Week 8

XMUT-NWEN 241 - 2024 T2

Systems Programming

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

Content

- Storage Classes
- C Process Layout

Storage Classes

Variable Lifetime and Scope

Variables have lifetime*: it begins when the variable is allocated memory and ends when the memory is “de-allocated”

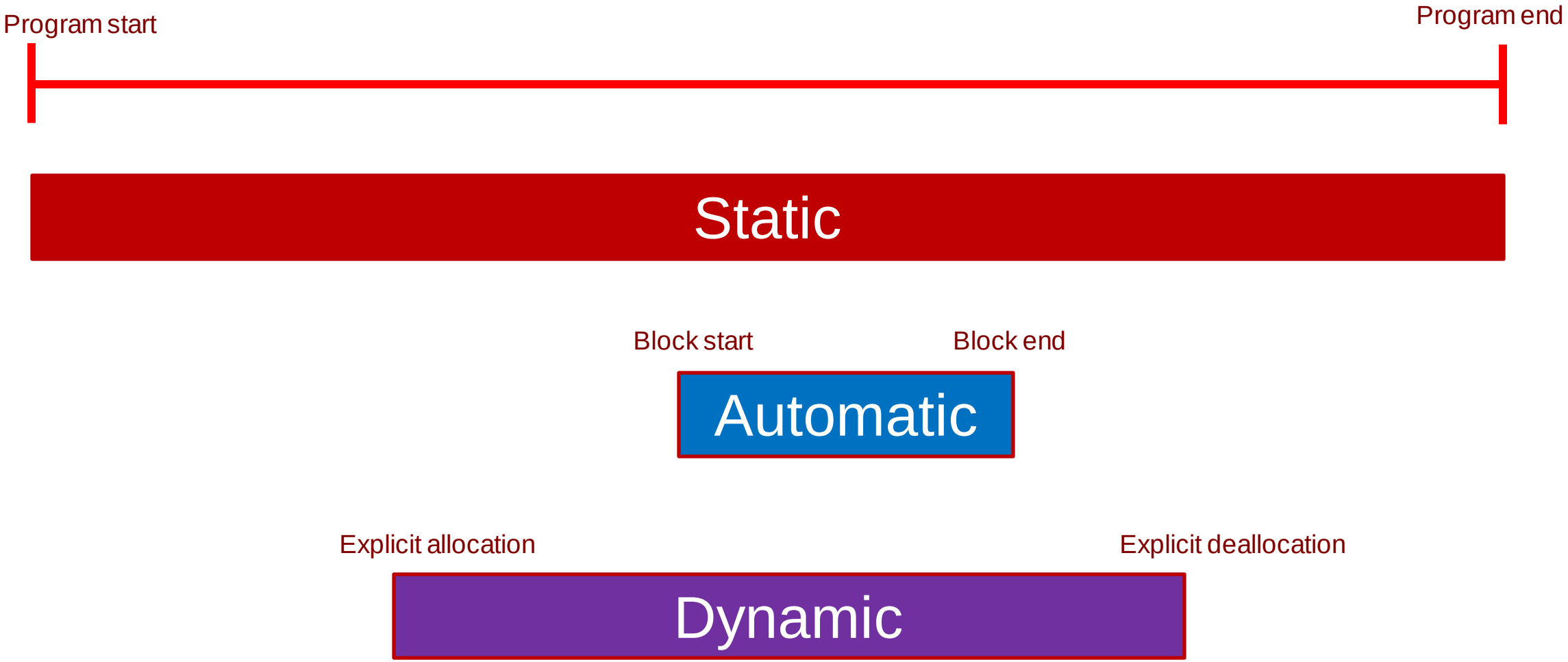
Variables have scope: these are parts of the program where a variable is visible

*Lifetime is also known as **storage duration**

Lifetime

- Static
  - A static variable exists for the entire program execution duration; allocated memory when program starts
- Automatic
  - An automatic variable exists within a block that contains it; allocated memory when execution enters the block and de-allocated when execution leaves the block
- Dynamic
  - A dynamic variable exists from allocation to de-allocation of memory; allocation and de-allocation are done explicitly through dynamic memory allocation function calls

