Introduction to Programming

Part I

Lecture 2
Introduction: Some Basic C Notions

Eugene Zouev
Fall Semester 2021
Innopolis University

Last Friday:

- Program lifecycle: compilation.
- The typical C program structure.
- How C programs are compiled and built.
- The memory model: code, heap & stack.
- C programs and the notion of stack.
- Variable scopes and program blocks.

Last Friday

How C Programs are Built

Translation units and independent compilation

Translation unit 1

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

Max.c

Translation unit 2

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

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

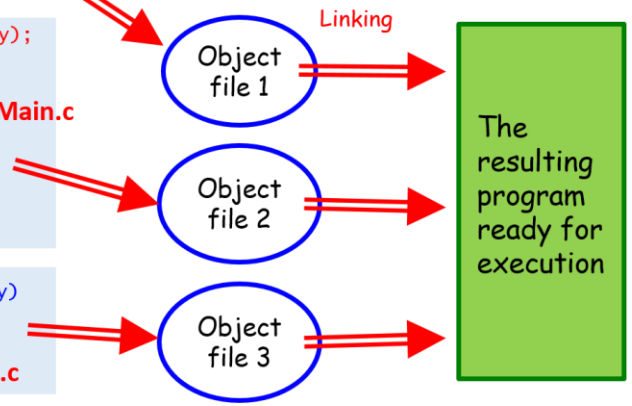
Main.c

Translation unit 3

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

Input.c

- 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.



12/24

Last Friday

How C Programs are Built

Translation units and independent compilation

Translation unit 1

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

Max.c

Translation unit 2

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

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

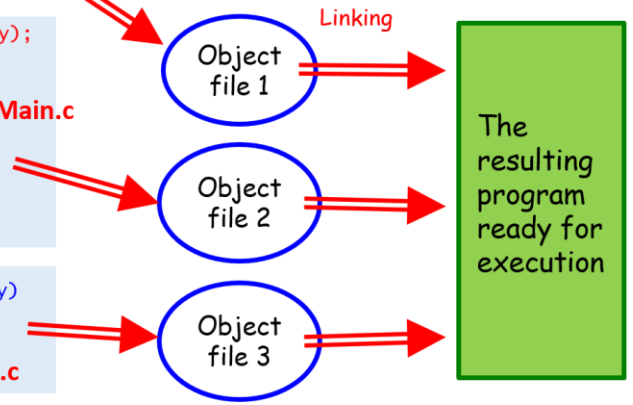
Main.c

Translation unit 3

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

Input.c

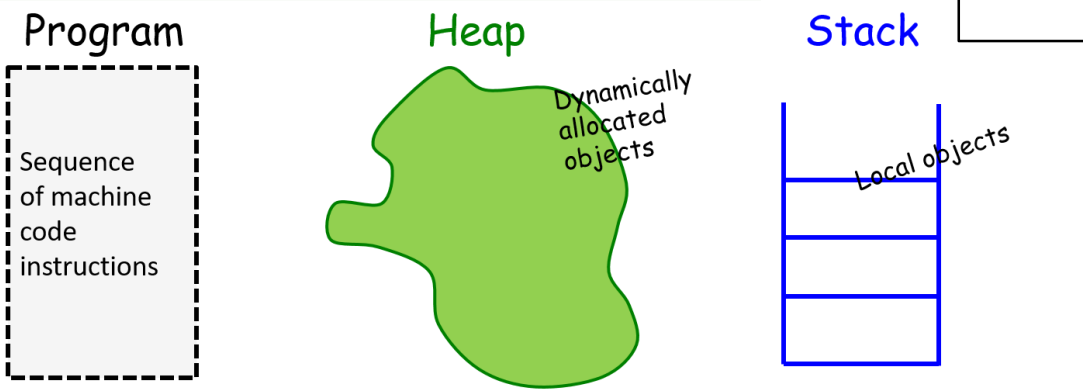
- Typically, any C program consists of several **translation units** each of which is located in a separate source file.
- The **independent compilation principle**; each TU gets compiled **independently** from others.



12/24

Each program uses three kinds of memory:

