Programming Software Systems

Introduction to Programming for the Computer Engineering Track

Lecture 1
Introduction

Eugene Zouev
Fall Semester 2020
Innopolis University

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++ front-end compiler
  - Interstron Ltd., Moscow, 1999-2000.
- Zonnon language implementation for .NET & Visual Studio
  - ETH Zürich, 2005.
- Swift prototype compiler for Tizen & Android
  - Samsung Research, 2015
- Six (or seven? ☺) books; the latest are
  - «Редкая профессия», ДМК Пресс, Москва 2014.
  - Software Design for Resilient Computer Systems, Springer, 2019

Why the Course?

- Programming is the fundamental skill in computer science – whatever area you choose in your professional career.
- A professional should know several programming languages…
- …Moreover: (s)he should be able to quickly learn any new language, software technology or a framework…
- And for that, you should know basic concepts that are common to many (if not all) programming languages: type, algorithm, control flow, expressions/statements, syntax/semantics, software lifecycle, OOP, and many other.

I'm sure you have some experience in practical programming. But do you really understand (and can explain) notions used in your code?

The Fall Semester: The Schedule

	B20-01	B20-02	B20-03	B20-04
Tuesday				
10:40-12:10	Programming Software Systems 1 (Lecture) **Eugene Zouev** 106			
12:40-14:10	Programming Software Systems 1 (Tutorial) **Eugene Zouev** 106			
Friday				
14:20-15:50		Programming Software Systems 1 (Lab) **Sirojiddin Komolov** 301		Programming Software Systems 1 (Lab) **Mansur Khazeev**
16:00-17:30	Programming Software Systems 1 (Lab) **Sirojiddin Komolov** 301		Programming Software Systems 1 (Lab) **Mansur Khazeev** 321	

Organization

Contents

- **Lectures**:
  Theory, general stuff.
  Language concepts will be presented first
- **Tutorials**:
  Extra stuff. Examples to illustrate what was presneted during the lecture + particular aspects
- **Labs**:
  Allow you to get practcal experince in programming

Organization

Contents

- **Lectures**:
  Theory, general stuff.
  Language concepts will be presented first

- **Tutorials**:
  Extra stuff. Examples to illustrate what was presneted during the lecture + particular aspects

- **Labs**:
  Allow you to get practcal experince in programming

Moodle

- ***All*** information will be on Moodle (http://moodle.innopolis.university)
- There you will find:
  - the lecture material, just after the class (sometime before)
  - and the lab sessions with exercises and information about the project and the assignments
- Plus any other information and all your grades

Exams, Evaluation & Grading

Examinations

- *Assignments:*
  to be evaluated each week
- Mid-term examination:
  written form (quiz; ~13th Oct.)
- Final exam:
  written form (program tasks)

Exams, Evaluation & Grading

Examinations

- *Assignments:*
  to be evaluated each week
- Mid-term examination:
  written form (quiz; ~13th Oct.)
- Final exam:
  written form (program tasks)

Assessment

- Mid-term Exam (25%),
- Final Exam (30%)
- Lab assignments (40%)
- Lab attendance (5%)

Grading

- A [90, 100]
- B [75, 90)
- C [60, 75)
- D [0, 60)

Required Background & Workload

The course is intended to be self-contained, requiring basic knowledge of math including binary calculus and common sense ☺

The will to learn is a key prerequisite !

Overall the course should take on average **12 hours per week** of your life ☺

Prof M. Mazzara: only 50% of material will be given on lectures/tutorials; the other is the matter of **your own study**

The Overall Structure of the Course

Three main parts of the course

- **The C language**

  Small, system-level (but still general-purpose) language

The Overall Structure of the Course

Three main parts of the course

The fall semester

- **The C language**

  Small, system-level (but still general-purpose) language

- **The Java language**

  Powerful application language

The Overall Structure of the Course

Three main parts of the course

The fall semester

- **The C language**

  Small, system-level (but still general-purpose) language

- **The Java language**

  Powerful application language

The spring semester

- **The C++ language**

  Fast and powerful general-purpose language **with deep semantics**

Before we start…

A remark about language syntax & semantics

Syntax:
A set of rules that regulate **the structure** of programs and their parts (constructs)

Before we start...

A remark about language syntax & semantics

Syntax:
A set of rules that regulate
the structure of programs
and their parts (constructs)

Semantics:
The **meaning** of the constructs

Static semantics:
- How programs get compiled
Dynamic semantics:
- How programs get executed.

Before we start…

A remark about language syntax & semantics

"Usual" view at a language:

Syntax:
A set of rules that regulate **the structure** of programs and their parts (constructs)

Semantics:
The **meaning** of the constructs

Static semantics:
- How programs get compiled

Dynamic semantics:
- How programs get executed.

Syntax

Semantics

Before we start…

A remark about language syntax & semantics

"Usual" view at a language:

Syntax:
A set of rules that regulate **the structure** of programs and their parts (constructs)

Semantics:
The **meaning** of the constructs

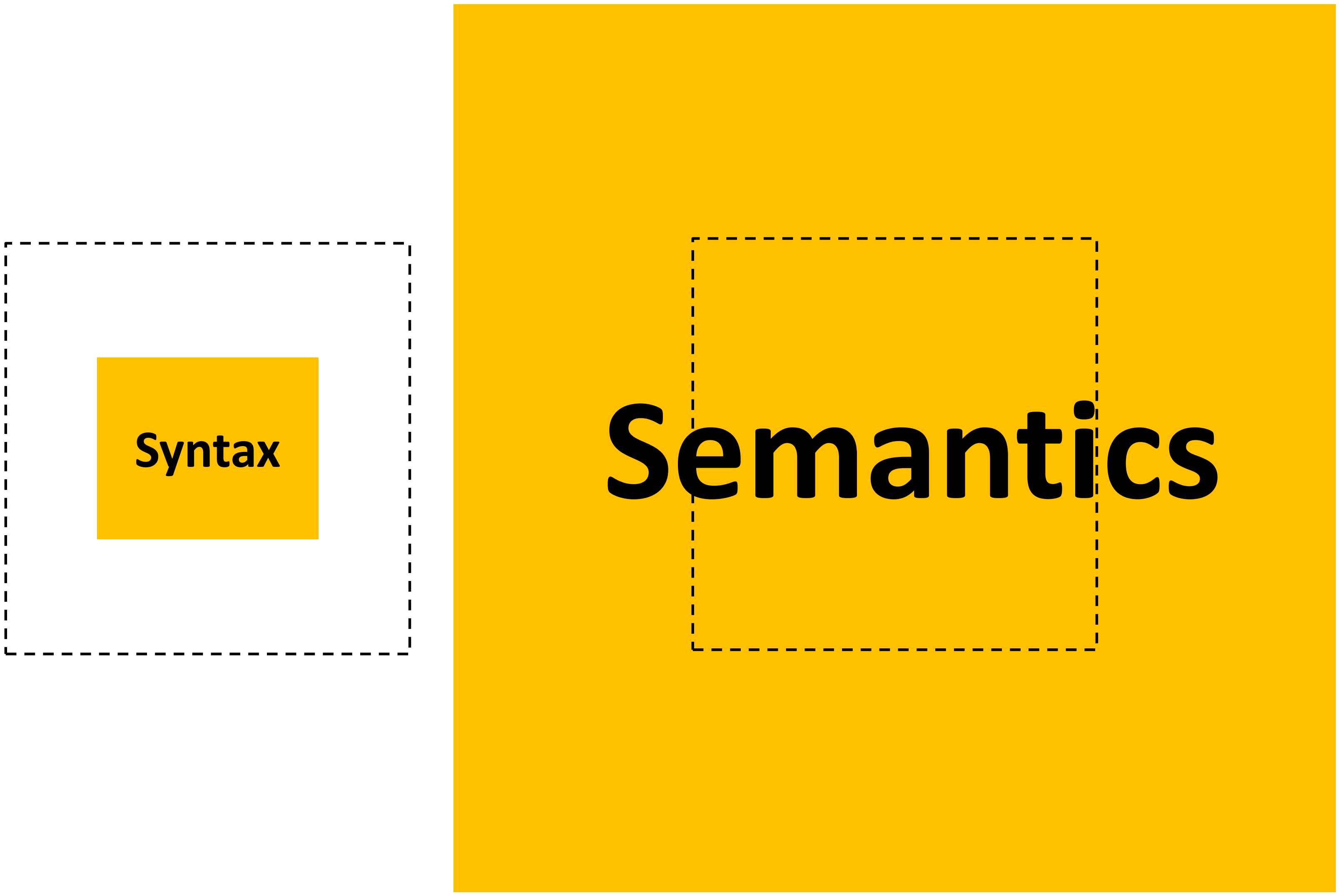
<u>Static</u> semantics:
- How programs get compiled

<u>Dynamic</u> semantics:
- How programs get executed.