Lifetime

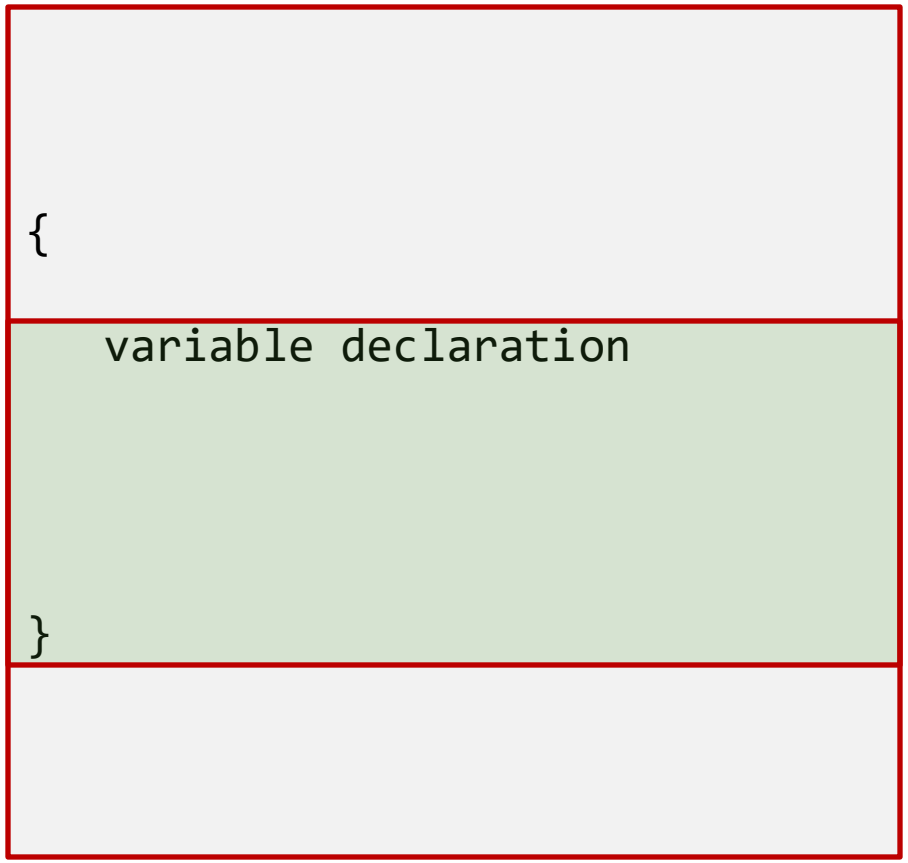

Program start
Program end
Static
Block start
Block end
Automatic
Explicit allocation
Explicit deallocation
Dynamic

Scope

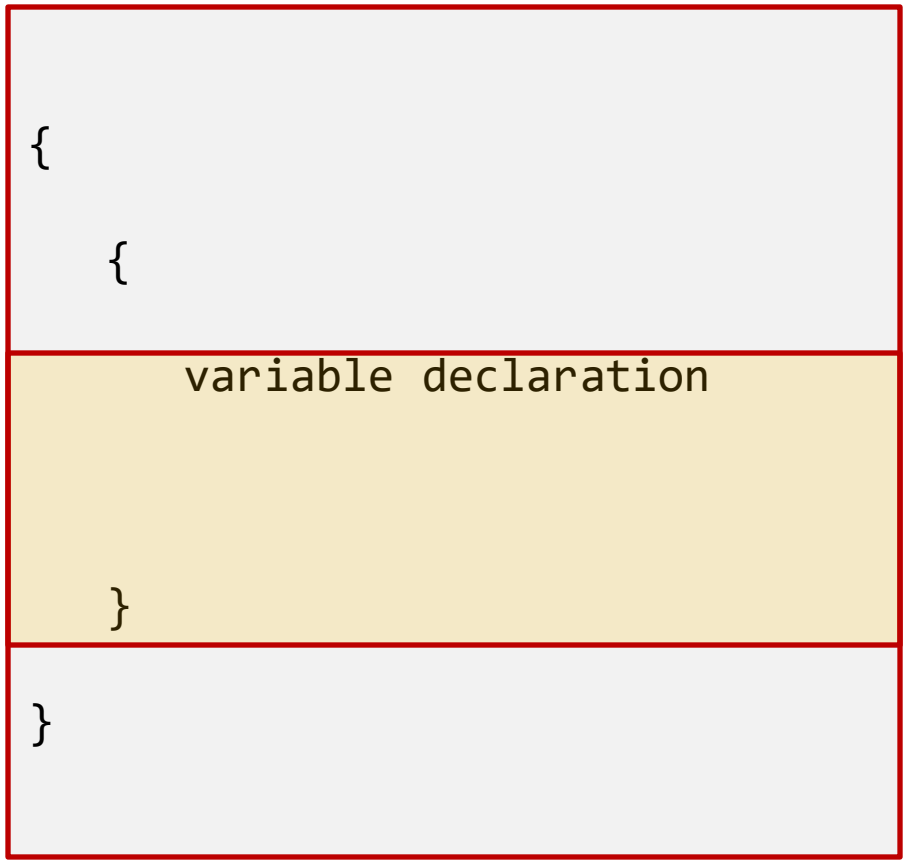
- Local
  - A local variable is only visible inside the current, innermost block
- Global
  - A global variable is visible in the *whole* compilation unit, from the line of declaration to the end of file
- External
  - An external variable is visible in *all* compilation units

