Programming Software Systems

Introduction to Programming for the Computer Engineering Track

Tutorial 2

Eugene Zouev
Fall Semester 2020
Innopolis University

Static vs Dynamic

Static typing

C, C++, Java, Scala, C#, Eiffel, …

- ☹ Requires more efforts while writing a program: need to explicitly specify object types.
- ☺ **The program is (much) more safe**: many bugs are detected before running (in compile time).
- ☺ The program is more readable; it's easier to read, understand and maintain it.

```
int x;
...
x = 7; // OK
...
x = "string"; // error
```

The binding between the variable and its type is **hard**: x can take any value but the type of the value must be always the same.

Dynamic typing

Javascript, Python, Ruby, …

- ☺ It's much easier to write a program: no need to take care about object types.
- ☺ The program is more flexible: no need to introduce different objects for different purposes.
- ☹ The program often looks cryptic; it's required much more efforts to understand and maintain them.
- ☹ **Programs are unsafe and inefficient** .

The binding between the variable and its type is **soft**: x can hold any value of any type.

```
x = 7;          // OK
...
x = "string"; // OK!
...
y = x + 7;      // OK!
```

Static vs Dynamic: Pros & Cons

Why dynamic programs are less safe?

```
a * b
```

Static vs Dynamic: Pros & Cons

Why dynamic programs are less safe?

```
a * b
```

A static language:
the compiler checks types of a and b and concludes (before programs starts) whether multiplication operator is valid for those types.

If types are inappropriate compiler reports a diagnostic message, and the bug can be fixed before launching the program.

Static vs Dynamic: Pros & Cons

Why dynamic programs are less safe?

```
a * b
```

A static language:
the compiler checks types of a and b and concludes (before programs starts) whether multiplication operator is valid for those types.

If types are inappropriate compiler reports a diagnostic message, and the bug can be fixed before launching the program.

A dynamic language:
the compiler knows nothing about types of a and b therefore it cannot judge whether this code is valid of not.

Typically, compiler trusts programmer and considers the code valid.

For some values of a and b the code could be really valid, but for other could be erroneous.

The bug can occur only while program is running and sometimes after long time after it is written...

Static vs Dynamic: Pros & Cons

Why dynamic programs are less efficient?

```
a * b
```

Static vs Dynamic: Pros & Cons

Why dynamic programs are less efficient?

```
a * b
```

A static language:
the compiler knows for sure that a and b both have integer type.
So, it generates the compact and fast code:

```
•   Load integers a & b to the stack
•   Perform integer multiplication
•   Remove operands from the stack
    and put the result to the stack
```

Static vs Dynamic: Pros & Cons

Why dynamic programs are less efficient?

```
a * b
```

A static language:
the compiler knows for sure that a and b both have integer type.
So, it generates the compact and fast code:

- Load integers a & b to the stack
- Perform integer multiplication
- Remove operands from the stack and put the result to the stack

A dynamic language:
the compiler knows nothing about types of a and b therefore it should generate code for **all possible cases**:

```
if a, b are integers:
  Perform integer multiplication
else if a, b are strings:
  Perform string concatenation
else if a is string, b is integer:
  Create repetition like a+a+a
else if a is integer, b is string:
  Convert a to string
  Perform string concatenation
else ...
```

Types & Memory

- Each object is of some type - *obvious*
- Each object (value) occupies some amount of memory – *trivial*

=> What's the **size** of memory for (values of) various types?

Types & Memory

- Each object is of some type - *obvious*
- Each object (value) occupies some amount of memory – *trivial*

=> What's the **size** of memory for (values of) various types?

Different approaches are taken in programming languages

Java

All type sizes are defined in the language reference explicitly

(for example, type `int` occupies 32 bits)

C

- (A value of) `char` type occupies exactly one byte
- Other type sizes depend on underlying hardware (RAM/CPU)

Types & Memory

- Each object is of some type - *obvious*
- Each object (value) occupies some amount of memory – *trivial*

=> What's the **size** of memory for (values of) various types?

C

```
int si = sizeof (int);
int se = sizeof a+b;
```

Different approaches are taken in programming languages

Java

All type sizes are defined in the language reference explicitly
(for example, type `int` occupies 32 bits)

C

- (A value of) `char` type occupies exactly one byte
- Other type sizes depend on underlying hardware (RAM/CPU)

Types & Memory

- Each object is of some type - *obvious*
- Each object (value) occupies some amount of memory – *trivial*

=> What's the **size** of memory for (values of) various types?