Before we start…

A remark about language syntax & semantics

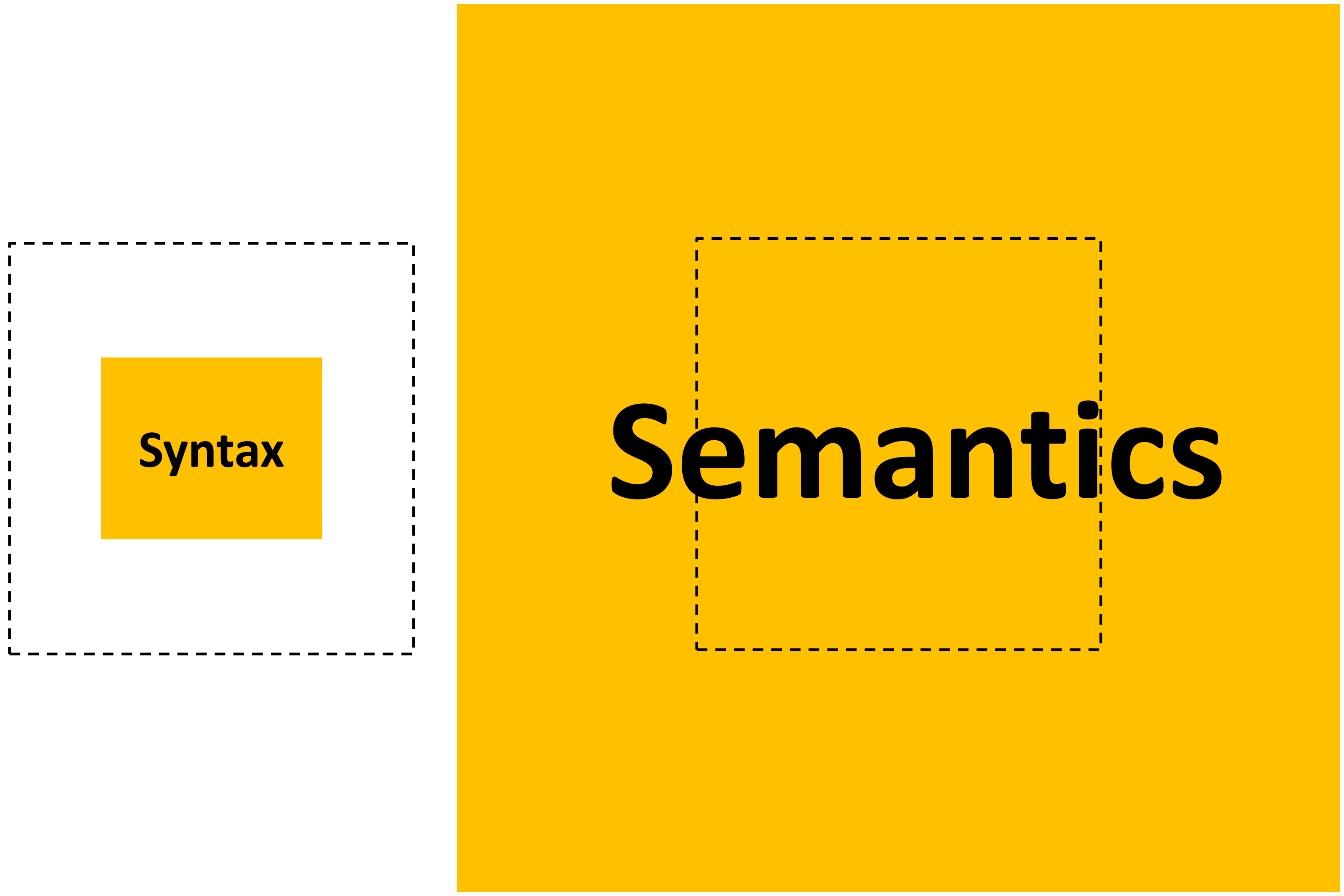
Reality:



Before we start…

A remark about language syntax & semantics

Reality:



Conclusion for programmers:

- Pay most attention on the language <u>semantics</u> rather then on syntax

C: Syntax vs Semantics

```
int x = a + b;
```

C: Syntax vs Semantics

Lexics

Keyword Identifiers Operator signs Delimiter

```
int x = a + b;
```

C: Syntax vs Semantics

Lexics

Keyword Identifiers Operator signs Delimiter

```
int x = a + b;
```

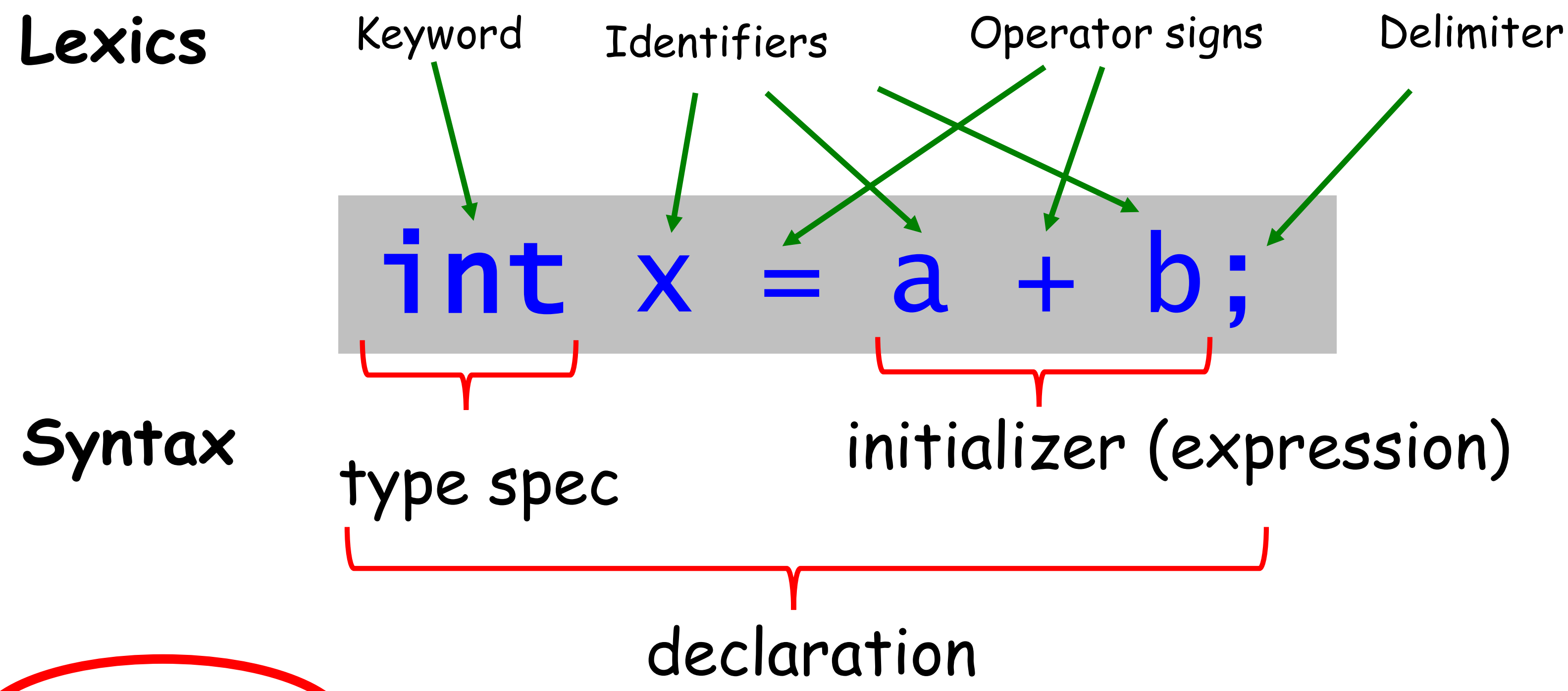
Syntax

type spec

initializer (expression)

declaration

C: Syntax vs Semantics



Semantics

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

Program Lifecycle: Compilation

Program source text

```
int main()
{
  Stack<double> stack1;
  Stack<int> stack2(5);
  int y = 1;
  double x = 1.1;
  int i, j;
  cout << "\n pushed values into stack1: ";
  for ( i=1; i<=11; i++)
  {
    if (stack1.push(i*x))
      cout << endl << i*x;
    else
      cout << "\n stack1 is full";
  }
  cout << "\n\n popd values from stack1:\n";
  for (i=1; i<=6; i++)
    cout << stack1.pop() << endl;
  ...
```