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

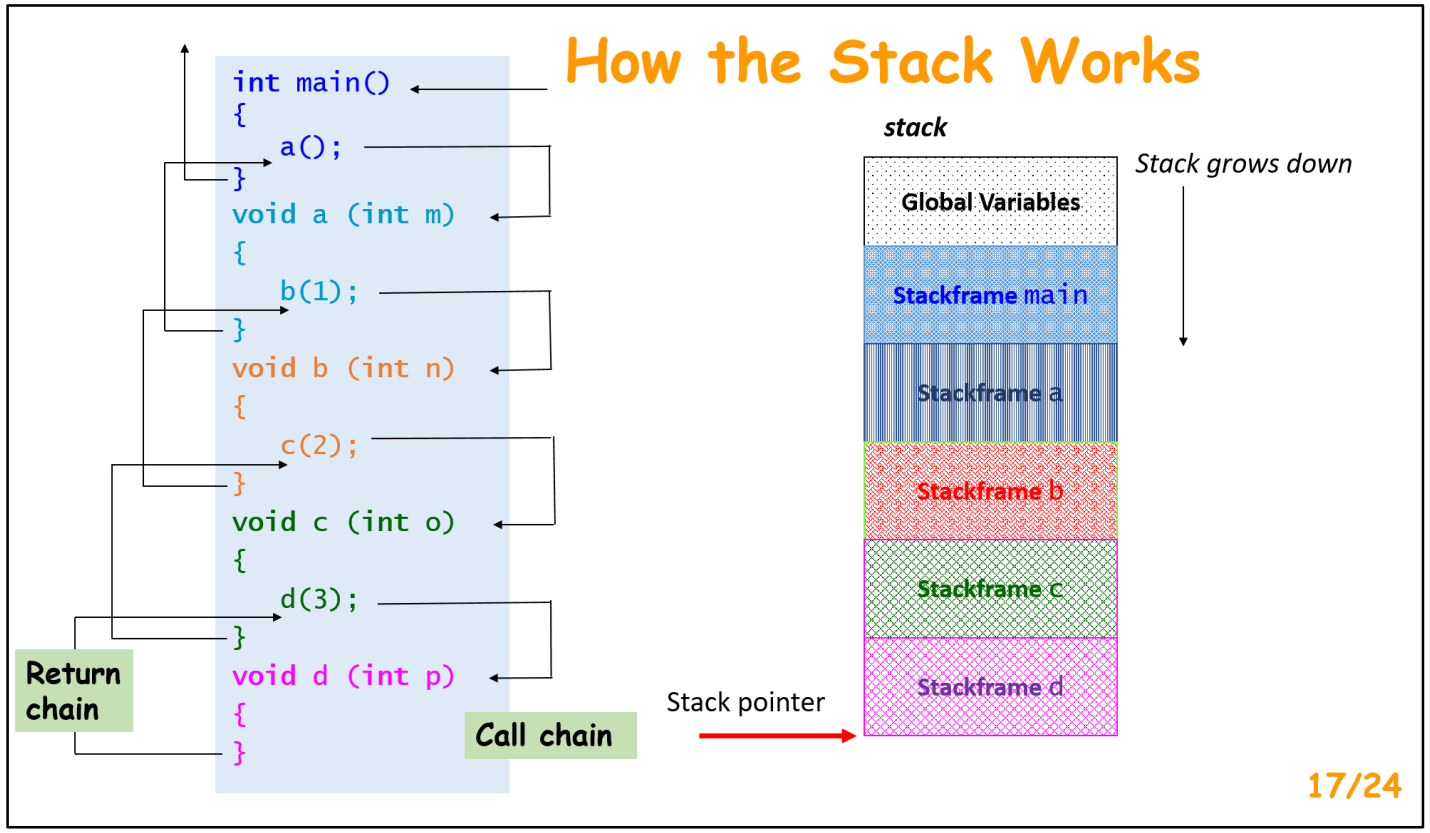


Program cannot modify this memory: self-modified programs are not allowed

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**

Last Friday

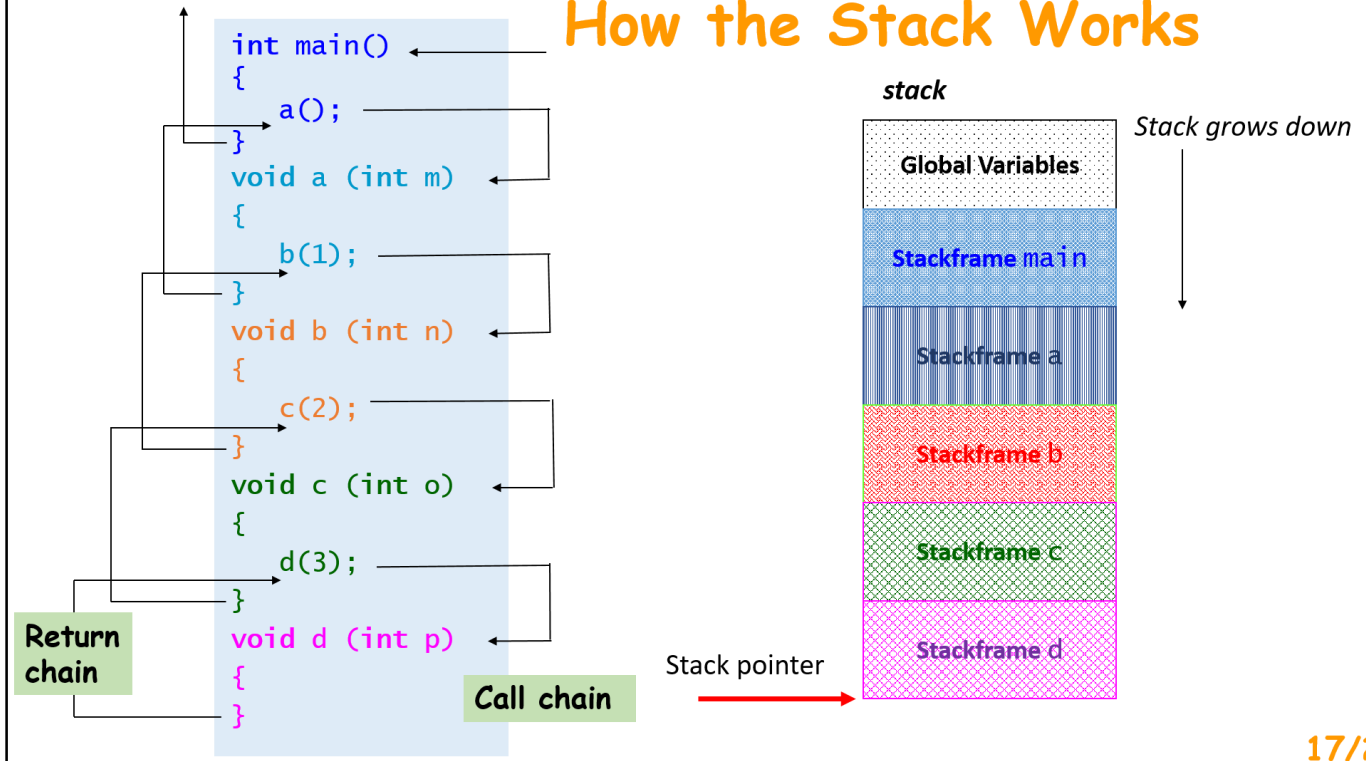

How the Stack Works
int main()
{
a();
}
void a (int m)
{
b(1);
}
void b (int n)
{
c(2);
}
void c (int o)
{
d(3);
}
void d (int p)
{
}
Return chain
Call chain
stack
Stack grows down
Global Variables
Stackframe main
Stackframe a
Stackframe b
Stackframe c
Stackframe d
Stack pointer
17/24

Last Friday

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.

How the Stack Works



17/24

Outline: Today

- The notion of **type**.
- Static and dynamic typing.
- Type categories.
- Storage class specifiers
- C type system: predefined & user-defined types.
- **Pointers & arrays**

The Notion of Type

Evolution of the Notion of Type (1)

Algol-60, Pascal, C:

Imperative programming

Predefined & user-defined data structures

Clu, Modula-2, Ada-83:

Abstract data types

+Data encapsulation with access control

C++, Ada-95, Eiffel & many followers:

Classes

+Inheritance & polymorphism

Evolution of the Notion of Type (2)

Type (of an object/entity) is:

Evolution of the Notion of Type (2)

Type (of an object/entity) is:

- A set of **values** that an object of the type can have

Evolution of the Notion of Type (2)

Type (of an object/entity) is:

- A set of **values** that an object of the type can have
- A set of **operators** on objects of that type

Evolution of the Notion of Type (2)

Type (of an object/entity) is:

- A set of **values** that an object of the type can have
- A set of **operators** on objects of that type
- A set or **relationships** between the type and other types

Evolution of the Notion of Type (3)

Type (of an object/entity) is:

- A set of **values** that an object of the type can have
- A set of **operators** on objects of that type
- A set of **relationships** between the type and other types