C

sizeof is a unary operator

```
int si = sizeof (int);
int se = sizeof a+b;
```

- sizeof returns the amount of memory <u>in bytes</u>
- sizeof applies to any expression or to any type
- sizeof is a compile time construct

Different approaches are taken in programming languages

Java

All type sizes are defined in the language reference <u>explicitly</u>
(for example, type int occupies 32 bits)

C

- (A value of) char type occupies exactly <u>one byte</u>
- Other type sizes <u>depend on underlying hardware</u> (RAM/CPU)

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;`

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;`

C Derived ("User-Defined") Types

Some tricks & flaws with C types and declarations

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

Usual declaration of a structure type…

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

```c
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;`

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

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

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.

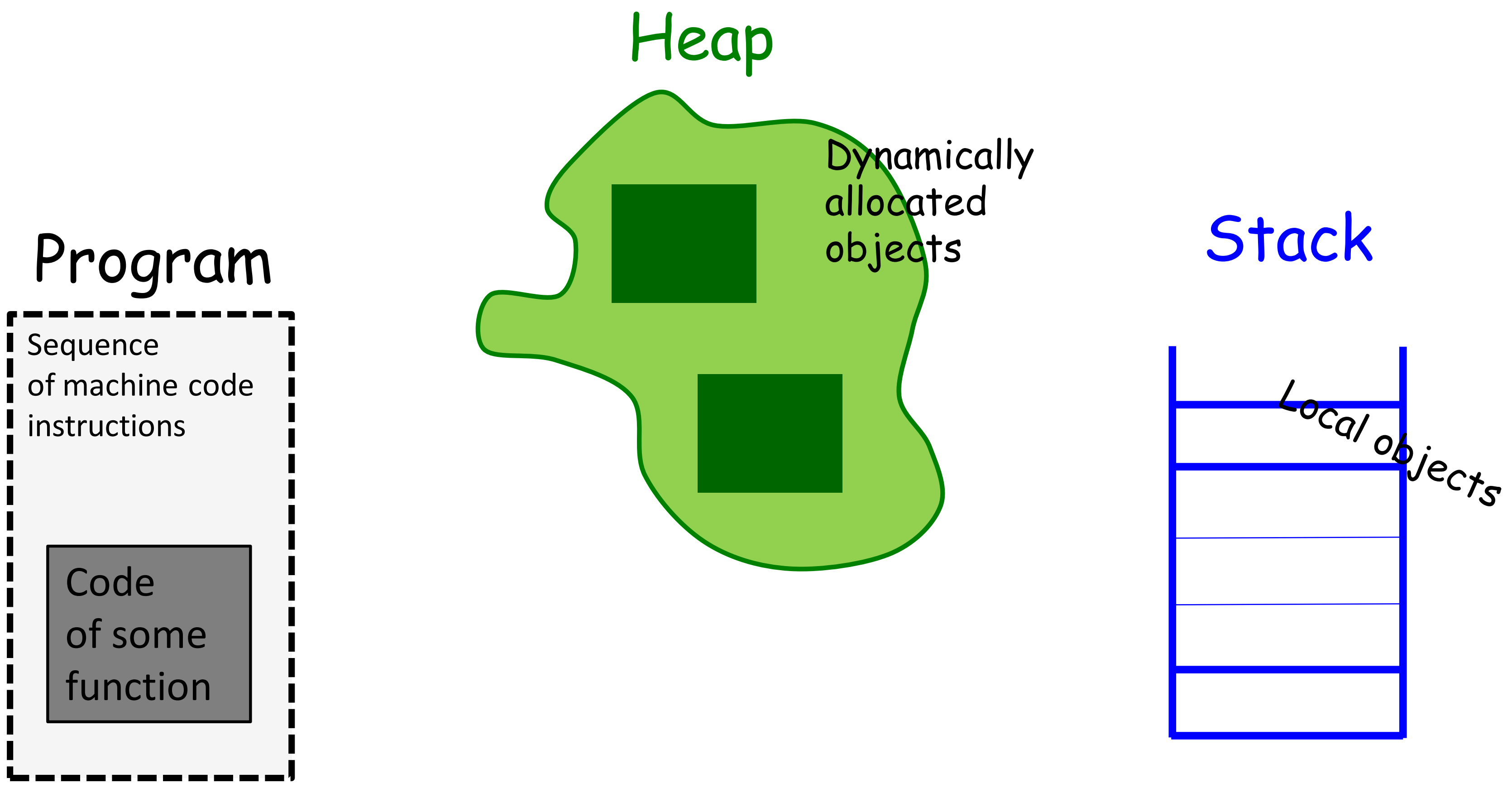
```
typedef struct {
    int a, b;
} S;
```

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

Later, we can use the synonym:

`S s1, s2;`

Pointers & The C Memory Model

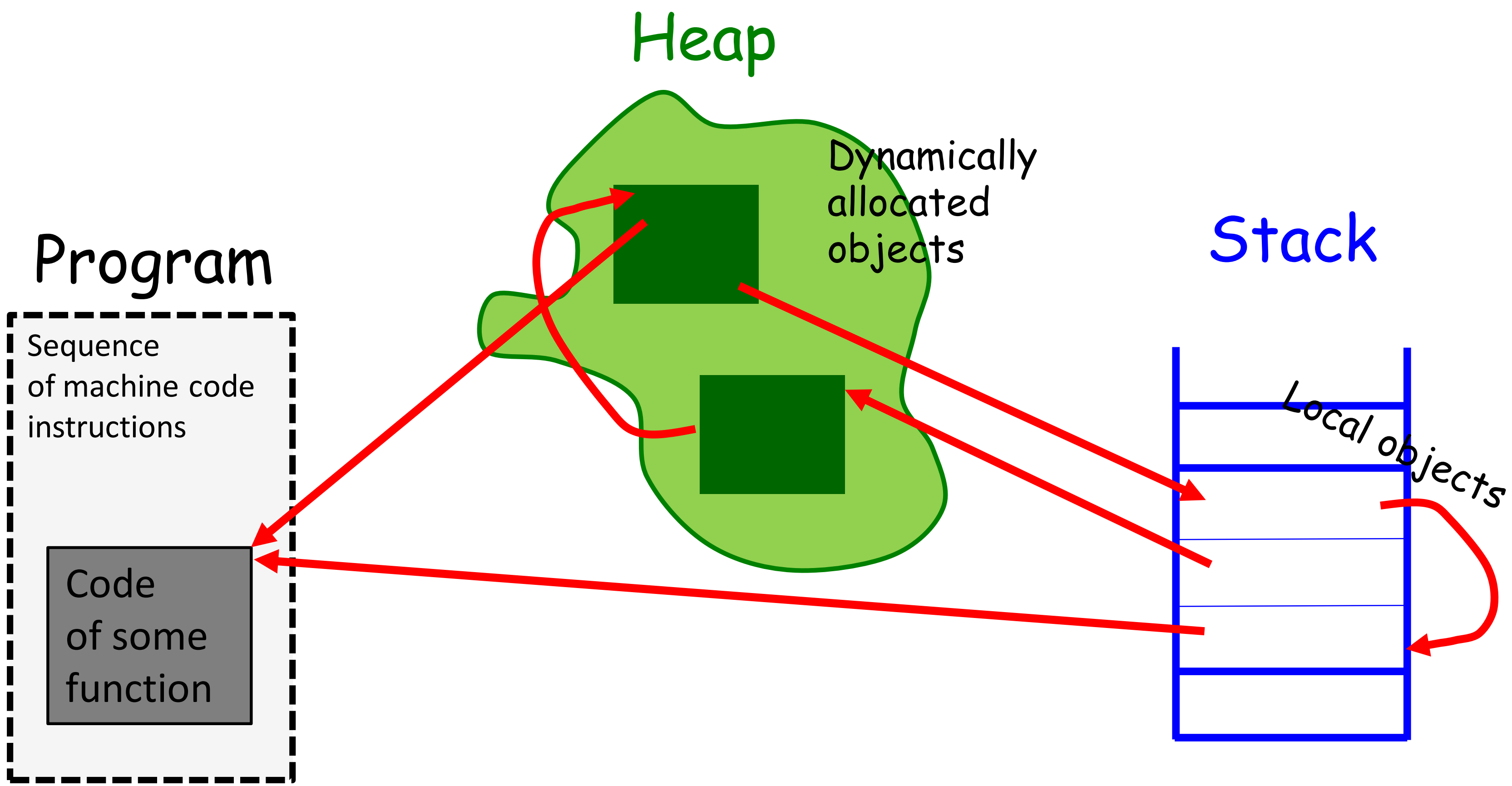


Program cannot modify this memory (self-modified programs are not allowed)

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

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

Pointers & The C Memory Model



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

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

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

Problems with C pointers

Example: pointers & scopes