Is this a program? ☺

Program Lifecycle: Compilation

Program source text

```
int main()
{
  Stack<double> stack1;
  Stack<int> stack2(5);
  int y = 1;
  double x = 1.1;
  int i, j;
  cout << "\n pushed values into stack1: ";
  for ( i=1; i<=11; i++)
  {
    if (stack1.push(i*x))
      cout << endl << i*x;
    else
      cout << "\n stack1 is full";
  }
  cout << "\n\n popd values from stack1:\n";
  for (i=1; i<=6; i++)
    cout << stack1.pop() << endl;
  ...
```

Machine code

```
0x006 77 22378EE
0x007 00 0000001
0x008 33 1017700
0x009 7B 00178AB
0x00A 7B 00178AB
0x00B 72 037CEFF
0x00C 3D AFFFFED
0x00E 72 037CEFF
0x00F 3D AFFFFED
0x00D 7B 00178AB
0x00E 3D CAFEBEB
0x00F 3D 00011FF
...
```

Execution

Is this a program? ☺
- **No**: this is just a text

This is this a program

Program Lifecycle: Compilation

Program source text

```
int main()
{
  Stack<double> stack1;
  Stack<int> stack2(5);
  int y = 1;
  double x = 1.1;
  int i, j;
  cout << "\n pushed values into stack1: ";
  for ( i=1; i<=11; i++)
  {
    if (stack1.push(i*x))
      cout << endl << i*x;
    else
      cout << "\n stack1 is full";
  }
  cout << "\n\n popd values from stack1:\n";
  for (i=1; i<=6; i++)
    cout << stack1.pop() << endl;
  ...
```

COMPILER

Machine code

```
0x006 77 22378EE
0x007 00 0000001
0x008 33 1017700
0x009 7B 00178AB
0x00A 7B 00178AB
0x00B 72 037CEFF
0x00C 3D AFFFFED
0x00E 72 037CEFF
0x00F 3D AFFFFED
0x00D 7B 00178AB
0x00E 3D CAFEBEB
0x00F 3D 00011FF
...
```

Execution

Is this a program? ☺
- **No**: this is just a text

This is this a program

Compiler transforms the program text into a semantically equivalent sequence of machine instructions

We will consider **other kinds** of program lifecycles later

The Common Memory Model
Conceptual View

Each program uses three kinds of memory:
- **Program**
- **Dynamic memory** ("Heap")
- **Stack**

The Common Memory Model

Conceptual View

Each program uses three kinds of memory:

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

Program

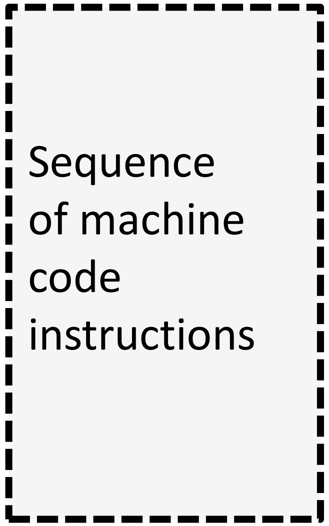
Sequence of machine code instructions

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

The Common Memory Model
Conceptual View

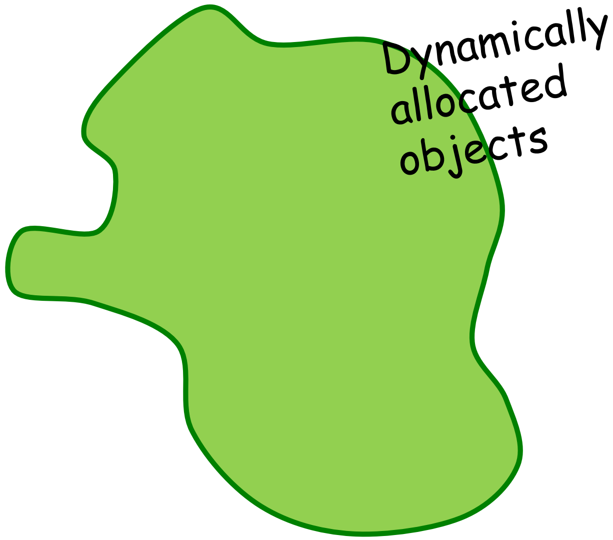
Each program uses three kinds of memory:
- **Program**
- **Dynamic memory** ("Heap")
- **Stack**

Program



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

Heap

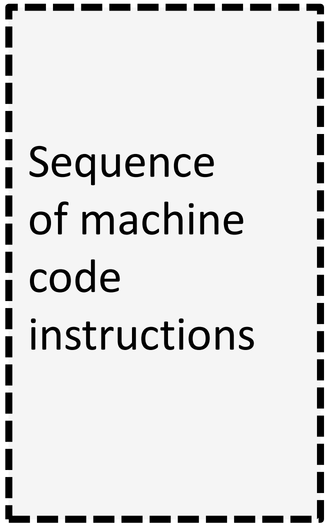


The discipline of using heap is defined by program **dynamic semantics**, i.e., at runtime (while program execution)

The Common Memory Model
Conceptual View

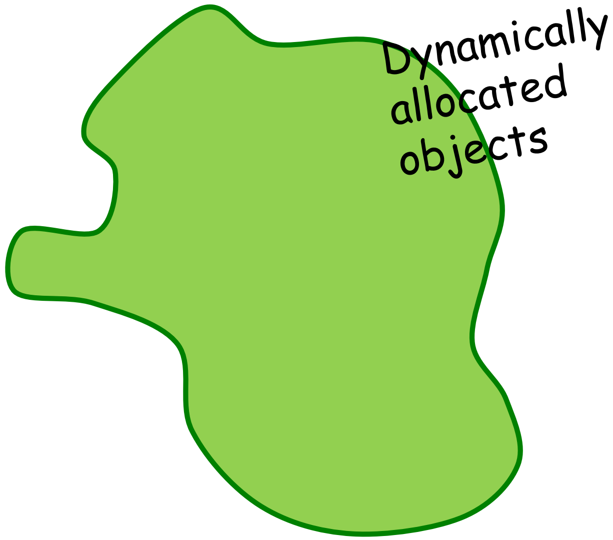
Each program uses three kinds of memory:
- **Program**
- **Dynamic memory** ("Heap")
- **Stack**

Program



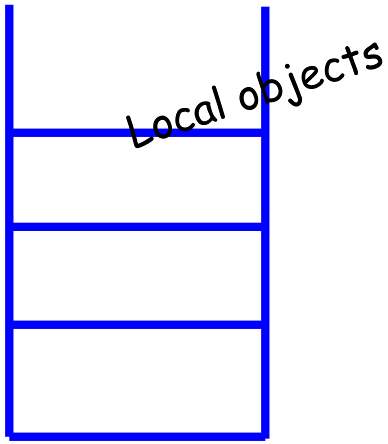
Program cannot modify this memory: self-modified programs are not allowed

