Data object

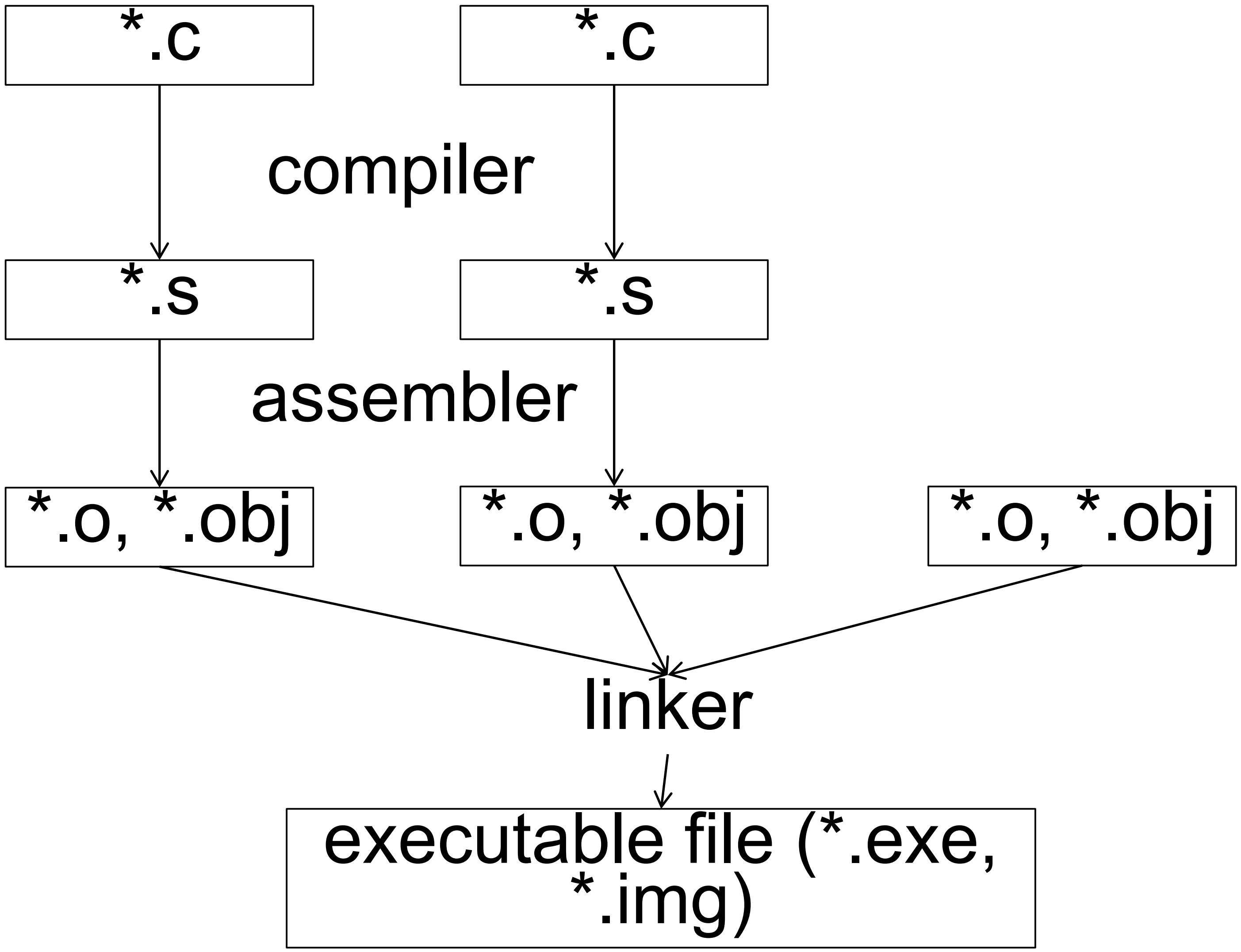
- Object in C is "region of data storage" ... "the contents of which can represent values" of particular type.
- We can attach a name to an object, a that is what we call: **variable**.
- In other words, a variable is an object (of some <u>type</u>) that has a <u>name</u> ("identifier")