```
int* p;

void f() {
    int A[10];
    p = A+2;
}

void main() {
    f();
    *p = 777;
}
```

p is the **global object**; it's created on the program's start and exist until its end

A is the **local object**; it's created on the f's start and exists **until exit from** f

What the hell will happen here?!

Problems with C pointers

Execution Stack

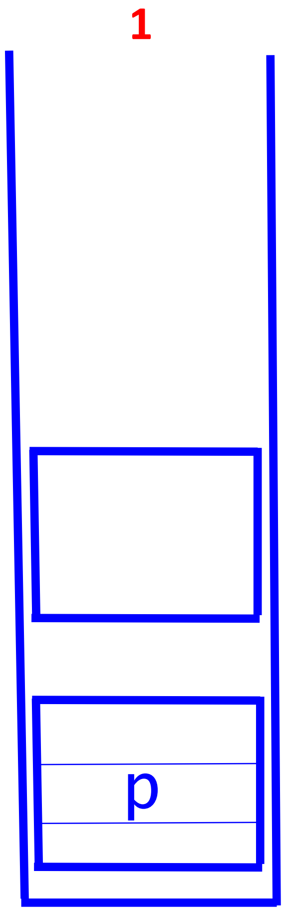
```
int* p;
void f() {          2
    int A[10];
    p = A+2;        3
}
void main() {       1
    f();
    *p = 777;       4
}
```

Problems with C pointers