Heap



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

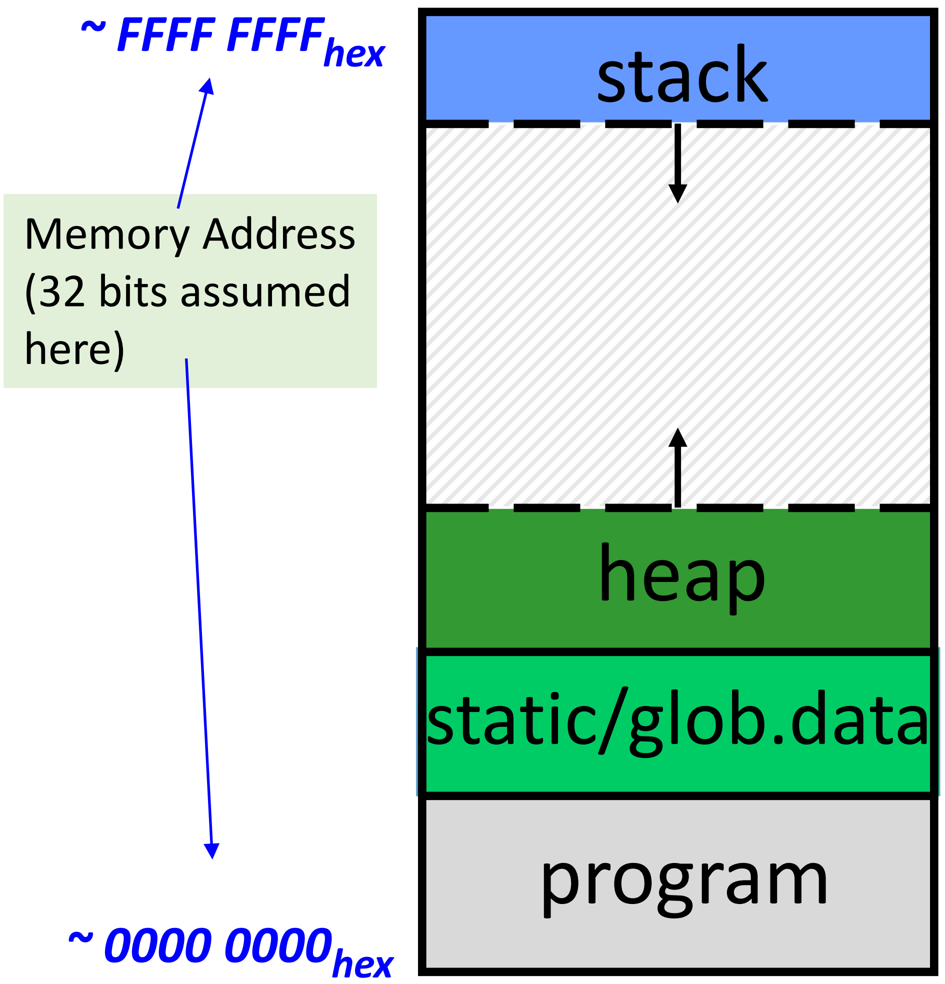
Stack



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

The Common Memory Model
More Detailed View

- **Everything's number and everything's in memory: both program and data**
- Program's *address space* contains 4 regions:
  - stack: local variables inside functions, <u>grows downward</u>
  - heap: space requested for dynamic data; resizes dynamically, <u>grows upward</u>
  - static data: variables declared outside functions, <u>does not grow or shrink</u>. Loaded when program starts, can be modified.
  - **code**: loaded when program starts, <u>does not change</u>



The C Language: Authors

Brian Kernighan

Dennis Ritchie

The C Language: Initial Remarks

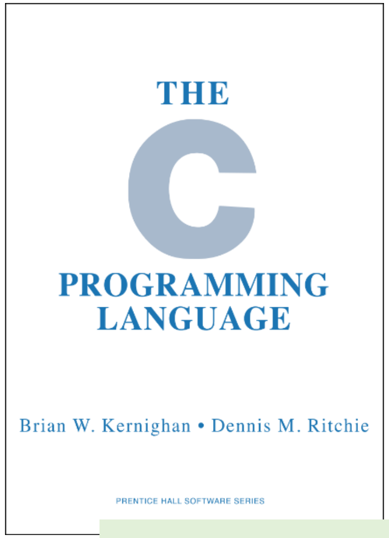
- C is not a "very high-level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

  – Kernighan and Ritchie

- Using C, we can write programs that allow us to exploit underlying features of the architecture – memory management, special instructions, parallelism.

References

- 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…)



“K&R C”

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.

"The universal assembly language"

- 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.
- 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 **very** popular (see any TIOBE index)
- C is the 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.
- Key C concepts: Variable, Pointer, Array, Structure, Function.
- 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.
- C is unsafe
  - C is an efficient language, but leaves safety to the programmer

The First C Program & Structure

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

char* hello = "Hello";

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

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

The First C Program & Structure

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

char* hello = "Hello";

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

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

Some concrete observations:

- The program contains four declarations: 3 functions, and one string.
- The whole program is placed within the single source file.
- The execution always starts from the function called `main`.

The First C Program & Structure

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

char* hello = "Hello";

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

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

Some concrete observations:

- The program contains four declarations: 3 functions, and one string.
- The whole program is placed 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 may consist of 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.

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

char* hello = "Hello";

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

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

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.

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

char* hello = "Hello";

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

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

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...

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

char* hello = "Hello";

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

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

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).

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

char* hello = "Hello";

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

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

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.

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

char* hello = "Hello";

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

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

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` contains two variable **declarations** and two **function calls**.

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

char* hello = "Hello";

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

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

The First C Program & Structure

Several **kinds of variables**:

- **Function parameters**: they are <u>local</u> to the function. Parameters are created and initialized automatically, when the function gets invoked. (Place: **stack**)

Function parameters

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

char* hello = "Hello";

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

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

The First C Program & Structure

Several **kinds of variables**:

- **Function parameters**: they are local to the function. Parameters are created and initialized automatically, when the function gets invoked. (Place: **stack**)

- **Global variables**: they are created once, automatically, when the program starts. (Place: **stack**)

```
int Max(int a, int b)
{                       Function parameters
    if ( a > b )
        return a;
    else
        return b;
}
          Global variable
char* hello = "Hello";

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

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

The First C Program & Structure

Several **kinds of variables**:

- **Function parameters**: they are <u>local</u> to the function. Parameters are created and initialized automatically, when the function gets invoked. (Place: **stack**)

- **Global variables**: they are created once, automatically, when the program starts. (Place: **stack**)

- **Local variables**: they exist (accessible) only within their scopes. (Here the scope is the body of a function.) Locals are created dynamically when the control flow enters the scope where they were declared. (Place: **stack**)

```c
int Max(int a, int b)
{                      // Function parameters
    if ( a > b )
        return a;
    else
        return b;
}
                       // Global variable
char* hello = "Hello";

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

int main()
{
    int x, y;          // Local variables
    input(&x,&y);
    return Max(x,y);
}
```

The First C Program & Structure

Several **kinds of variables**:

- **Function parameters**: they are <u>local</u> to the function. Parameters are created and initialized automatically, when the function gets invoked. (Place: **stack**)

- **Global variables**: they are created once, automatically, when the program starts. (Place: **stack**)

- **Local variables**: they exist (accessible) only within their scopes. (Here the scope is the body of a function.) Locals are created dynamically when the control flow enters the scope where they were declared. (Place: **stack**)

- **All functions are global**: there are no local (nested) functions.