Local Scope

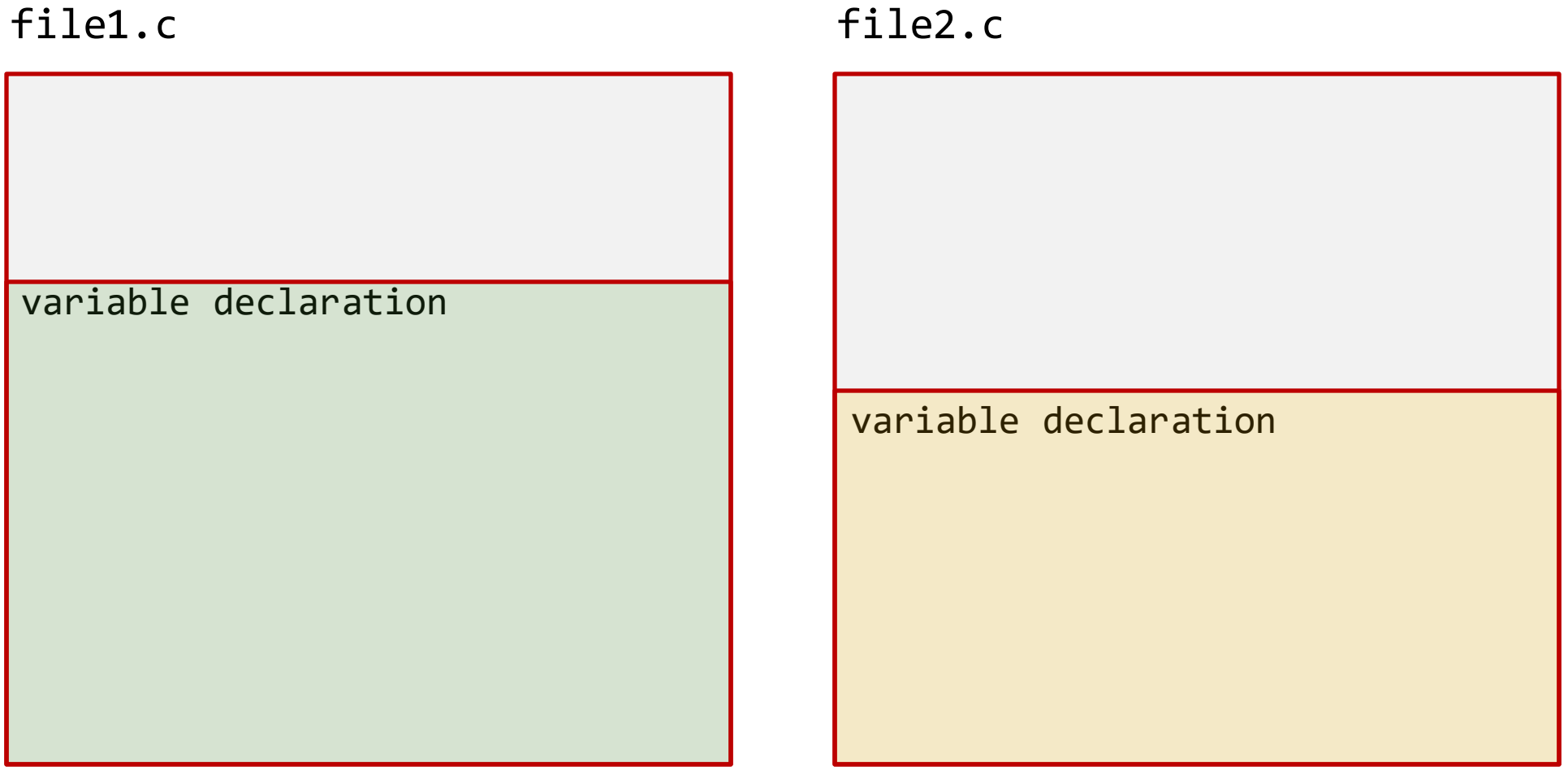
file1.c



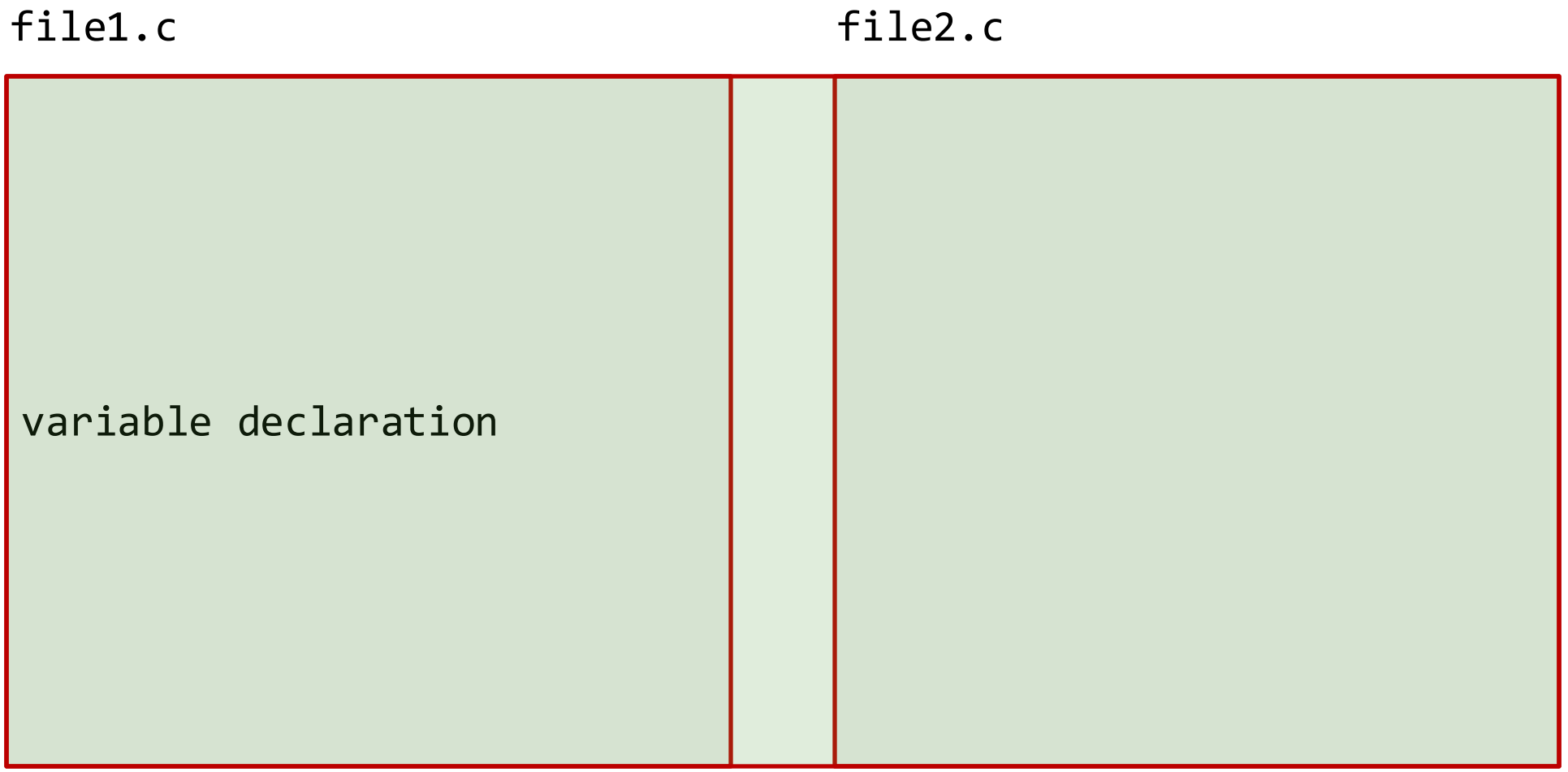
file2.c



Global Scope



External Scope



Storage Classes at a Glance

C storage class	Declaration	Default init value	Init frequency	Stored in	Scope	Lifetime
`auto`	Inside block	Garbage	Every time block is entered	Memory	Local	Automatic
`static`	Inside block	0	Once at program start	Memory	Local	Static
	Outside any block	0	Once at program start	Memory	Global	Static
`extern`	Outside any block	0	Once at program start	Memory	External	Static
`register`	Inside block	Garbage	Every time block is entered	Maybe in register	Local	Automatic