```
int i;
```

Evolution of the Notion of Type (3)

Type (of an object/entity) is:

- A set of **values** that an object of the type can have
- A set of **operators** on objects of that type
- A set of **relationships** between the type and other types

```
int i;
```

The set of **values**:
- Integer numbers within the range …

The set of (predefined) **operators**:
- Creation, destruction, copying, moving
- Arithmetic & comparison operators;
- Shifts; …

The set of (predefined) **relationships**:
- Conversions to boolean, float, …

Evolution of the Notion of Type (3)

Type (of an object/entity) is:

- A set of **values** that an object of the type can have
- A set of **operators** on objects of that type
- A set of **relationships** between the type and other types

```
struct S { ... };
```

```
int i;
```

The set of **values**:
- Integer numbers within the range …

The set of (predefined) **operators**:
- Creation, destruction, copying, moving
- Arithmetic & comparison operators;
- Shifts; …

The set of (predefined) **relationships**:
- Conversions to boolean, float, …

Evolution of the Notion of Type (3)

Type (of an object/entity) is:

- A set of **values** that an object of the type can have
- A set of **operators** on objects of that type
- A set of **relationships** between the type and other types

```
int i;
```

The set of **values**:
- Integer numbers within the range …

The set of (predefined) **operators**:
- Creation, destruction, copying, moving
- Arithmetic & comparison operators;
- Shifts; …

The set of (predefined) **relationships**:
- Conversions to boolean, float, …

```
struct S { ... };
```

The set of **values**:
- Cartesian product(*) of struct members' sets

The set of **operators**:
- Creation, destruction, copying, ~~moving~~
- Access to struct members ("fields")
- ~~User-defined operators~~

~~The set of **relationships**:~~
- ~~Between this type and its base class(es)~~
- ~~User-defined conversion operators~~