```c
int Max(int a, int b)
{                        // Function parameters
    if ( a > b )
        return a;
    else
        return b;
}
                         // Global variable
char* hello = "Hello";

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

int main()
{
    int x, y;            // Local variables
    input(&x,&y);
    return Max(x,y);
}
```

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
```

The C language compiler

A user invokes the compiler explicitly

An assembler

Assembling is a hidden action; it's performed silently

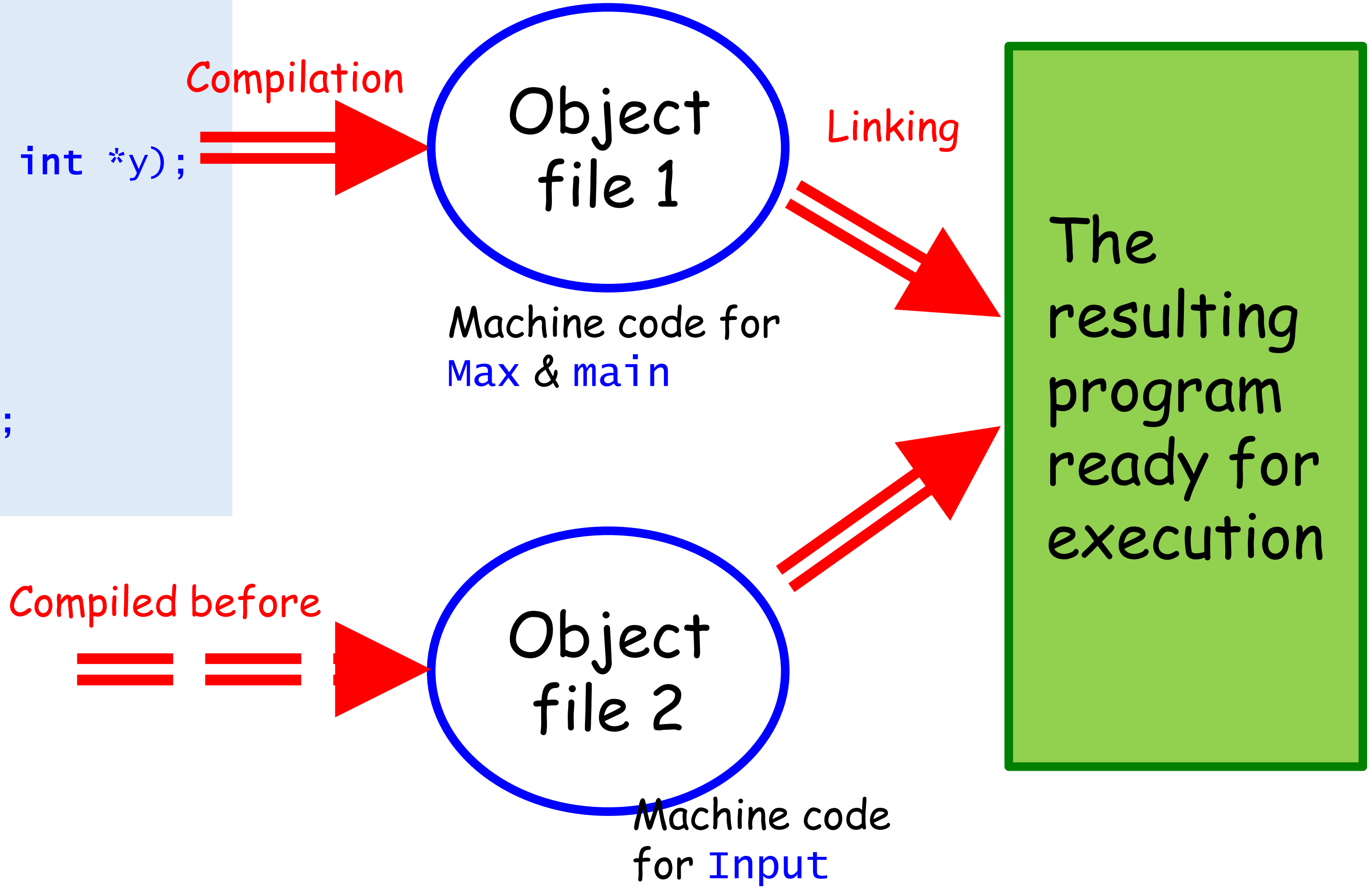
How C Programs are Built

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);
}
```



How C Programs are Built

Translation units and independent compilation

Max.c

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

Main.c

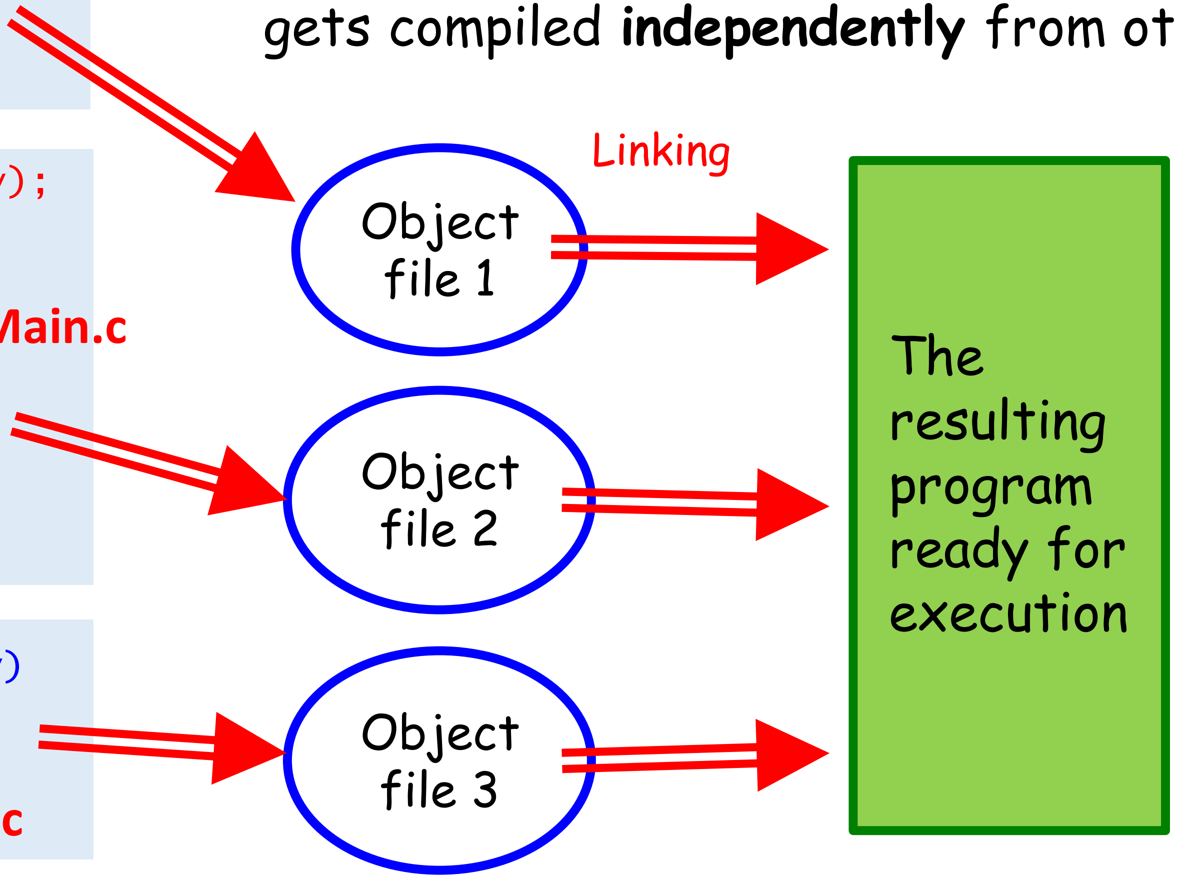
```c
void input(int* x, int *y);
int Max(int a, int b);

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

Input.c

```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 **independent compilation principle**; each TU gets compiled **independently** from others.



C Memory Management: Stack

Where are variables allocated?

- If declared outside a function, the are allocated in "static" storage
- If declared inside function, they are allocated on the "stack" and freed when function returns.

