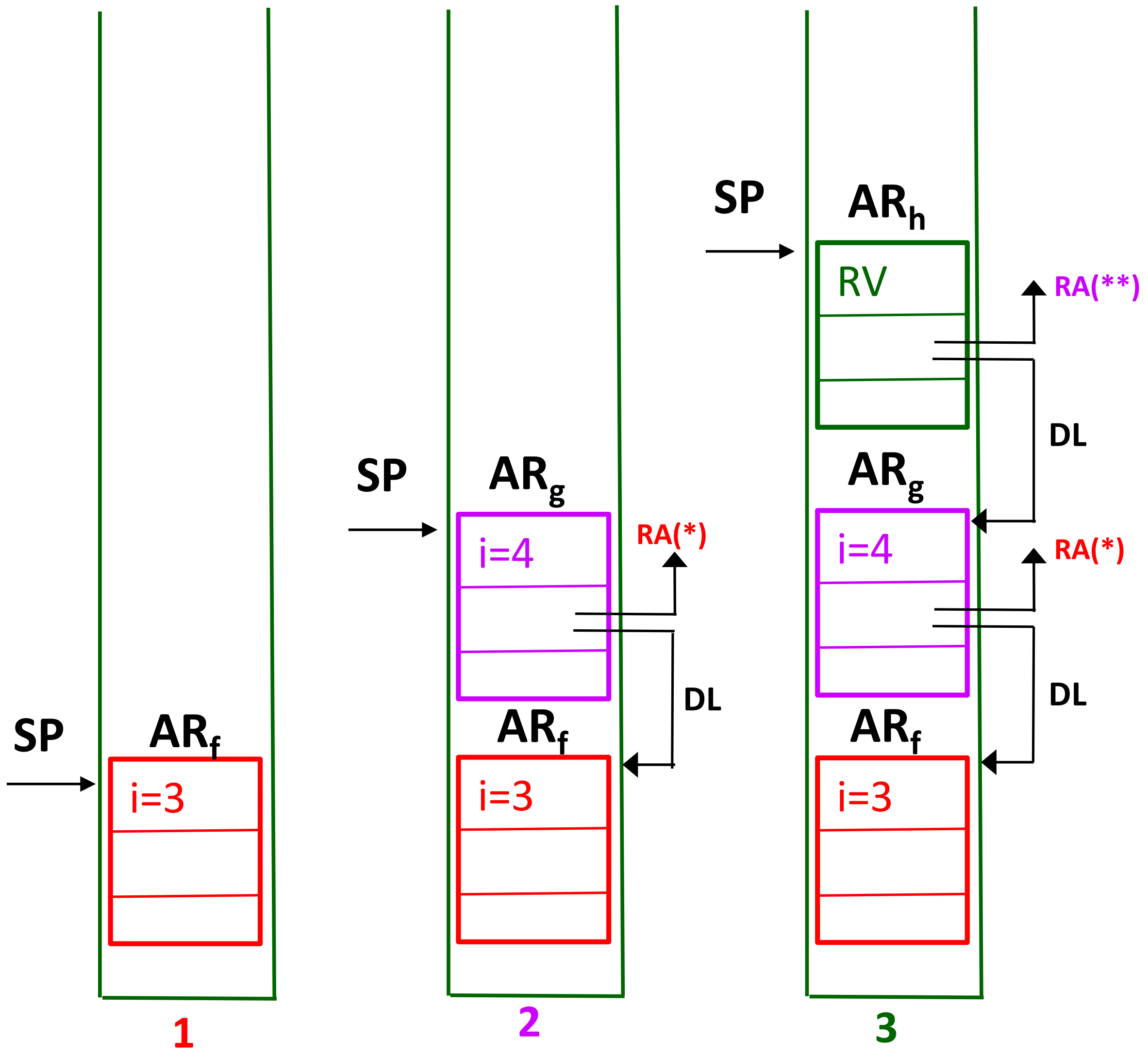


AR – Activation Record
SP – Stack Pointer
RA – Return Address
RV – Return Value
DL – Dynamic Link

Stack & Activation Record 3

```
1  void f() {
       int i = 3;
       g();            (*)
   }
2  void g() {
       int i = 4;
       i = i + h();  (**)
   }
3  int h() {
       return 1 + k(5); (***)
   }
4  int k(int z) {
       return z+1;
   }
```