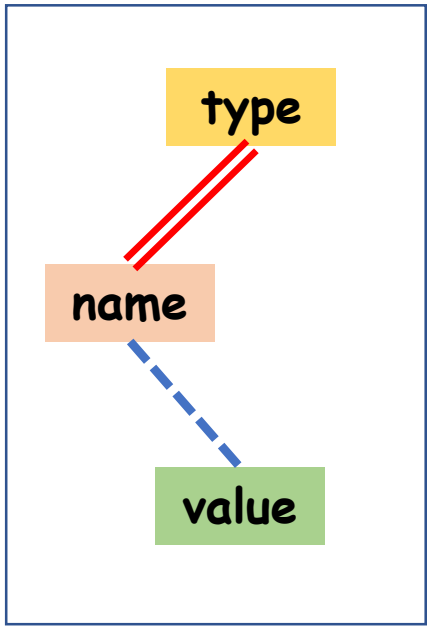
C++ only,
but not C…

(*) Cartesian product
Декартово произведение

Static vs Dynamic: Pros & Cons

Static typing

C, C++, Java, Scala, C#, Eiffel, ...




```
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.

Static vs Dynamic: Pros & Cons

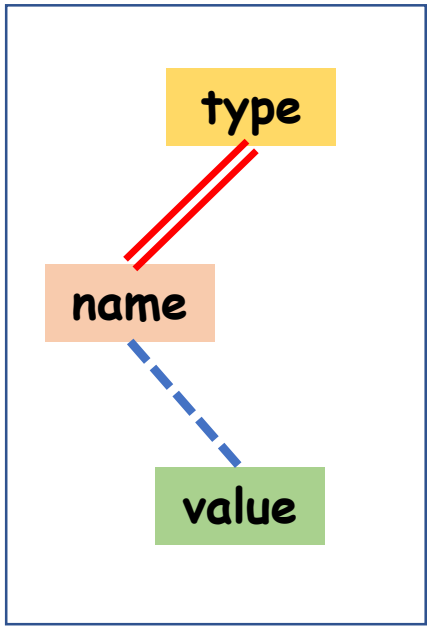
Static typing

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.



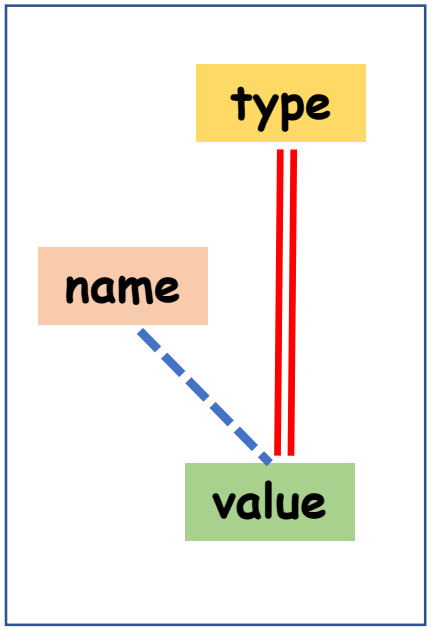

```
int x;
...
x = 7; // OK
...
x = "string"; // error
```

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.

Static vs Dynamic: Pros & Cons

Dynamic typing

Javascript, Python, Ruby, …




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

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

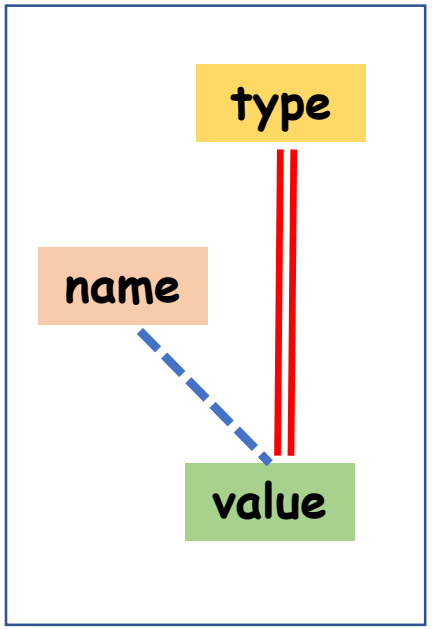
Formally correct, but what the hell does it mean??

Static vs Dynamic: Pros & Cons

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** .




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

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

Formally correct, but what the hell does it mean??

Static vs Dynamic: Pros & Cons

Dynamic programs are less safe

A point for discussion

Dynamic programs are less efficient

Type Categories

Types:

- Fundamental (atomic) `int` `char` `long double`
- Structured (compound) `int[10]`

- Predefined (language-defined)
- User-defined `struct` `class`

B. Stroustrup:
Class is a type

C Standard (Predefined) Types

```
char
```

```
_Bool
```

See tutorial for the low-level view on types

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

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)

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;
```

Storage Class Specifiers

```
auto
static
extern
```

Are introduced together with type specifiers in object declarations

Storage Class Specifiers

auto
static
extern

Are introduced together with type specifiers in object declarations

```
int a;
static char b;
extern float c;

void f()
{
    double d;
    static int e;
    auto int f;
}
```