```
int* p;
void f() {
    int A[10];
    p = A+2;
}
void main() {
    f();
    *p = 777;
}
```

2
3
1
4

Execution Stack

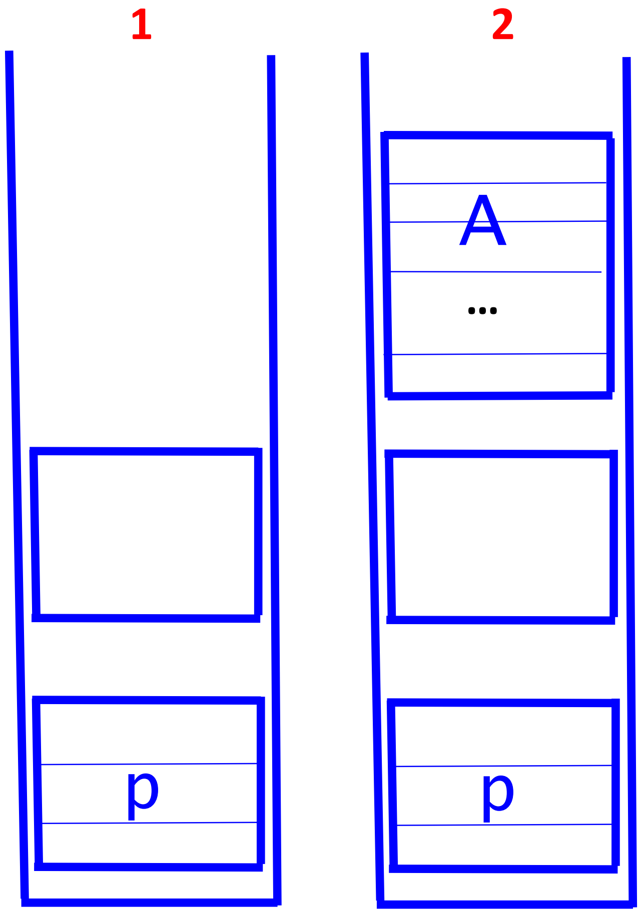


Stackframe for global variables

Problems with C pointers

```
int* p;
void f() {          2
    int A[10];
    p = A+2;        3
}
void main() {       1
    f();
    *p = 777;       4
}
```

Execution Stack



Stackframe for the call to f

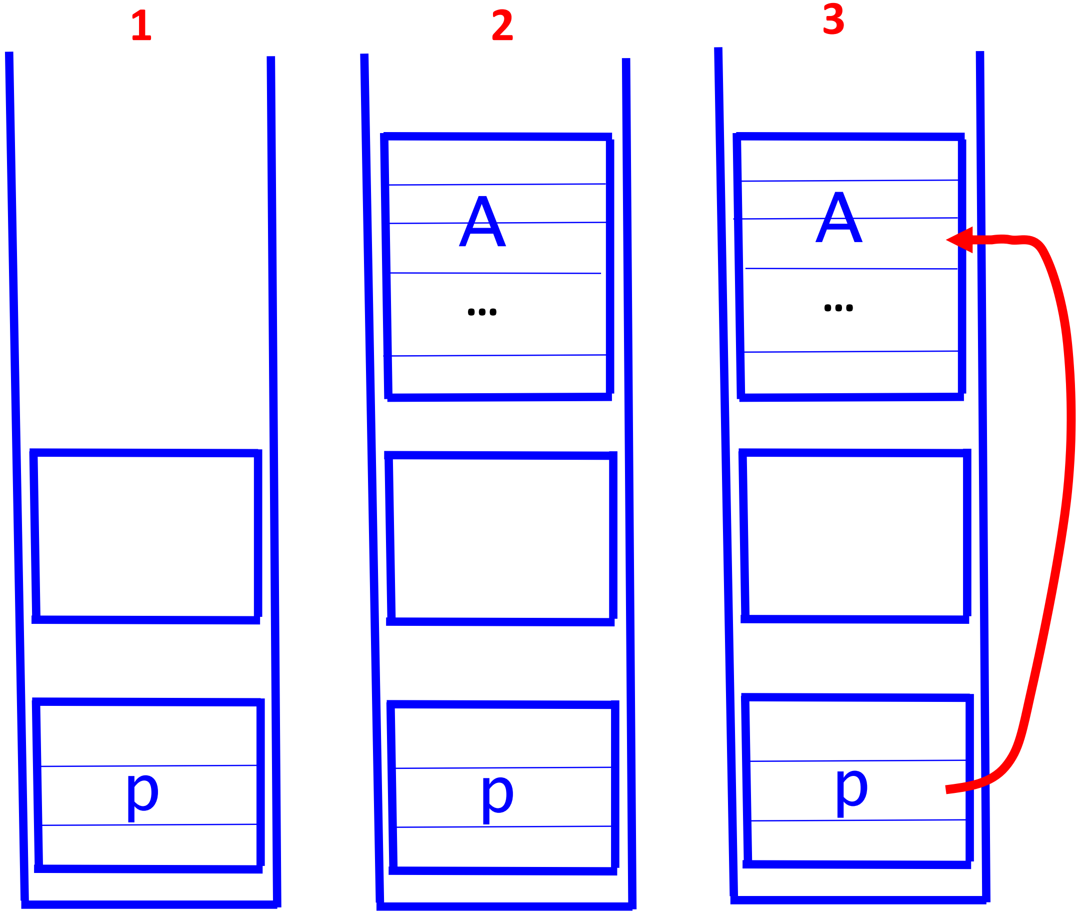
Stackframe for the call to main

Stackframe for global variables

Problems with C pointers