Variable properties

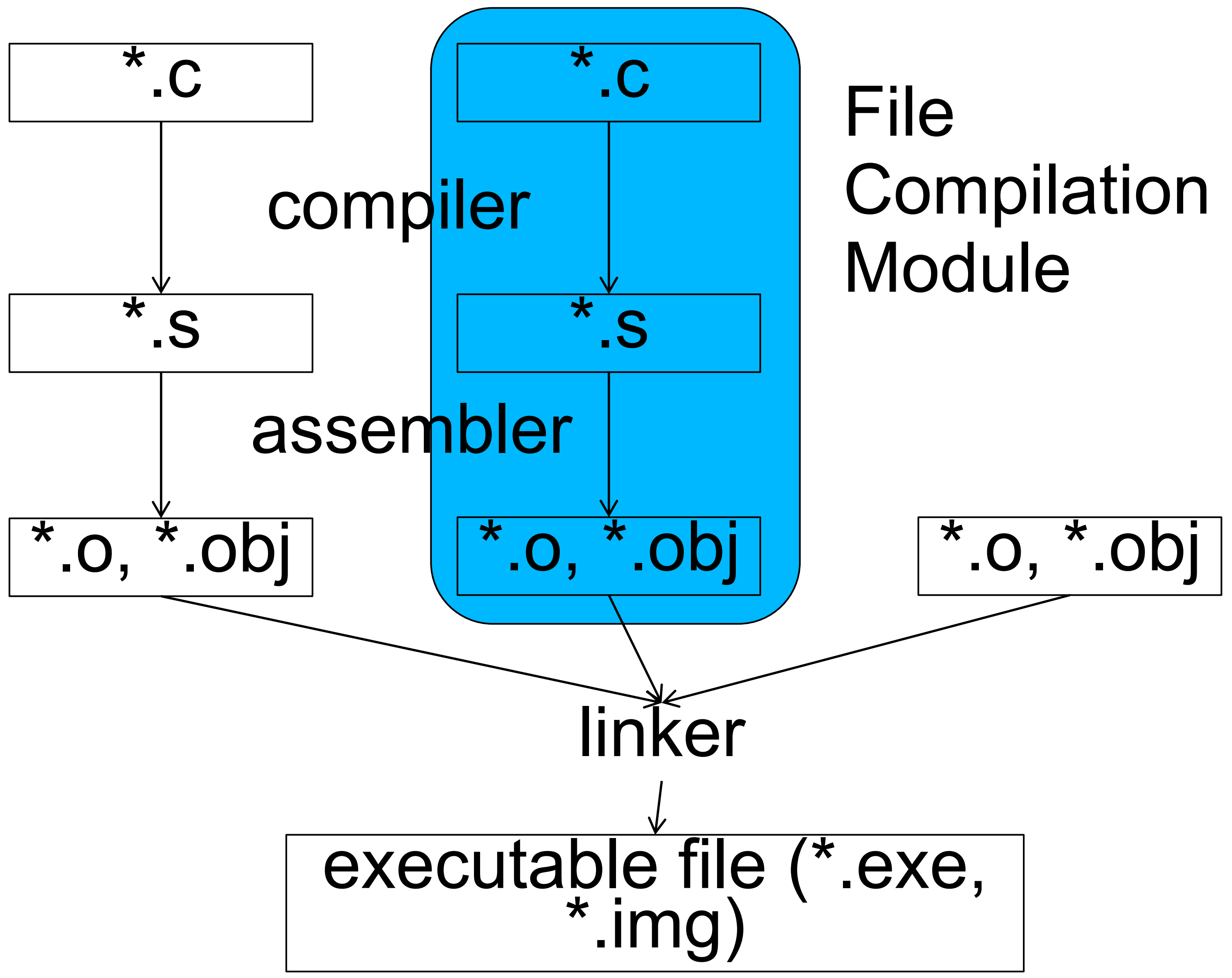
- Name and type are not the only two properties of a variable!
- All variable properties are expressed and attached to it in its declaration.
- Some properties are covered by the standard, but there might exist some platform, or compiler, specific properties.
- (Essentially, all of the following are either properties of object or properties of identifier, but we mostly deal with variables that indirectly then encompass all those properties)

- Three key properties:
  - Scope
  - Linkage
  - Storage duration
- Two additional properties:
  - const
  - volatile

Program building - picture



Program building - picture



File
Compilation unit
Module

Variable name

- The first character must be letter or _.
- After the first character, digits can be used too.
- Consequence of the two previous points: no spaces.
- Variable name can not be the same as a reserved word.
- Names are case sensitive, so **Mile** is not the same name as **mile**.
- Only first 63 characters are guarantied to uniquely identify a variable.
  - Only 32 in case of extern variables.
- Using _ at the beginning is discouraged because it is usually reserved for some hidden library and system identifiers.