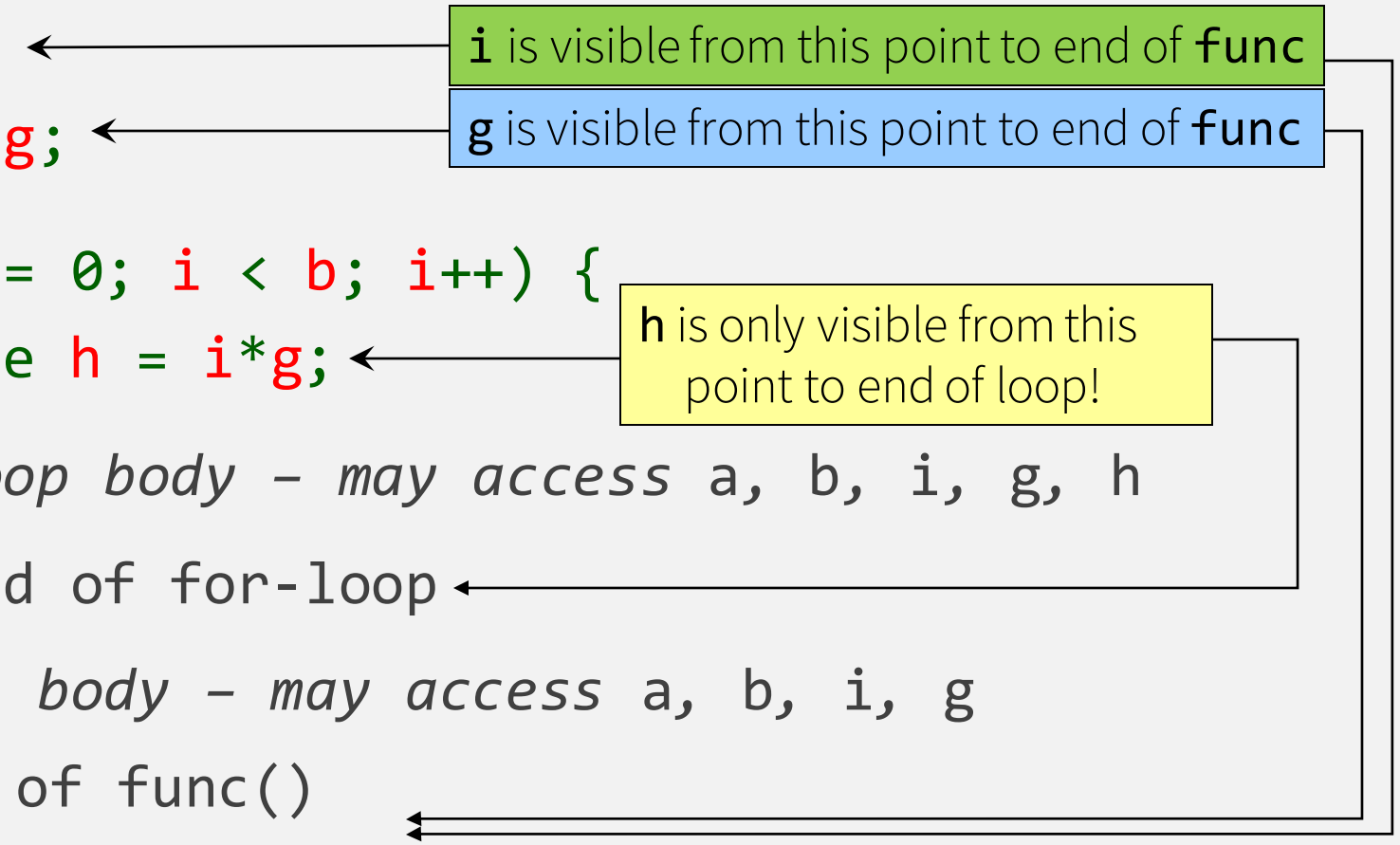
auto Storage Class Declaration

```
{
   auto double x; /* Same as: double x */
   int num;       /* Same as: auto int num; */
   ...
}
```

- **auto** is the default storage class for a variable defined inside a function body or a statement block
- **auto** prefix is optional; *i.e.,* any locally declared variable is automatically **auto**, unless specifically defined to be static