```
int* p;
void f() {
    int A[10];
    p = A+2;
}
void main() {
    f();
    *p = 777;
}
```

Execution Stack



Stackframe for the call to f

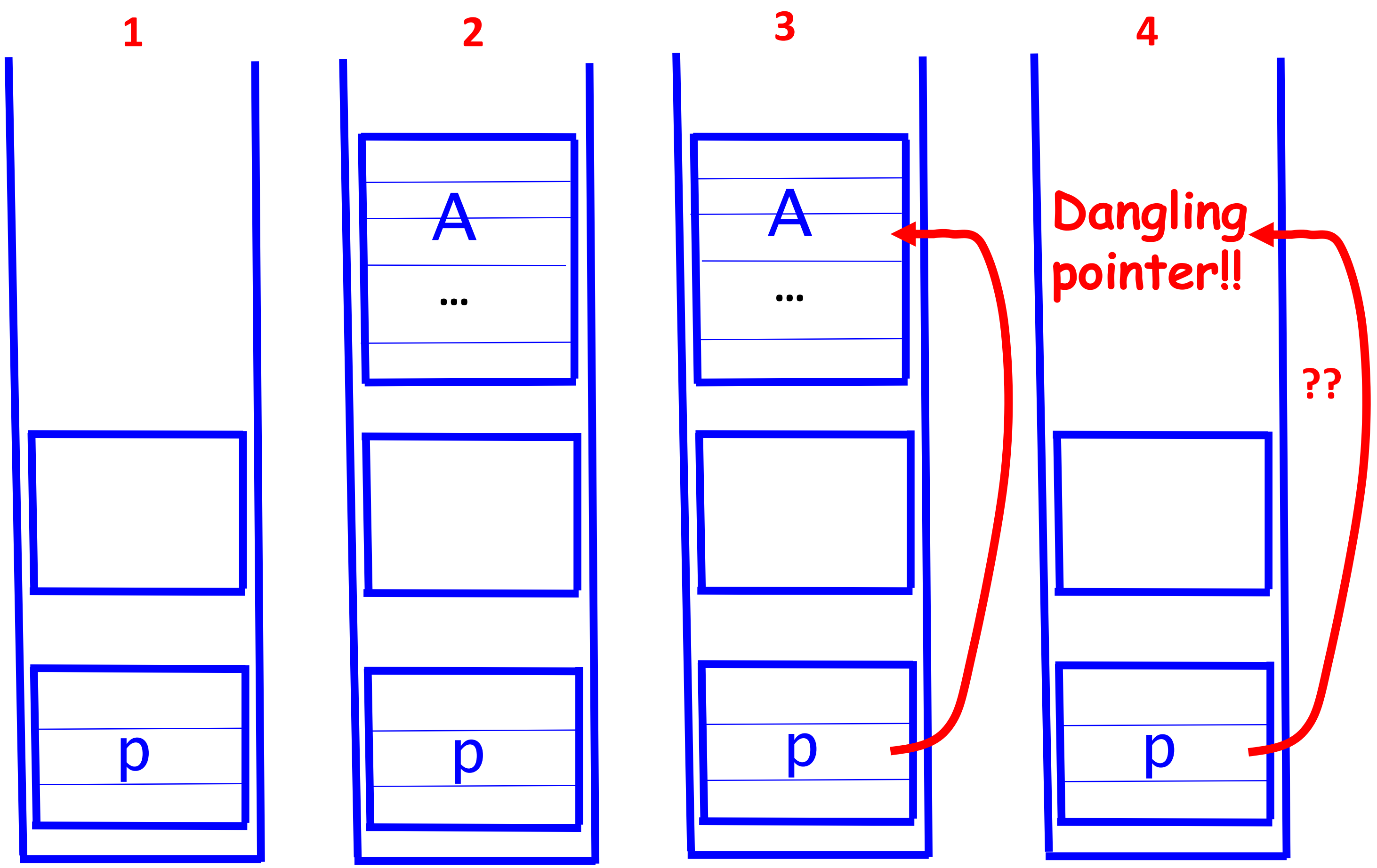
Stackframe for the call to main

Stackframe for global variables

Problems with C pointers