Advices:

- The names have to be meaningful and informative.
  - **studentAge** or **student_age** gives more information about the variable, and its purpose, than just **age**, or, God forbid, just **a**.
- When name is formed out of several words, those words can be visually separated by inserting _, or by capitalizing the first letter of each word.
- Choose one approach and stick to it. In case when you work on a project with other people, and some coding standard is already accepted, then follow that standard.

Variable definition and declaration

- **Declaration** describes a variable, giving enough information so that compiler could know how to deal with that variable.
- **Definition** represents allocation of a physical resource for the variable. It can also include its initialization.
- Definition can not exist without declaration, but declaration can be done without definition.
  - For variables, all declarations are also definitions, except in one case.
  - For functions, the difference is clearer.

Examples of declaration (and definition)

- One declaration (with definition) in one line

```
int age;
float amountOfMoney;
char initial;
```

- Several declarations (with definitions) in one line

```
int age, houseNumber, quantity;
float distance, rateOfDiscount;
char firstInitial, secondInitial;
int* p1, p2, p3; // sta je ovde problem?
```

- For functions
  - Declaration (but no definition):

    ```
    int foo(int x);
    ```

  - Declaration with definition:

    ```
    int foo(int x)
    {
      return x + 1;
    }
    ```

Variable scope

- Scope is determined by location of the declaration.
- Variables can have three types of scope:
  - **File scope** – global variables (in C++ it is called “global scope”).
    - Declared outside any block or parameter list
  - **Block scope** – local variables (in C++ it is called “local scope”)

```
{
   int x;
   ...
}
```

```
void foo(int x)
{
    ...
}
```

  - **Prototype scope** – special case for conceptual clarity, but not very significant.

```
void foo(int x);
```

Variable scope

- Two variables in a different scope can have the same name.
- But:

```
int milorad;
void foo()
{
    int milorad;
    ...
}
```

- Try to avoid this!

Variable visibility from other files (Linkage)

- Only applies to variables of file scope.
- **Never use it on variables of block scope.**
- 1. Variable is defined in the file and is only used in it – **private (internal linkage)**

```
static int x; // declaration and definition – variable not visible form the
outside
```

- 2. Variable is defined in the file but it can be used in other files as well – **public (external linkage definition)**

```
int x; // declaration and definition – variable is visible from the outside
```

- 3. Variable is not defined in the file, but it is used in it (it is defined in some other file) – **extern (external linkage declaration only)**

```
extern int x; // only declaration - x has to be defined somewhere else
```

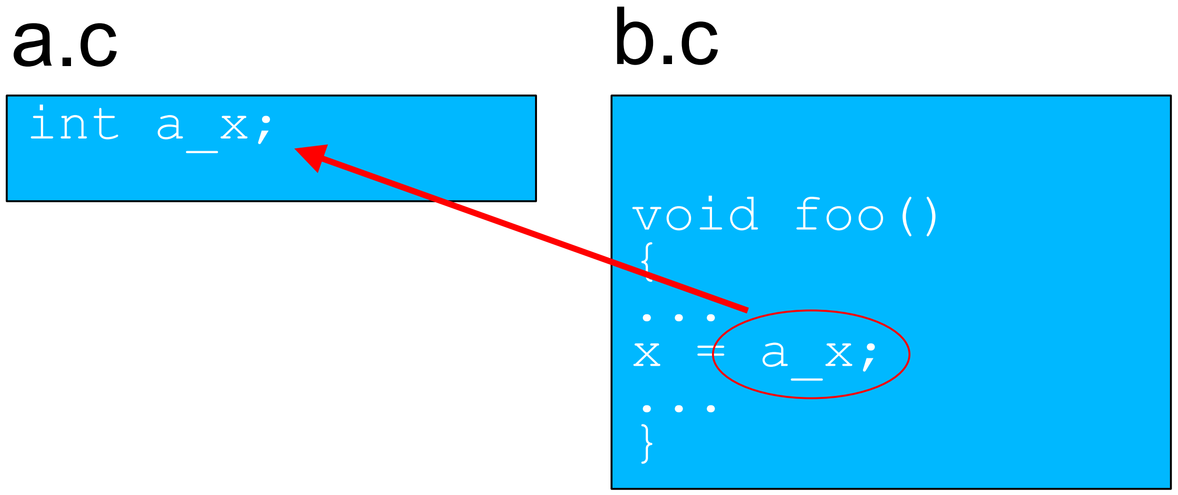
- This is also allowed to be in the same file:

```
extern int x;

int x;
```

- This too, and also some other combinations, but you shouldn't use it – only the above is useful:

```
static int x;

int x;
```

Extern - public


a.c

```
int a_x;
```

b.c

```
void foo()
{
...
x = a_x;
...
}
```

Extern - public

From the whole program point of view, there is only one variable a_x and it has “external linkage”.