Storage Class Specifiers

auto
static
extern

Are introduced together with type specifiers in object declarations

a is the **global non-static object**
- it "belongs" to the whole program;
- it is available throughout the program;
- it is created only once: before the program starts.

```
int a;
static char b;
extern float c;

void f()
{
    double d;
    static int e;
    auto int f;
}
```

Storage Class Specifiers

auto
static
extern

Are introduced together with type specifiers in object declarations

```
int a;
static char b;
extern float c;

void f()
{
    double d;
    static int e;
    auto int f;
}
```

a is the **global non-static object**
- it "belongs" to the whole program;
- it is available throughout the program;
- it is created only once: before the program starts.

b is the **global static object**
- it "belongs" to the whole program;
- it is available **only from within the translation unit** it belongs to;
- it is created only once: before the program starts.

Storage Class Specifiers

auto
static
extern

Are introduced together with type specifiers in object declarations

```
int a;
static char b;
extern float c;

void f()
{
    double d;
    static int e;
    auto int f;
}
```

a is the **global non-static object**
- it "belongs" to the whole program;
- it is available throughout the program;
- it is created only once: before the program starts.

b is the **global static object**
- it "belongs" to the whole program;
- it is available **only from within the translation unit** it belongs to;
- it is created only once: before the program starts.

c is the **global external object**
- this is not a definition but declaration; it's assumed that the object is (really) defined in some other translation unit;
- The memory for the object is not allocated here but in other TU.

Storage Class Specifiers

`auto`
`static`
`extern`

Are introduced together with type specifiers in object declarations

```
int a;
static char b;
extern float c;

void f()
{
    double d;
    static int e;
    auto int f;
}
```

`a` is the **global non-static object**
- it "belongs" to the whole program;
- it is available throughout the program;
- it is created only once: before the program starts.

`b` is the **global static object**
- it "belongs" to the whole program;
- it is available **only from within the translation unit** it belongs to;
- it is created only once: before the program starts.

`c` is the **global external object**
- this is not a definition but declaration; it's assumed that the object is (really) defined in some other translation unit;
- The memory for the object is not allocated here but in other TU.

`d` and `f` are **automatic local objects**
- they "belong" to the function in which they are declared;
- they are available only from within the function (i.e., they are local to the function);
- it's created each time the function is invoked.

Storage Class Specifiers

auto
static
extern

Are introduced together with type specifiers in object declarations

```
int a;
static char b;
extern float c;

void f()
{
    double d;
    static int e;
    auto int f;
}
```

a is the **global non-static object**
- it "belongs" to the whole program;
- it is available throughout the program;
- it is created only once: before the program starts.

b is the **global static object**
- it "belongs" to the whole program;
- it is available **only from within the translation unit** it belongs to;
- it is created only once: before the program starts.

c is the **global external object**
- this is not a definition but declaration; it's assumed that the object is (really) defined in some other translation unit;
- The memory for the object is not allocated here but in other TU.

d and f are **automatic local objects**
- they "belong" to the function in which they are declared;
- they are available only from within the function (i.e., they are local to the function);
- they are created each time the function is invoked.

e is the **local static object**
- it "belongs" to the function in which it's created;
- it is available only from within the function;
- it is created only once: before the program starts.

Pointers

1. **Pointer**:
   An object containing an address to some other object

```
int x;
int* p;
...
p = &x;
```

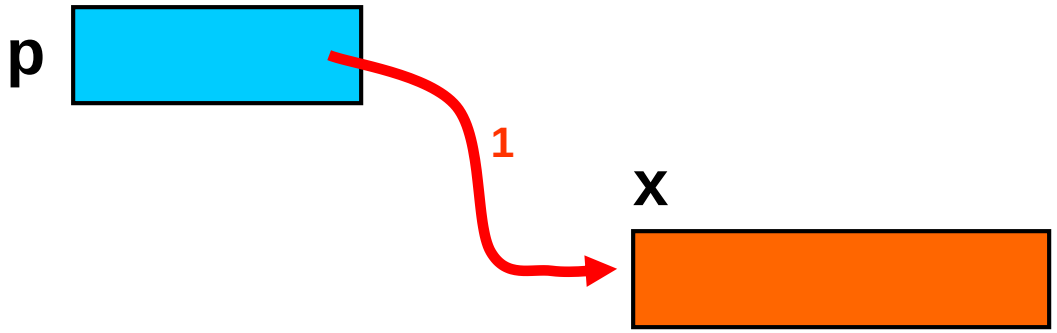
Unary "address-of" operator

Pointers

1. **Pointer**:
   An object containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```