```
int aGlobal;

int main()
{
    int aLocal;
}
```

`aGlobal` is declared outside any function; it is the **global variable**

`aLocal` is declared within the function; it is the **local variable**

How the Stack Works

LIFO memory: "Last in –First out"

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

The rules

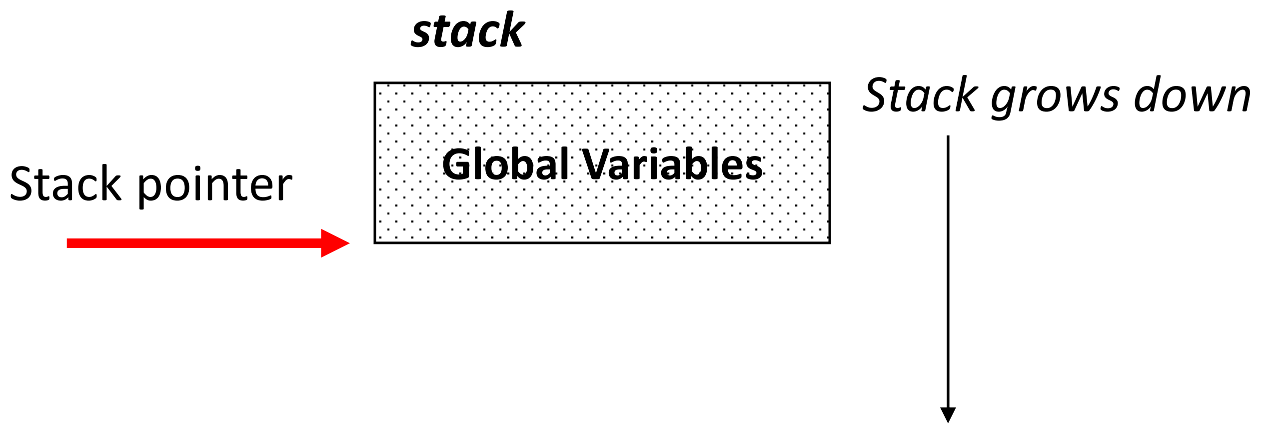
- Every time a function is called, **a new frame is allocated** on the stack.

  Activation record, or **Stackframe**

- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Space for local variables
- Stack frames are adjacent blocks of memory; **stack pointer** indicates the start of the stack frame.
- When function ends, the stack frame is popped off the stack; frees memory for future stack frames.

How the Stack Works

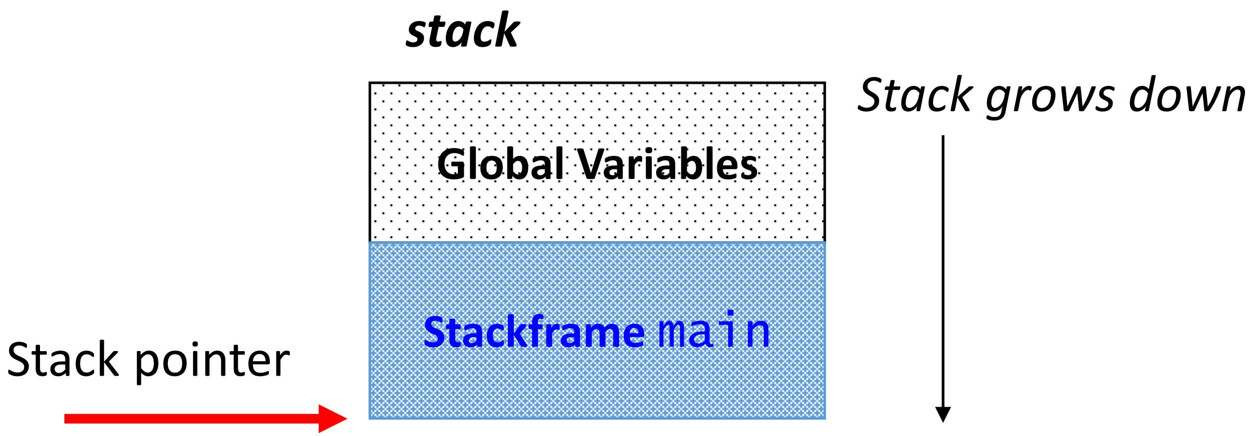
```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```