Looking from a.c compilation unit, a_x is its public data/variable.
Looking from b.c compilation unit, a_x is external data/variable.

a.c

```
int a_x;
```

b.c

```
extern int a_x;

void foo()
{
...
x = a_x;
...
}
```

Extern - public

a.h

```
#ifndef _A_H_
#define _A_H_

extern int a_x;

#endif
```

a.c

```
#include "a.h"


int a_x;
```

b.c

```
#include "a.h"


void foo()
{
...
x = a_x;
...
}
```

Function visibility from other files (Linkage)

- Applies on functions of file scope (global functions).
- **You should not declare functions in blocks.**
- 1. Function is defined in the file and is only used in it – **private (internal linkage)**

```
static void foo(int x) { ... }
```

- 2. Function is defined in the file but it can be used from other files as well – **public (external linkage definition)**

```
void foo(int x) { ... }
```

- 3. Function is not defined in the file, but it is used in it (it is defined in some other file) – **extern (external linkage declaration only)**

```
void foo(int x); // only declaration, extern keyword is not necessary
```

- This is also allowed to be in the same file :

```
void foo(int x);
void foo(int x) {...}
```

- This too, and also some other combinations, but you shouldn't use it – only the above is useful :

```
static void foo(int x);
void foo(int x) {...}
```

Extern - public

a.c

```
int a_bar()
{
...
}
```

b.c

```
int a_bar();

void foo()
{
...
x = a_bar();
...
}
```

Extern - public

a.h

```
#ifndef _A_H_
#define _A_H_

int a_bar();

#endif
```

a.c

```
#include "a.h"


int a_bar()
{
...
}
```

b.c

```
#include "a.h"


void foo()
{
...
x = a_bar();
...
}
```

Storage duration of variables (objects)

- Determines when variable (object) is created (resources are taken for it) and when it is destroyed (resources released).
- Three kinds of storage duration:
  - **Automatic**

    Creation and destruction of the variable is controlled by compiler, i.e. the variable is created when it is needed (in its scope) and destroyed when not needed.

    Keyword **auto**. The default storage duration for variables of block scope. File scope variables **can not** have automatic storage duration. They are created uninitialized.
  - **Static**

    The variable is created once, when program starts, and is alive until the end of the program execution.

    The default (and only) storage duration of file scope variables. Block scope variables **can** have static storage duration. In those cases the keyword is **static (but not "static" for specifying private variables and function, but different "static" ☺)**. They are created initialized to 0 (or NULL).
  - **Allocated**

    Creation and destruction of objects is controlled by programmer himself, using functions from stdlib.h. This can not be done for variables, only objects, which are accessed only through pointers. Whether the object would be initialized or not depends on the used function.

Storage duration of variables (objects)

- There is the forth kind of storage duration (since C11):
  - **Thread**

    The variable is created once, when a thread starts, and it is alive until the end of that thread. Each thread has its own instance of that variable. Keyword is **thread_local**. It can be applied to both local and global variables (same as static storage duration).

Table

	Private (internal)	Public (external definition)	Extern (external declaration)
Automatic Global	None	None	None
Static Global	`static int x;`	`int x;`	`extern int x;`
Automatic Local	`{ auto int x; }` (*`auto`* italic)	None	None
Static Local	`{ static int x; }`	None	None

Code in italic is optional..

Note two meanings of `static` keyword: 1) `static` as opposed to `auto`, and 2) `static` as specifier that the variable is private (not public or extern, i.e. it has internal linkage)

Example

```
/* uninitialized global variable */
int x_global_uninit;

/* initialized global variable */
int x_global_init = 1;

/* uninitialized global variable (static)*/
static int y_global_uninit;

/* initialized global variable (static) */
static int y_global_init = 2;

/* global variable that exists somewhere
 * else in the program */
extern int z_global;
```

```
int foo(int x_local)
{
  /* uninitialized local variable */
  int y_local_auto_uninit;

  /* initialized local variable */
  int y_local_auto_init = 3;

  /* uninitialized local variable */
  static int y_local_static_init;

  /* initialized local variable */
  static int y_local_static_init = 4;
}
```