```c
int* p;
void f() {
    int A[10];
    p = A+2;
}
void main() {
    f();
    *p = 777;
}
```

Execution Stack



Stackframe for the call to f

Stackframe for the call to main

Stackframe for global variables

Problems with C pointers

Example: dynamic objects, pointers & scopes

```
void f() {
    int* p = (int*)malloc(10);
}

void main() {
    f();
    ...
}
```

p is the **local object**; it lives only within the f function

The unnamed object created by malloc() is **dynamic object**; it doesn't follow the scoping rules!

Problems with C pointers

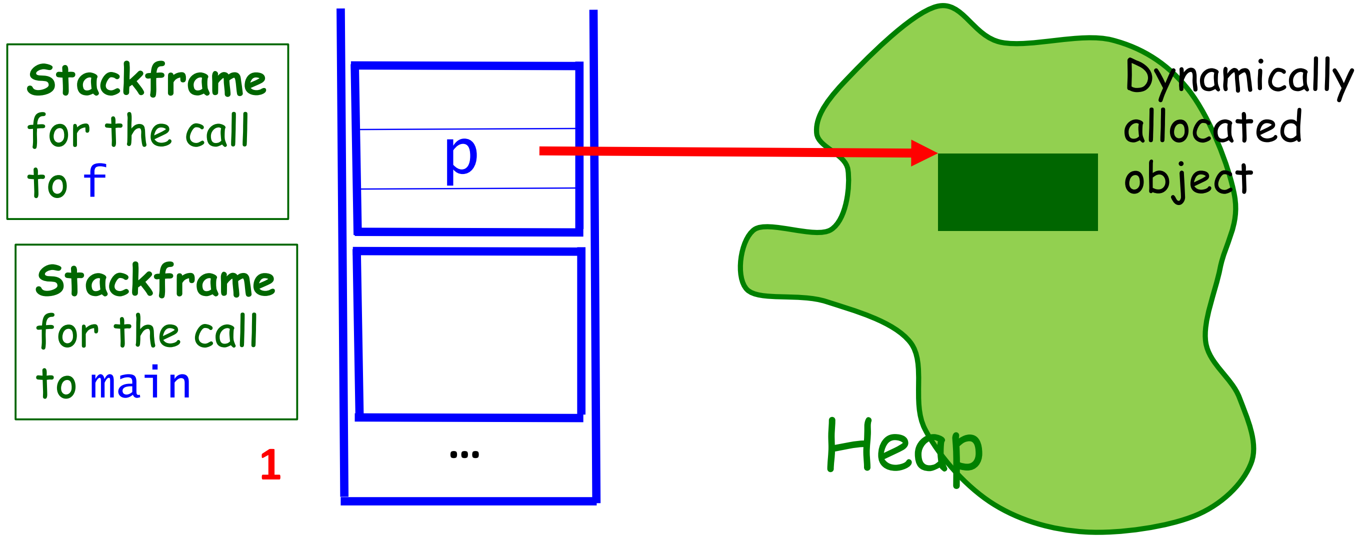
Example: dynamic objects, pointers & scopes