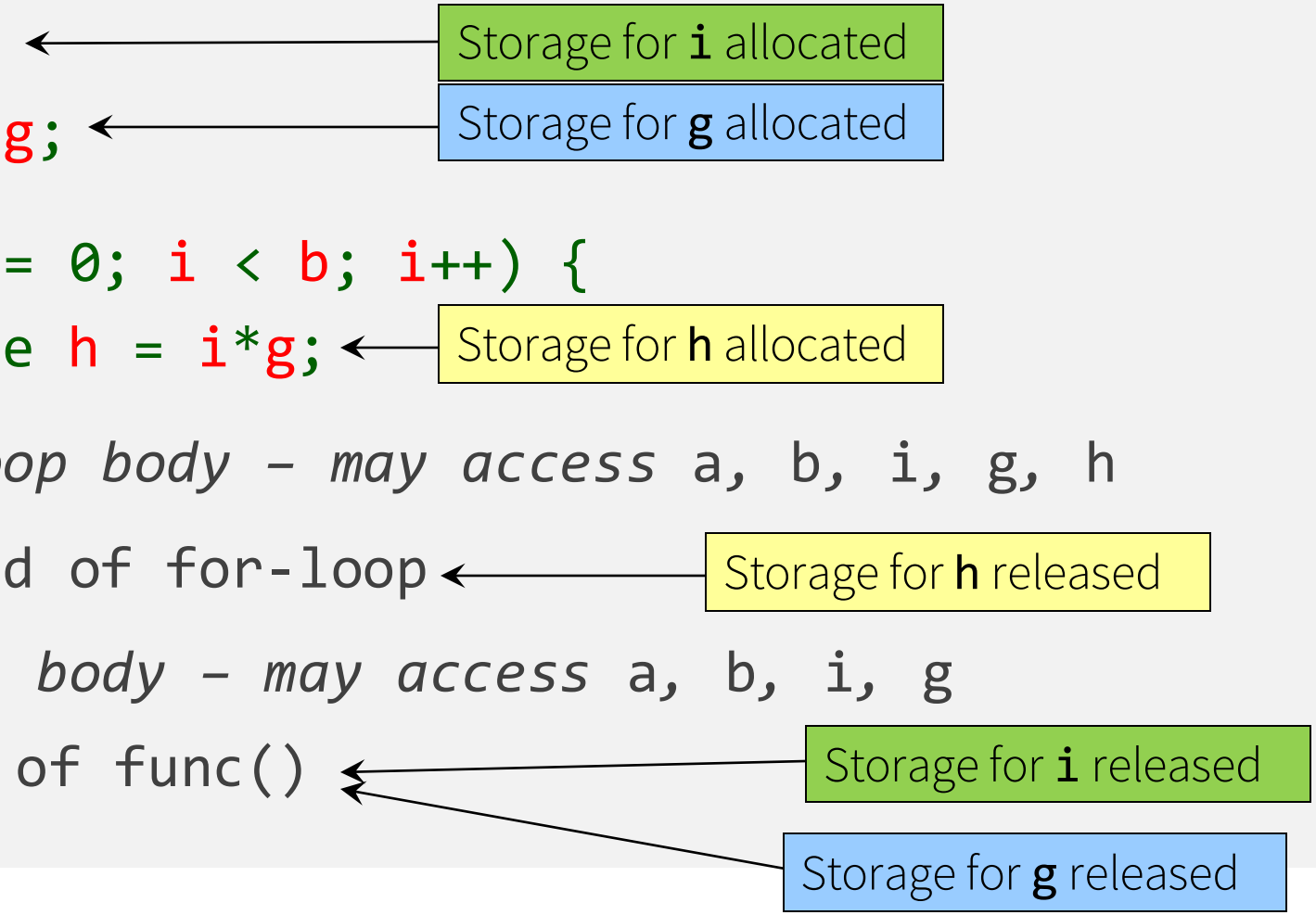
auto Storage Class Example (Scope)

```
int func(float a, int b)
{
  int i;
  double g;

  for (i = 0; i < b; i++) {
    double h = i*g;

    // loop body - may access a, b, i, g, h

  } // end of for-loop

  // func body - may access a, b, i, g
} //  end of func()
```



auto Storage Class Example (Lifetime)

```
int func(float a, int b)
{
  int i;
  double g;

  for (i = 0; i < b; i++) {
    double h = i*g;

    // loop body – may access a, b, i, g, h

  } // end of for-loop

  // func body – may access a, b, i, g
} //  end of func()
```



static Storage Class Declaration

- The **static** prefix *must* be included

```
{
    static double local_static;
    …
}
```

```
void func1(int a)
{
    …
}

static int global_static;

void func2(int b)
{
    …
}
```

static Storage Class Example

```
#include <stdio.h>

void strange( int x )
{
        static int y;
        if ( x == 0 )
                printf( "%d\n", y );
        else if ( x == 1 )
                y = 100;
        else if ( x == 2 )
                y++;
}
int main (void)
{
        strange(1);  /* Set y in strange to 100 */
        strange(0);  /* Will display 100        */
        strange(2);  /* Increment y in strange  */
        strange(0);  /* Will display 101        */
        return 0;
}
```