How the Stack Works

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

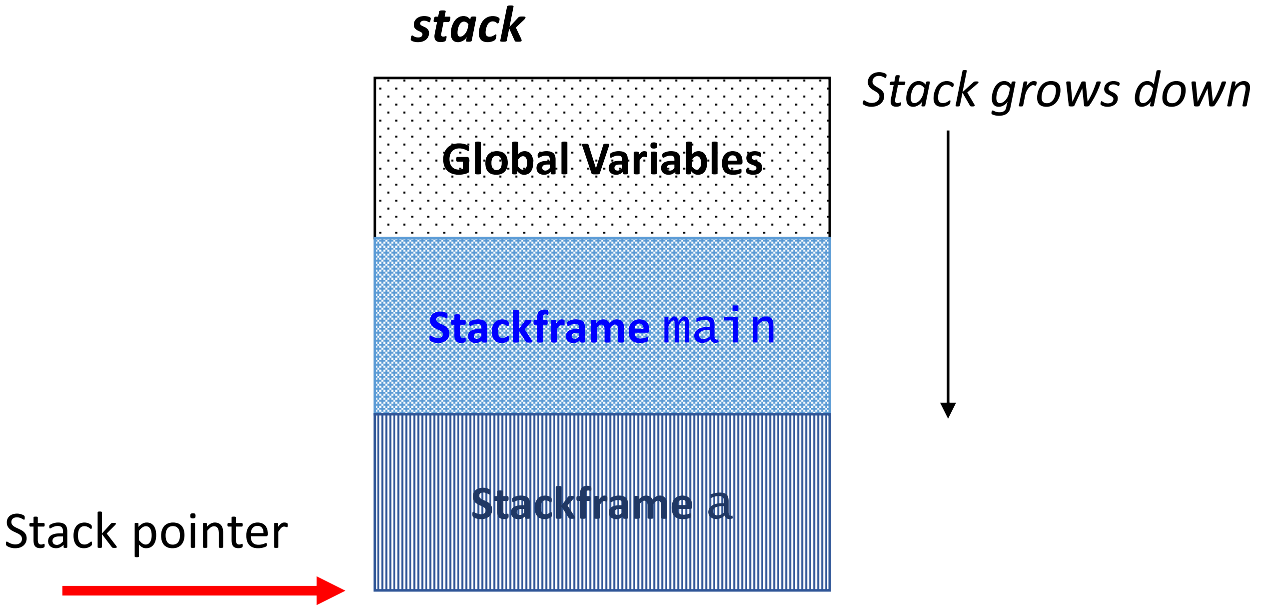
Call chain



How the Stack Works

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

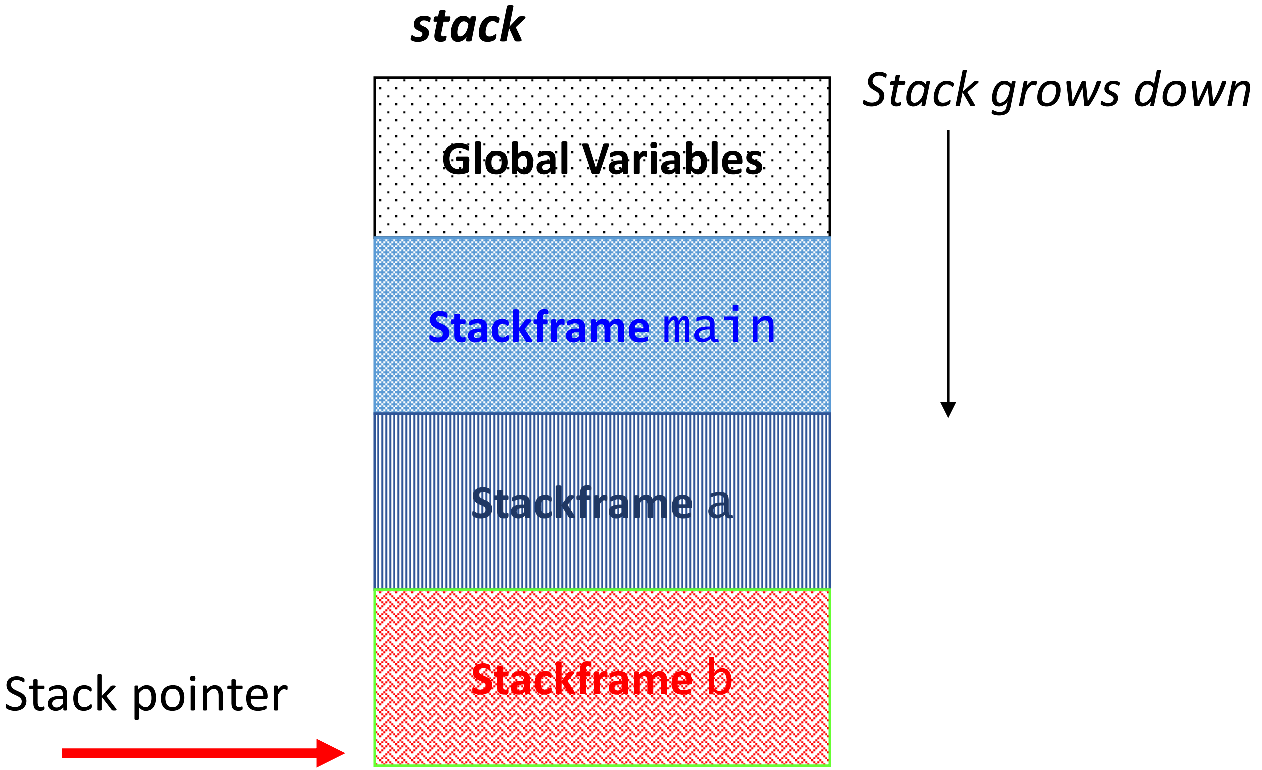
Call chain



How the Stack Works

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

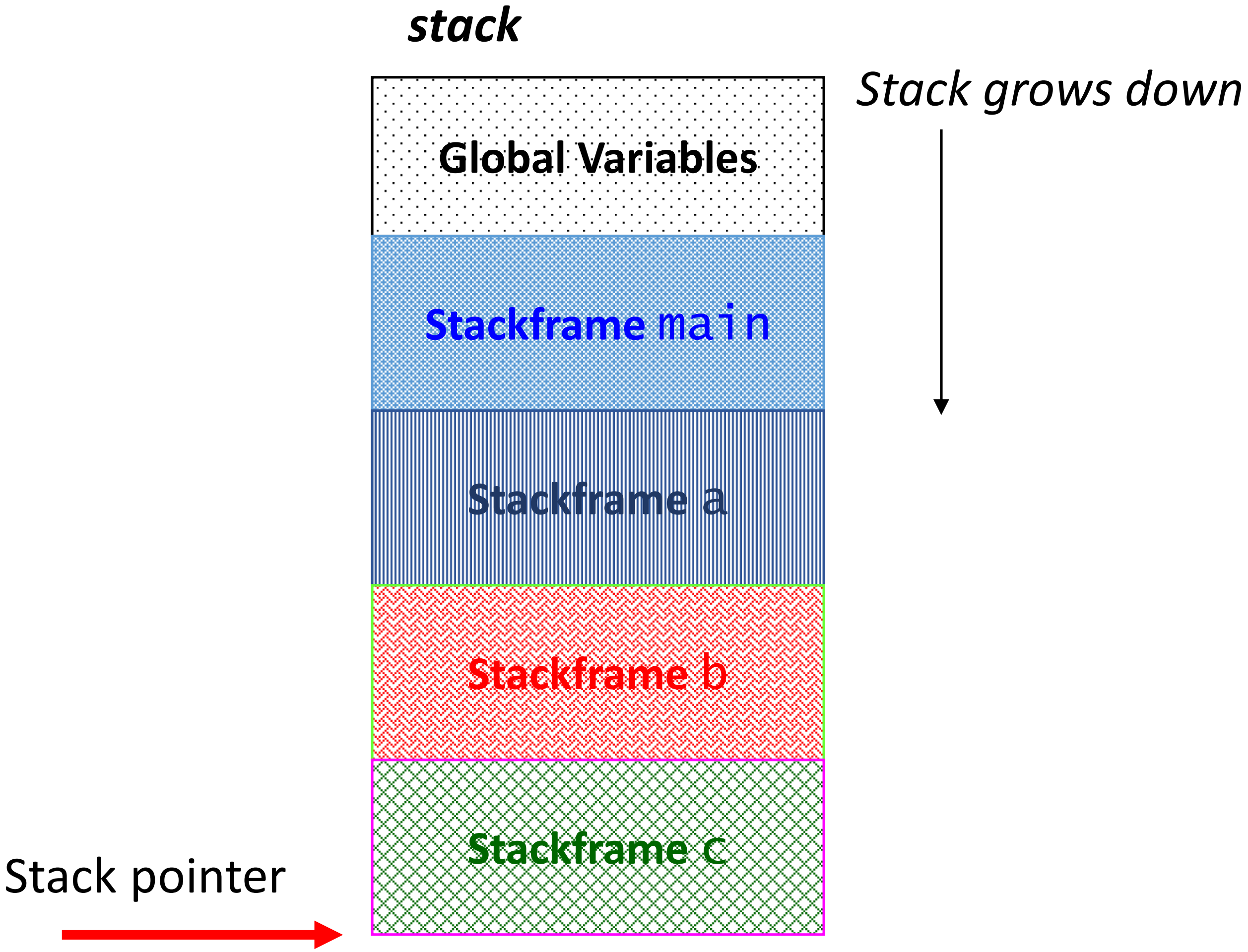
Call chain



How the Stack Works

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

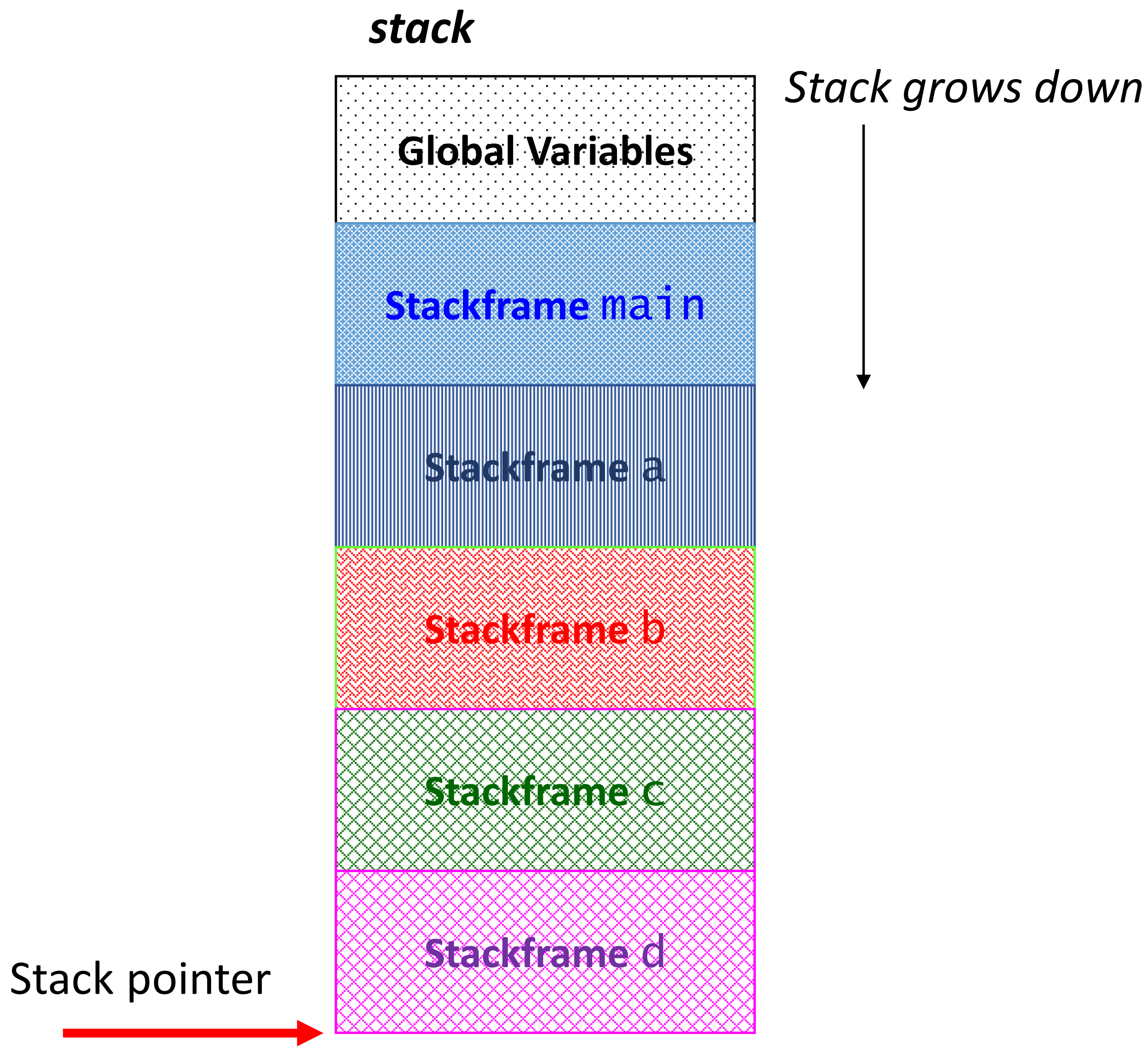
Call chain



How the Stack Works

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

Call chain

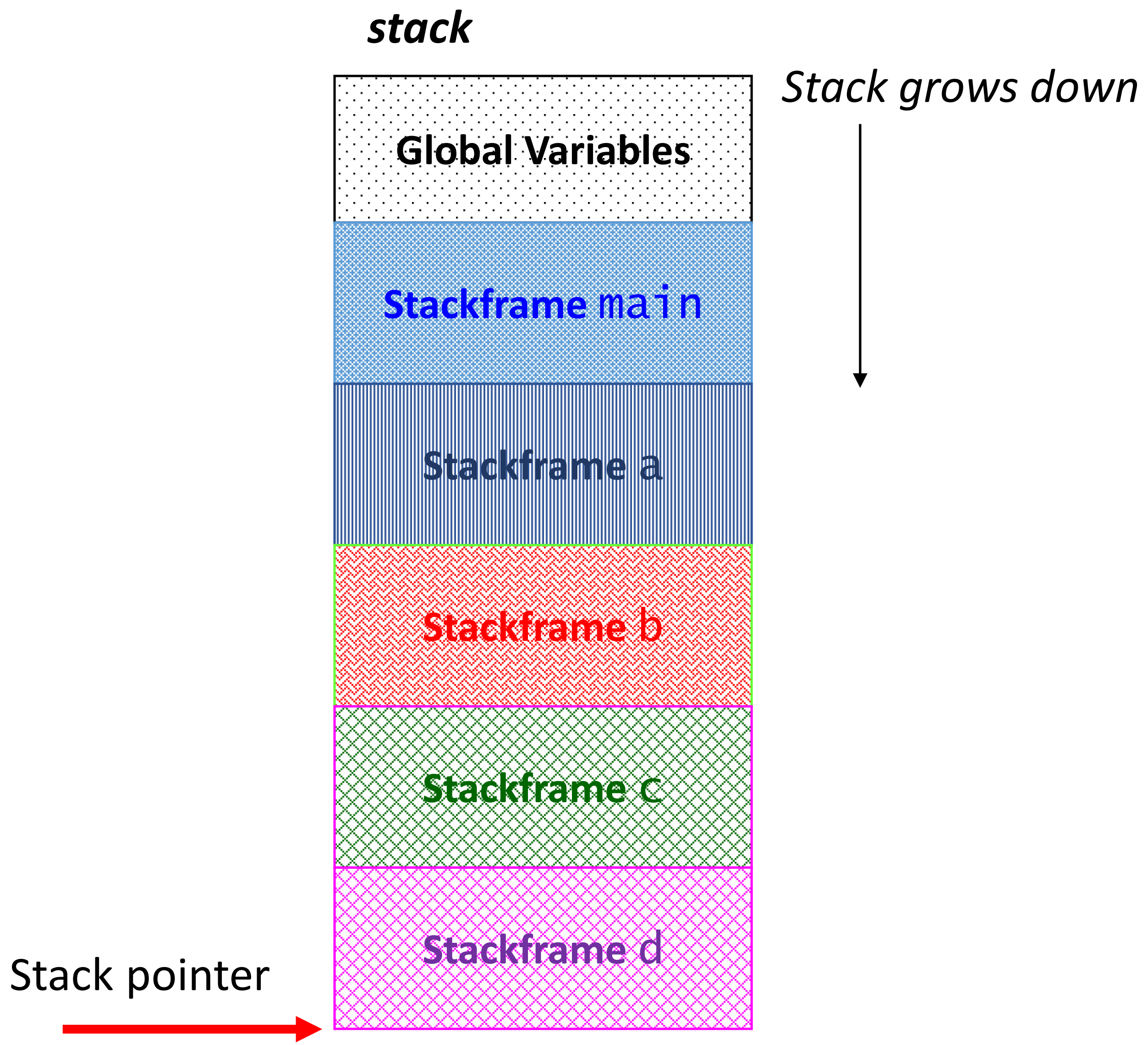


How the Stack Works

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

Return chain

Call chain



C Memory Management: Stack

We will continue considerations of the stack functionality in more details on **tutorial** today.

Scope of a Variable

- The **scope** of a variable is a portion of the (source) code in which that variable is visible
  - the scope is where in the code we can refer to the variable declared
- Scoping rules (of some language, e.g., C) define scopes of variables
- Scoping rules may vary from language to language and also among different declaration types in the same language
  - i.e. scoping rules for variable declarations may be different from those for function declarations

Blocks

- In most structured high-level languages the notion of **block** is central to scope identification
- A block is a portion of code enclosed between two special symbols, which mark the beginning and the end of the block.
  - In C (in Java, C++ etc.) blocks are marked by curly braces:

    { *this is a block* }

  - In some other languages blocks are marked by **begin** and **end** keywords or in some other manner (e.g. implicitly).
- Usually, blocks can be **nested**; but some language-dependent limitations are possible.

Scopes & Blocks

- Variable is visible
  - In the block it is defined
    - Starting from the line of definition
  - In all inner bocks unless a variable of the same name is declared within
- Global variables (if exist in the language)
  - Defined outside the scope of any block
- Hiding a variable
  - A homonymous variable declared within a block makes a variable of the same name declared outside <u>invisible</u>

Scopes & Blocks

- Scope is a **rule** determining existence and visibility of variables.
- Block is a compound language **construct** where variables (and other program entities) are declared.
- Declared entities are valid only within their scope, e.g. a variable exists only in its scope. The system is unaware of these entities in other parts of the code.

Scopes & Blocks: an Example

```