```
void f() {
   int* p = (int*)malloc(10);
}

void main() {
   f();
   ...
}
```

p is the **local object**; it lives only within the f function

The unnamed object created by malloc() is **dynamic object**; it doesn't follow the scoping rules!



Problems with C pointers

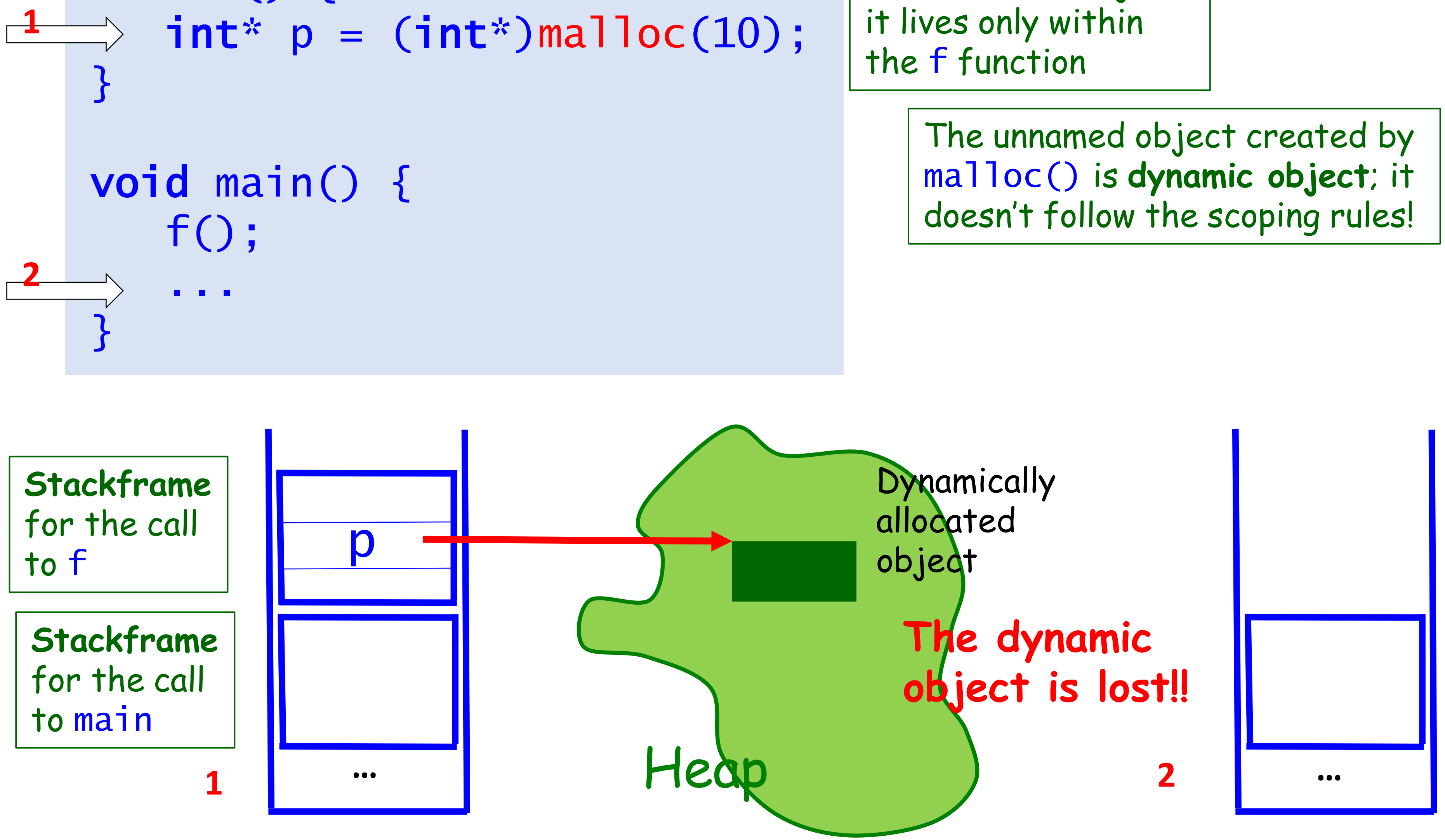
Example: dynamic objects, pointers & scopes

```
void f() {
    int* p = (int*)malloc(10);
}

void main() {
    f();
    ...
}
```

p is the **local object**; it lives only within the f function

The unnamed object created by malloc() is **dynamic object**; it doesn't follow the scoping rules!



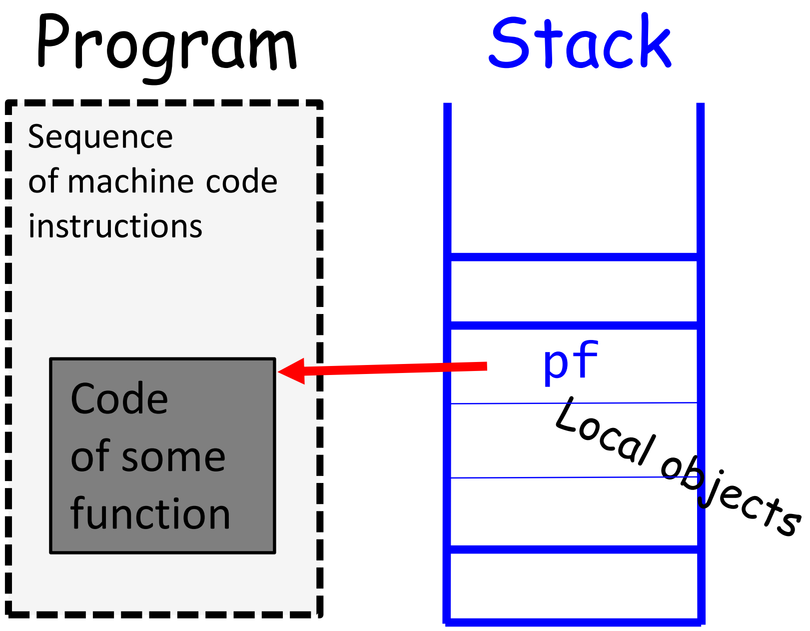
Pointers to Functions

```
void f(int p)
{
    ...
}
...