Program output

```
$ gcc -o strange strange.c
$ ./strange
100
101
$
```

extern Storage Class Declaration

- `extern` is the default storage class for a variable defined *outside* any function's body

```
void func1(int a)
{
    …
}

int global;

void func2(int b)
{
    …
}
```

extern Storage Class Example 1

```
#include <stdio.h>

float x = 1.5;

void show (void)
{
     printf("%f\n", x); /* Access external x */
}

int main (void)
{
     printf("%f\n", x); /* Access external */
     show();
     return 0;
}
```

extern Storage Class Example 2a

What if x is defined after main and you want to use it in main?

```
#include <stdio.h>

extern float x;

void show (void)
{
      printf("%f\n", x); /* Access external x */
}

int main (void)
{
      printf("%f\n", x); /* Access external x */
      show();
      return 0;
}

float x = 1.5;
```

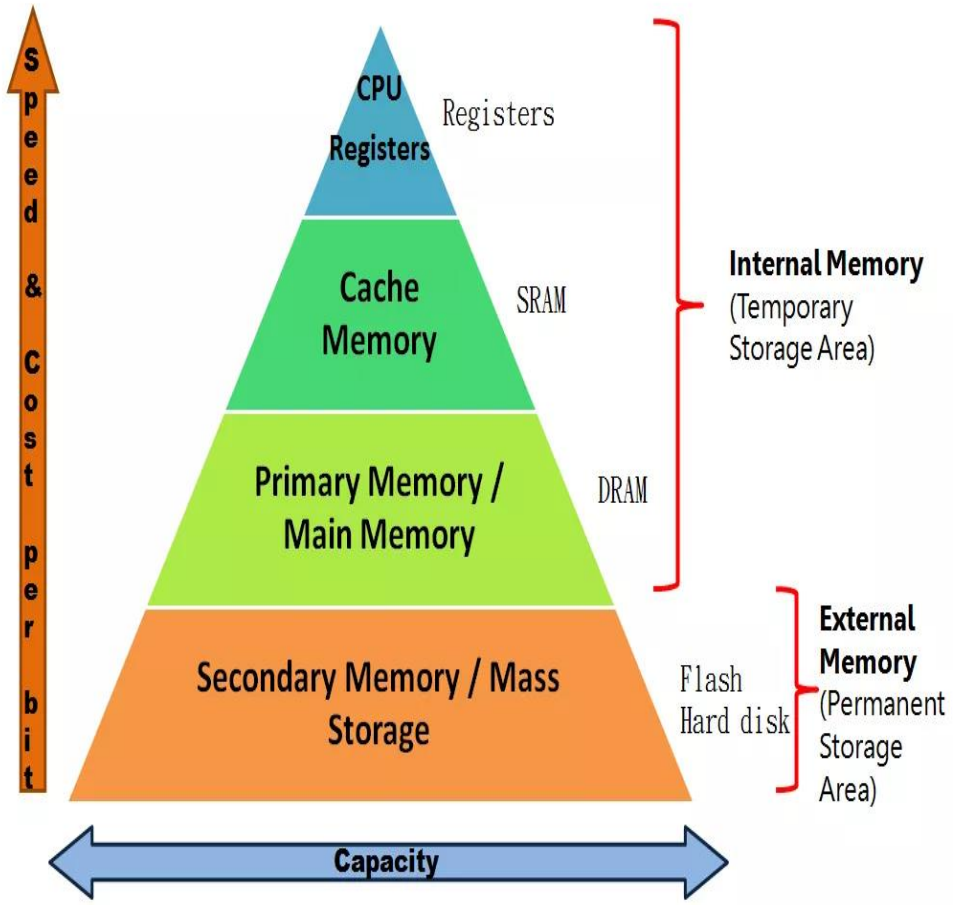
extern Storage Class Example 2b

What if x is defined in another source file?

```c
#include <stdio.h>

void show (void);

int main (void)
{
    printf("%f\n", x); /* Access external x */
    show();
    return 0;
}

float x = 1.5;
```

```c
#include <stdio.h>

extern float x;

void show (void)
{
    printf("%f\n", x); /* Access external x */

}
```

register storage class



- The fastest storage resides within the CPU itself in high-speed memory cells called *registers*
- The programmer can *request* the compiler to use a CPU register for storage

- Example:

```
register int k;
```

register storage class

- The compiler can ignore the request, in which case the storage class defaults to `auto`

- A register variable is local to the block which contains it

Variable shadowing

- Variable shadowing happens when a variable within an inner scope is declared and given the same name as another variable that is declared in an outer scope

```
int x = 100;

int main (void)
{
   int x = 1;

   printf("%d\n", x);

   return 0;
}
```

- The *local variable* x inside main shadows the *global variable* x
- References to x inside main results in access to local variable x