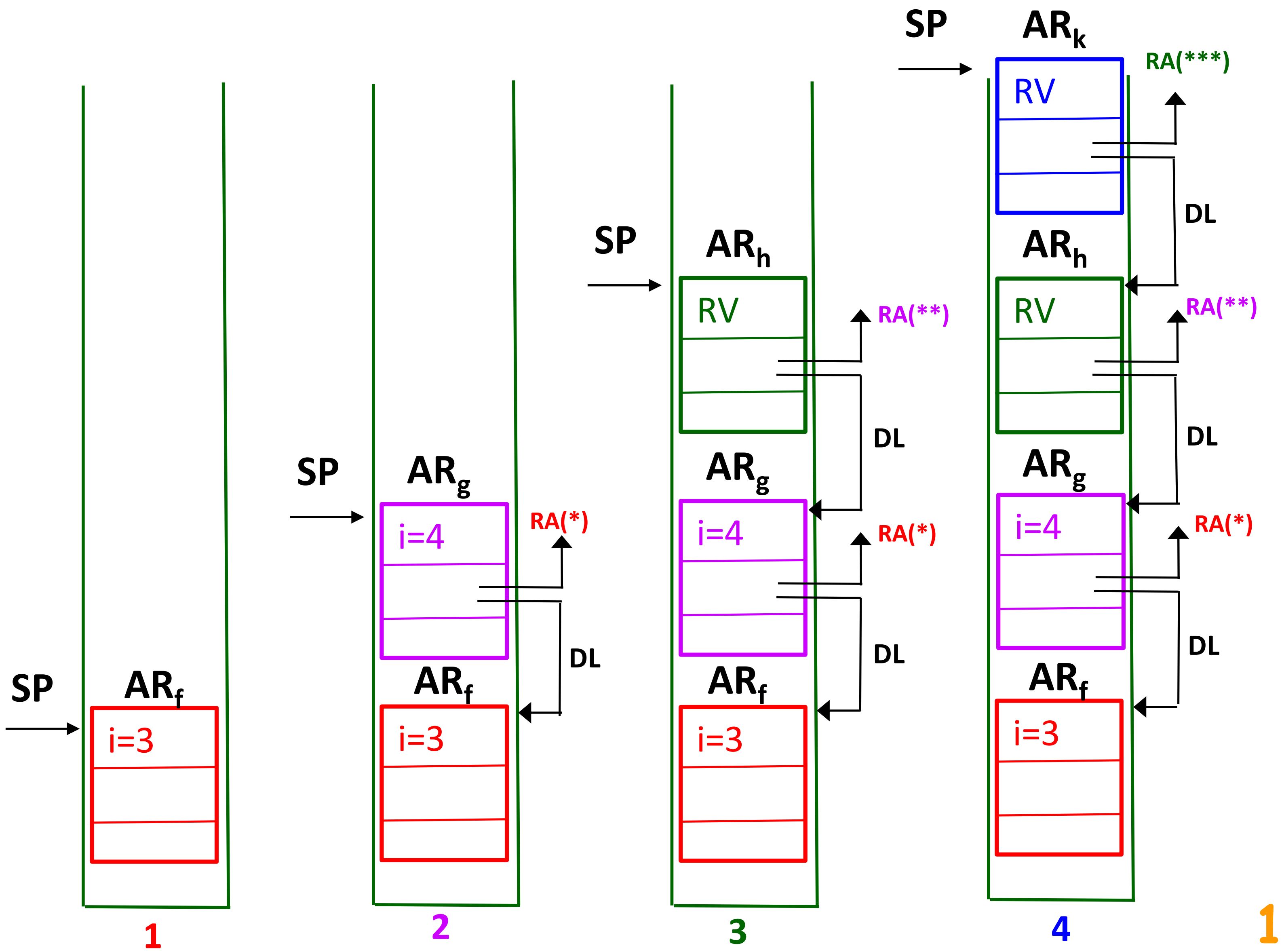
AR – Activation Record
SP – Stack Pointer
RA – Return Address
RV – Return Value
DL – Dynamic Link



Stack & Activation Record 3

```
1  void f() {
       int i = 3;
       g();          (*)
   }
2  void g() {
       int i = 4;
       i = i + h();  (**)
   }
3  int h() {
       return 1 + k(5);  (***)
   }
4  int k(int z) {
       return z+1;
   }
```

AR – Activation Record
SP – Stack Pointer
RA – Return Address
RV – Return Value
DL – Dynamic Link