...
void main()
{
    f(1);  // call to f

}
```

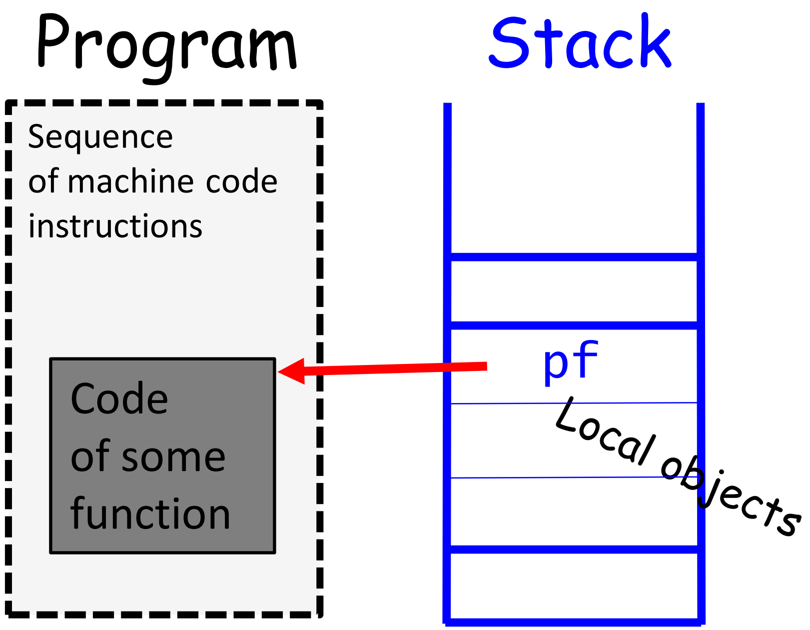
Pointers to Functions




```
void f(int p)
{
    ...
}
...
void (*pf)(int) = &f;
...
void main()
{
    f(1);  // call to f

}
```

Pointers to Functions




```
void f(int p)
{
    ...
}
...
void (*pf)(int) = &f;
...
void main()
{
    f(1);  // call to f
    pf(1); // call to f!
}
```

Automatic & static objects

```
void f() {
    int x = 0;
    ...
    x += 1;
    printf("%d", x);
}

void main() {
    for(int i=1; i<=100; i++)
        f();
}
```

x is the **local object**
- it “belongs” to the f function;
- it is **available** only from within the f function;
- it is created and gets initialized ***each time*** the f function is invoked

The loop prints the same value 1 on each iteration.

Automatic & static objects

```
void f() {
    int x = 0;
    ...
    x += 1;
    printf("%d", x);
}

void main() {
    for(int i=1; i<=100; i++)
        f();
}
```

x is the **local object**
- it "belongs" to the f function;
- it is **available** only from within the f function;
- it is created and gets initialized ***each time*** the f function is invoked

The loop prints the same value 1 on each iteration.

The x variable is often called as **automatic local variable.**

```
...
auto int x = 0;
...
```

Automatic & static objects

```
void f() {
    static int x = 0;
    ...
    x += 1;
    printf("%d", x);
}

void main() {
    for(int i=1; i<=100; i++)
        f();
}
```

x is still the **local object**
- it "belongs" to the f function;
- it is **available** only from within the f function;
- it is created and gets initialized **only once**: before the very first call to the function it belongs to.

The loop prints values 1,2,3,… on each iteration.

Automatic & static objects

```
void f() {
    static int x = 0;
    ...
    x += 1;
    printf("%d", x);
}

void main() {
    for(int i=1; i<=100; i++)
        f();
}
```

x is still the **local object**
- it "belongs" to the f function;
- it is **available** only from within the f function;
- it is created and gets initialized **only once**: before the very first call to the function it belongs to.

The loop prints values 1,2,3,… on each iteration.

The x variable is often called as **static local variable.**

Algol-60:
Own variables

Automatic & static objects

Example: Fibonacci numbers

Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-2) + Fib(n-1)

Automatic & static objects

Example: Fibonacci numbers

Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-2) + Fib(n-1)

```
long long Fib(int N)
{
    if ( N == 0 || N == 1 ) return N;
    return Fib(N-2) + Fib(N-1);
}
```

Automatic & static objects

Example: Fibonacci numbers

Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-2) + Fib(n-1)

```
long long Fib(int N)
{
    if ( N == 0 || N == 1 ) return N;
    return Fib(N-2) + Fib(N-1);
}
```

```
long long Fib() {
    static long long first = 0;
    static long long second = 1;
    long long out = first + second;
    first = second;
    second = out;
    return out;
}
```

Automatic & static objects

Example: Fibonacci numbers

Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-2) + Fib(n-1)

```
long long Fib(int N)
{
    if ( N == 0 || N == 1 ) return N;
    return Fib(N-2) + Fib(N-1);
}
```

The stateless function

The function with its own state, OR

<u>Finite automat</u>

```
long long Fib() {
    static long long first = 0;
    static long long second = 1;
    long long out = first + second;
    first = second;
    second = out;
    return out;
}
```