Unary "address-of" operator

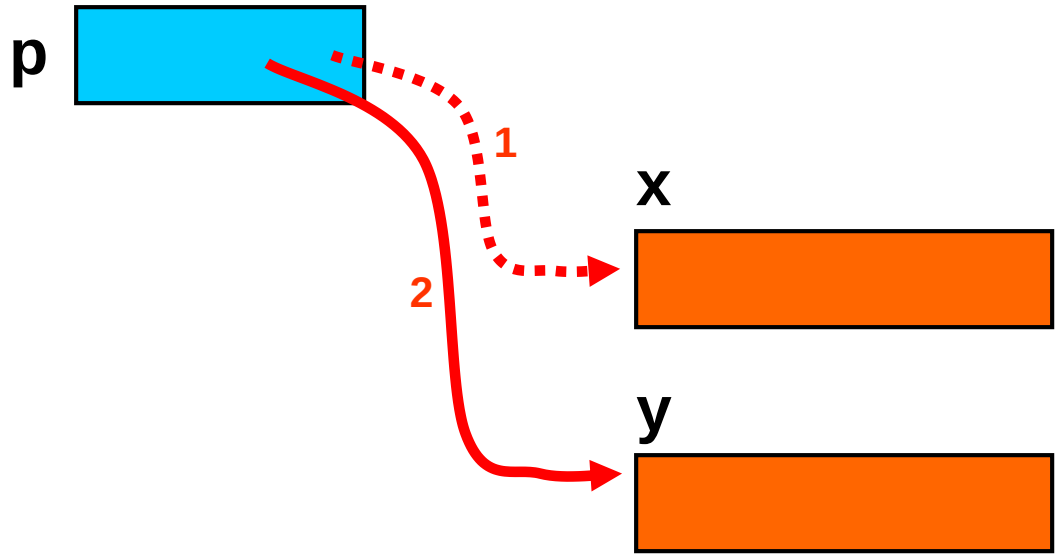


Pointers

1. **Pointer**:
   An object containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```

Unary "address-of" operator



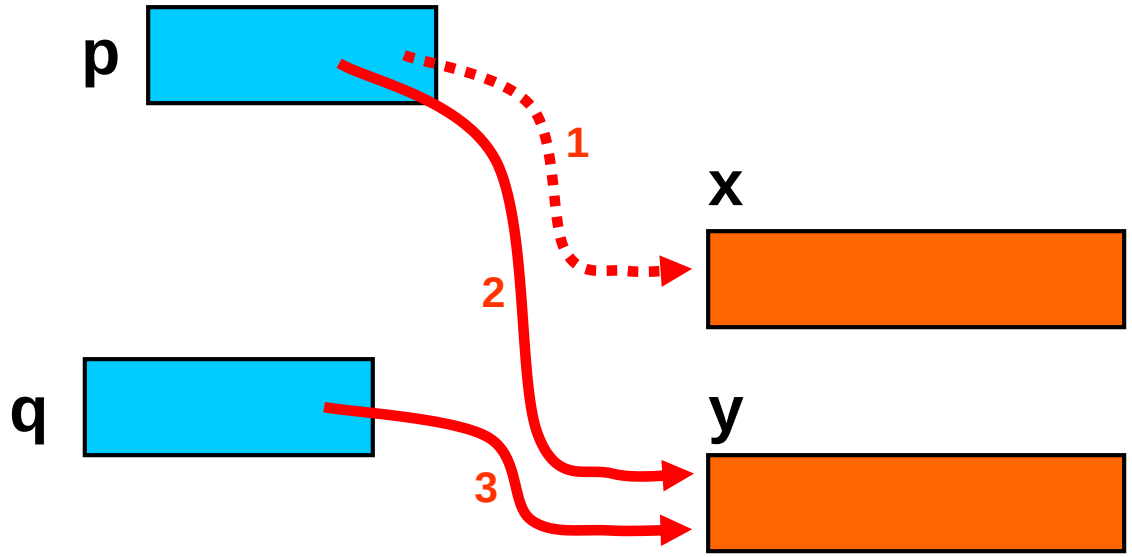

```
int y;
...
p = &y; 2
```

Pointers

1. **Pointer**:
   An object containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```

Unary "address-of" operator




```
int y;
...
p = &y; 2
```

```
int* q;
...
q = p;  3
```

Pointers

2. Pointer types

```
T* p;
```

Declaration of an object of a pointer type, where *T* denotes a type pointed

Examples:

- Pointers to (simple) variables; `int* pv;`
- Pointers to objects of struct types; `struct S* ps;`
- Pointers to functions; `int (*pf)(int);`
- Pointers to pointers; `int** p;`
- Pointers to values of any type `void* p;`

Pointers

3. Operators on pointers

&object Taking address of object

Unary prefix operator

```
int x;
int* p;
...
p = &x;
```

Pointers

3. Operators on pointers

&object

Taking address of object

Unary prefix operator

```
int x;
int* p;
...
p = &x;
```

***pointer**

Dereferencing:
Getting object pointed to by "pointer"

Unary prefix operator

```
int x;
int* p = &x;
...
*p = 777;      // x is 777
int z = *p+1; // z is 778
```

Pointers

3. Operators on pointers

&object

Taking address of object

Unary prefix operator

```
int x;
int* p;
...
p = &x;
```

***pointer**

Dereferencing:
Getting object pointed to by "pointer"

Unary prefix operator

See tutorial for other operators on pointers

Notice
The same token * is used for two different purposes:
a) for specifying a pointer type
b) as dereferencing operator.