C Entities & Declarations

- Syntactically, a C program consists of a sequence of **declarations**.
- Each declaration 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

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

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

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

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];
```

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 the 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;
```

The First "Real" C Function

```
int gcd(int x, int y)
{
  int a = x, b = y;
  while ( a != 0 )
  {
    int temp = a;
    a = b % a
    b = temp
  }
  return b;
}
```

Euclid algorithm:
Finds the greatest common denominator for two numbers
Наибольший общий делитель

The First "Real" C Function

```
int gcd(int x, int y)
{
  int a = x, b = y;
  while ( a != 0 )
  {
    int temp = a;
    a = b % a
    b = temp
  }
  return b;
}
```

Euclid algorithm:
Finds the greatest common denominator for two numbers

Наибольший общий делитель

Some important points:

- The algorithm is organized as a series of **steps**.
- The variables **change their values** on each step.
- There are **three local variables** used in the algorithm.
- This is the **iterative** algorithm (with loop).

The First “Real” C Function

```
int gcd(int x, int y)
{
   int a = x, b = y;
   while ( a != 0 )
   {
      int temp = a;
      a = b % a
      b = temp
   }
   return b;
}
```

Euclid algorithm:
Finds the greatest common denominator for two numbers
Наибольший общий делитель

Imperative paradigm

Some important points:

- The algorithm is organized as a series of **steps**.
- The variables **change their values** on each step.
- There are **three local variables** used in the algorithm.
- This is the **iterative** algorithm (with loop).

The First "Real" C Function

```
int gcd(int x, int y)
{
   int a = x, b = y;
   while ( a != 0 )
   {
      int temp = a;
      a = b % a
      b = temp
   }
   return b;
}
```

Euclid algorithm:
Finds the greatest common denominator for two numbers
Наибольший общий делитель

Imperative paradigm

Some important points:

- The algorithm is organized as a series of **steps**.
- The variables **change their values** on each step.
- There are **three local variables** used in the algorithm.
- This is the **iterative** algorithm (with loop).

Is it the best implementation of the Euclid algorithm?

The First "Real" C Function

```
int gcd(int x, int y)
{
  int a = x, b = y;
  while ( a != 0 )
  {
    int temp = a;
    a = b % a
    b = temp
  }
  return b;
}
```

```
int gcd(int x, int y)
{
  if (y == 0)
    return x;
  else
    return gcd(y, x%y);
}
```

The First "Real" C Function

```c
int gcd(int x, int y)
{
  int a = x, b = y;
  while ( a != 0 )
  {
    int temp = a;
    a = b % a
    b = temp
  }
  return b;
}
```

```c
int gcd(int x, int y)
{
  if (y == 0)
    return x;
  else
    return gcd(y, x%y);
}
```

Functional paradigm

Important points:

- **No** local variables.
- Variables (parameters) do not **change their values**.
- This is the **recursive** algorithm: recursion is used instead of iteration
- The code is much more concise and readable.

The First "Real" C Function

```
int gcd(int x, int y)
{
  int a = x, b = y;
  while ( a != 0 )
  {
    int temp = a;
    a = b % a
    b = temp
  }
  return b;
}
```

```
int gcd(int x, int y)
{
  if (y == 0)
    return x;
  else
    return gcd(y, x%y);
}
```

Can we make the function even more compact? ☺

Functional paradigm

Important points:

- **No** local variables.
- Variables (parameters) do not **change their values**.
- This is the **recursive** algorithm: recursion is used instead of iteration
- The code is much more concise and readable.

The First "Real" C Program

```
int gcd(int x, int y)
{
  return (y == 0) ? x : gcd(y, x%y);
}
```

The First "Real" C Program

```
#include <stdio.h>

int gcd(int x, int y)
{
  return (y == 0) ? x : gcd(y, x%y);
}

int main()
{
  int m, n;
  scanf("%d%d",&m,&n);
  printf("%d\n",gcd(m,n));
  return 1;
}
```

The First "Real" C Program

Textual inclusion of function declarations for input/output from the standard C library (scanf & printf are among them)

The main function: C programs always start execution from it

scanf reads values from the console; printf outputs its arguments to the console.

```
#include <stdio.h>

int gcd(int x, int y)
{
  return (y == 0) ? x : gcd(y, x%y);
}

int main()
{
  int m, n;
  scanf("%d%d",&m,&n);
  printf("%d\n",gcd(m,n));
  return 1;
}
```