Variable in outer scope is shadowed by variable in inner scope

Pop quiz

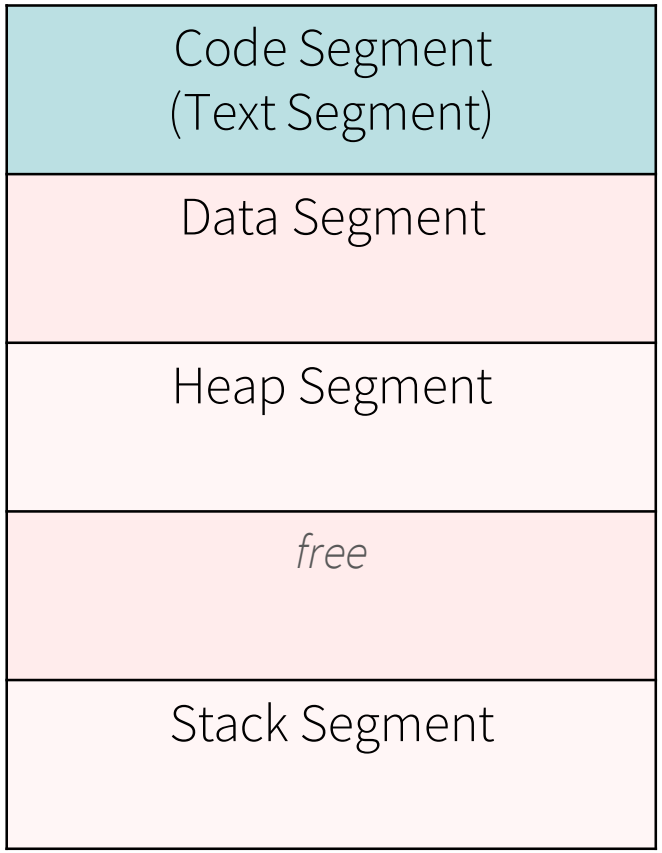
- How to access the global variable x from within `main()` without renaming the local variable x?

```
int x = 100;

int get_x(void)
{ return x; }

int main (void)
{
   int x = 1;

   printf("%d\n", get_x());

   return 0;
}
```

C Process Layout

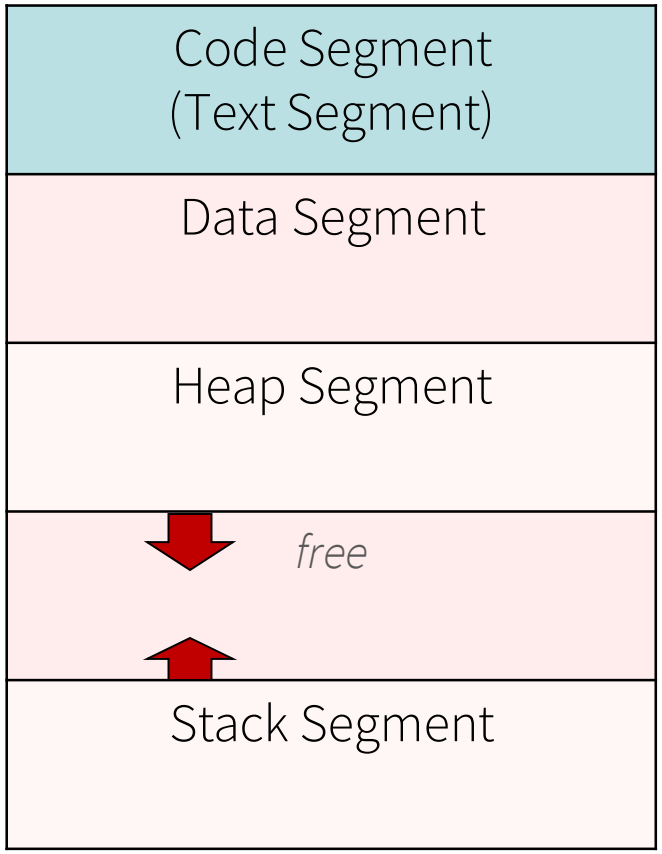
C Process Layout

- Memory space for program code includes space for machine language **code** and **data**
- **Text / Code Segment**
  - Contains program's machine code
- Data spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs



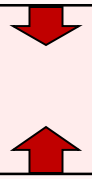
C Process Layout

- Memory space for program code includes space for machine language **code** and **data**
- **Text / Code Segment**
  - Contains program's machine code
- Data spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs



Storage Layout

- Where are auto, static, and extern variables stored?

Contains the program's machine code	Code Segment (Text Segment)
Contains static data (e.g., **`static`** class, **`extern`** class)	Data Segment
Contains dynamically allocated data – later…	Heap Segment
Unallocated memory that the stack and heap can use	 *free*
Contains temporary data (e.g., **`auto`** class)	Stack Segment

Next Lecture

- Dynamic Memory Allocation