...and for multiplication! ☺

```
int x;
int* p = &x;
...
*p = 777;       // x is 777
int z = *p+1; // z is 778
```

Arrays

A fixed-size indexed group of variables of the same type

```
T A[size];
```

`T` is the type of array elements

`A` is the array identifier

`size` specifies the number of array elements; this is an expression of an integer type

In general, `size` should be a **constant expression**

A fixed-size indexed group of variables of the same type

Arrays

```
T A[size];
```

`T` is the type of array elements

`A` is the array identifier

`size` specifies the number of array elements; this is an expression of an integer type

In general, `size` should be a **constant expression**

```
int Array[10];

const int x = 7;
void* Ptrs[x*2+5];

int Matrix[10][100];
```

The only operator on arrays:

- **Getting access to an element**

```
int el5 = Array[5];

Array[7] = 7;
```

A fixed-size indexed group of variables of the same type

Arrays

Arrays are very low-level and non-safe language feature

Why? - see examples in tutorial

```
T A[size];
```

`T` is the type of array elements

`A` is the array identifier

`size` specifies the number of array elements; this is an expression of an integer type

In general, `size` should be a **constant expression**

```
int Array[10];

const int x = 7;
void* Ptrs[x*2+5];

int Matrix[10][100];
```

The only operator on arrays:

- **Getting access to an element**

```
int el5 = Array[5];

Array[7] = 7;
```

The Program Example

The task:
- Find a given value in an array.

Version 1

```
int find1 ( float array[20], int x )
{
   for ( int i = 0; i<20; i++ )
   {
       if ( array[i] == x ) return i;  // success
   }
   return -1;  // fail
}
```

The Program Example

The task:
- Find a given value in an array.

Version 1

```
int find1 ( float array[20], int x )
{
    for ( int i = 0; i<20; i++ )
    {
        if ( array[i] == x ) return i;  // success
    }
    return -1;  // fail
}
```

Are you happy with this solution?

Arrays & Pointers

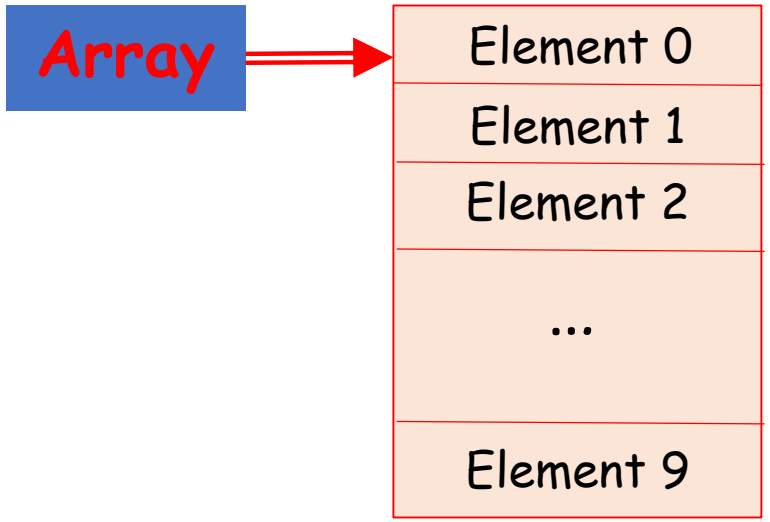
```
int Array[10];
```

By definition, array name is treated as a **pointer** to the first array element.

Arrays & Pointers

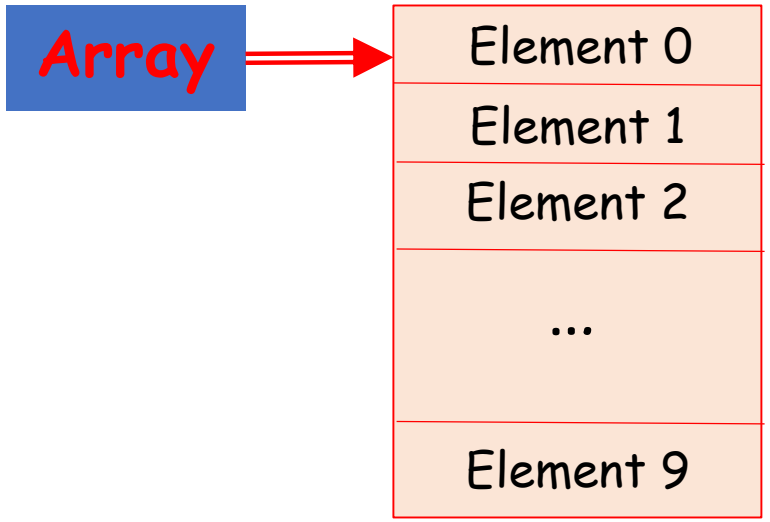
```
int Array[10];
```

By definition, array name is treated as a **pointer** to the first array element.



Arrays & Pointers

```
int Array[10];
```



By definition, array name is treated as a **pointer** to the first array element.