void f()
{
    int i = 3;
    for ( int j=0; j<20; j++ )
    {
        int k;
        if ( condition )
        {
            int i = 7;
            ...i+k...g(k)...
        }
        else
        {
            int j = g(k+i);
            ...
        }
    }
}
int g(int z) {
    int i = z+1;
    ...
    return i*i;
}
```

Scopes & Blocks: an Example

```
void f()
{
    int i = 3;
    for ( int j=0; j<20; j++ )
    {
        int k;
        if ( condition )
        {
            int i = 7;
            ...i+k...g(k)...
        }
        else
        {
            int j = g(k+i);
            ...
        }
    }
}
int g(int z) {
    int i = z+1;
    ...
    return i*i;
}
```

The scope of inner i is this block. The local i hides the i from the outer block

The scope of inner j is this block. The local j hides the j from the outer block

Scopes & Blocks: an Example

The loop body is the block. j and k are declared in the block that is the scope for them

The scope of inner i is this block. The local i <u>hides</u> the i from the outer block

The scope of inner j is this block. The local j <u>hides</u> the j from the outer block

```
void f()
{
    int i = 3;
    for ( int j=0; j<20; j++ )
    {
        int k;
        if ( condition )
        {
            int i = 7;
            ...i+k...g(k)...
        }
        else
        {
            int j = g(k+i);
            ...
        }
    }
}
int g(int z) {
    int i = z+1;
    ...
    return i*i;
}
```

Scopes & Blocks: an Example

Function body is the block

The scope of i starts from its declaration until the end of the block except inner scope where local i is declared

The loop body is the block. j and k are declared in the block that is the scope for them

The scope of inner i is this block. The local i hides the i from the outer block

The scope of inner j is this block. The local j hides the j from the outer block

Function body is the block. The scope for z and i is the body. g's i is not related to f's i.

```
void f()
{
    int i = 3;
    for ( int j=0; j<20; j++ )
    {
        int k;
        if ( condition )
        {
            int i = 7;
            ...i+k...g(k)...
        }
        else
        {
            int j = g(k+i);
            ...
        }
    }
}
int g(int z) {
    int i = z+1;
    ...
    return i*i;
}
```

Summary

- Personal introduction: you know me, I will learn who you are ☺.
- Course introduction: you know what and how we will be doing.
- Languages' syntax & semantics.
- Program lifecycle: compilation.
- The memory model: code, heap & stack.
- The typical C program structure.
- How C programs are compiled and built.
- C programs and the notion of stack.
- Variable scopes and program blocks.