What is the problem?

```
void count()
{
    int k;
    int i;

    for (i = 0; i < 10; i++)
    {
        k = k + 1;
    }

    printf("%d", k);
}
```

What is the problem?

```
void foo(int a)
{
   int a;

   ...

   if (a != 0)
   {
      /* do something here */
   }
}
```

What is the problem?

```
void bar(int a)
{
   char* b;

   if (a != 0)
   {
     b = malloc(a);
   }

   if (b != NULL)
   {
     *b = a;
   }
}
```

How much you should use external linkage variables

- In general, you should avoid using external linkage variables. It constitutes good programming practice because:
  - extern variables reduce program modularity
  - they make debugging harder
- But, sometimes it is necessary, so the advice is to use them carefully.

How much you should use public variables

- Public variables are necessary if it is expected that there will be corresponding extern variables in other files.
- But why do we see a lot of public variables in some code?

  Answer: Due to lack of knowledge about the programming language, or out of laziness.
- If global variable is not planed for access from other files, then it should be declared as private (internal linkage).
- If you do not do that, problems can happen: What if there are two public variables of the same name in two different files?
- Also, do not forget that functions can be private (have internal linkage) as well. Use that.

Where to declare/define variables?

- Variables should be defined exclusively in .c files, never in headers (exceptions are static const variables, but we will talk about that later).
- In case of global variables, you should define/declare them at the top of the file, usually after header inclusions.
- In case of local variables starting from C99 standard you can do it anywhere in a block – and you should do that when you need a variable and you know how to initialize it; which brings us to the next advice:
- Variables should be initialized as soon as possible – most preferably immediately when you define them.

Qualifiers

- `volatile` – means that a certain variable can be changed in some way other than by program itself. This is rarely encountered outside the code that more directly works with hardware.
- `const` – means “read-only” by the code, i.e. the code can not change the value of that variable. In a way, it is a promise that compiler helps you keep.

Note the difference:

1. **`const char* p`**
2. **`char const* p`**
3. **`char* const p`**

1. and 2. are the same thing: declaration of pointer to `const char`. It means that the pointer can be modified, but that to which it points to can not.

3. declares const pointer to `char`. It means that the pointer itself can not be modified, but that to which it points to can be.

Constant variables ☺

- Declaration of constant variable is the same as declaration of regular variable - keyword `const` is the only difference.
- Word `const` can be placed on any side of the type:

```
int const a = 1; // "east const"
const int a = 2; // "west const"
```

  It is mostly a matter of style (although the first variant has advantages is you want to declare multiple pointers with some const qualifier). Choose one variant and stick to it through out the code.
- The const variables have to be initialized when defined, because they can not be changed later from the code.
- #define is an alternative approach used to create named constants in the code but there are some important differences.

#define vs const

```
#define SOME_CONST 1

static const int some_const = 1;
```

Difference?

- Constant variable can be of more complex type
- Constant variable has address and physical representation in memory, if it is needed (if it is not needed, compiler can optimize it out)

  ```
  int* x = &some_const;
  ```

  Constant variables have scope (and linkage).

  With const variables you are avoiding potential problems, e.g.:

  ```
  #define SOME_CONST 5 + 2
  const int some_const = 5 + 2;
  x = 7 * SOME_CONST; y = 7 * some_const;
  ```

Example for volatile

```
volatile int _flag;

void wait_for_flag()
{
  while(_flag == 0);
}
```

```
volatile int x;

int foo()
{
  x = 5;
  return x;
}

int foo1()
{
  x = 5;
  return 5;
}
```

This optimization is valid if x is not marked as `volatile`

Example for volatile

```
volatile int _flag;

void wait_for_flag()
{
   while(_flag == 0);
}
```

```
volatile int x;

int foo()
{
   x = 5;
   return x;
}

int foo1()
{
   //x = 5;
   return 5;
}
```

This optimization is valid if x is not marked as `volatile`, and x is only used in this function.

Example for volatile

```
volatile uint8_t* flag = (uint8_t volatile*)0x12345678;
```

const volatile?

```
volatile uint8_t* flag = (uint8_t volatile*)0x12345678;

volatile uint8_t* const flag
    = (uint8_t volatile*)0x12345678;

volatile const uint8_t* const flag
    = (uint8_t const volatile*)0x12345678;

const uint8_t volatile * const flag
    = (volatile uint8_t const*)0x12345678;
```