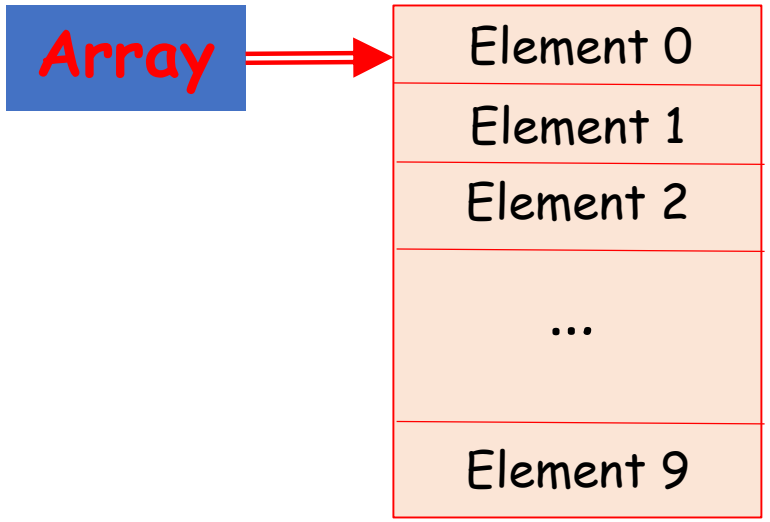
To be more precise, array name is a **constant pointer**

```
int Array[10];
```

```
const int* Array;
```

Arrays & Pointers

```
int Array[10];
```



By definition, array name is treated as a **pointer** to the first array element.

To be more precise, array name is a **constant pointer**

```
int Array[10];
```

```
const int* Array;
```

Therefore, these two constructs are semantically identical:

```
Array[0]
```

```
*Array
```

Do you see a problem here?

Arrays & Pointers

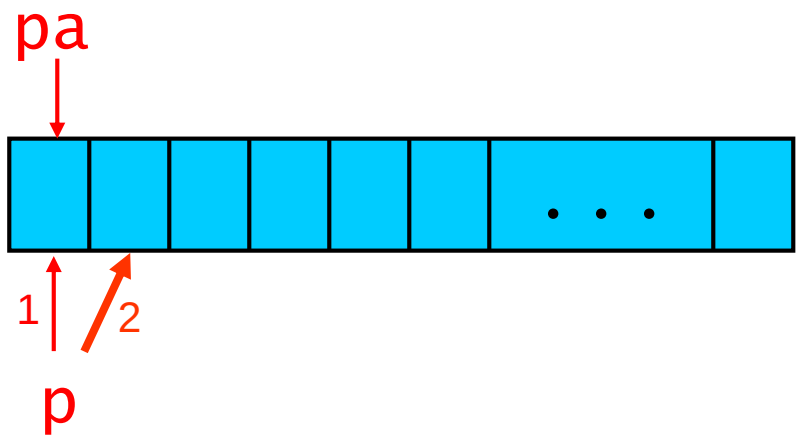
Operators on pointers: pointer arithmetic

```
pointer+i
pointer-i
pointer++
pointer--
ptr1-ptr2
```

```
int pa[10];
int* p = pa;  1

p++;          2
```

A question: Why pa++ is illegal?




```
T* p;
...
p+i;
```

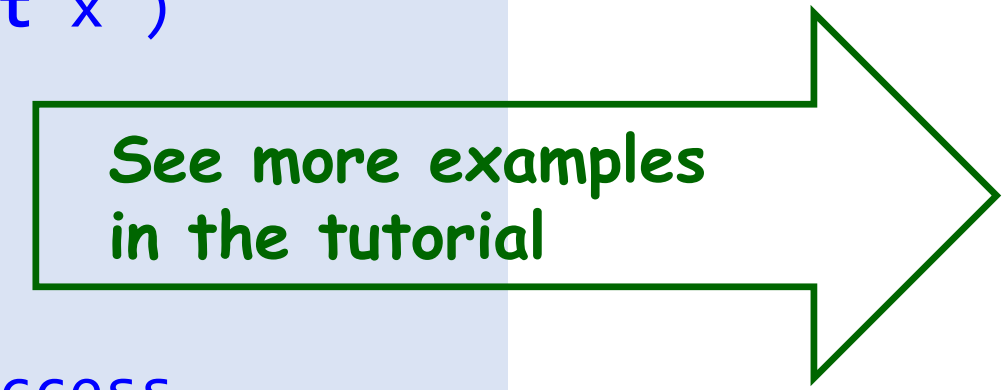
The same as

```
(T*)((char*)p + sizeof(T)*i)
```

The Program Example

Version 2

```
float* find2 ( float* array, int n, int x )
{
    const int* p = array;
    for ( int i = 0; i<n; i++ )
    {
        if ( *p == x ) return p;  // success
        p++;
    }
    return NULL;  // fail
}
```




```
float A[20];
...
float* res = find2(A, 20, 77);
```

Conclusion: What We Have Learnt Today

- The notion of **type**.
- Static and dynamic typing.
- Type categories.
- Storage class specifiers
- C type system: predefined & user-defined types.
- **Pointers & arrays**