

CSCI 2410 Systems Programming

Fall 2026

Instructor: Jie Zhou

**Department of Computer Science
George Washington University**



Slides materials are mostly credited to Gabriel Parmer and Sibin Mohan at GWU.

Scoping Rules

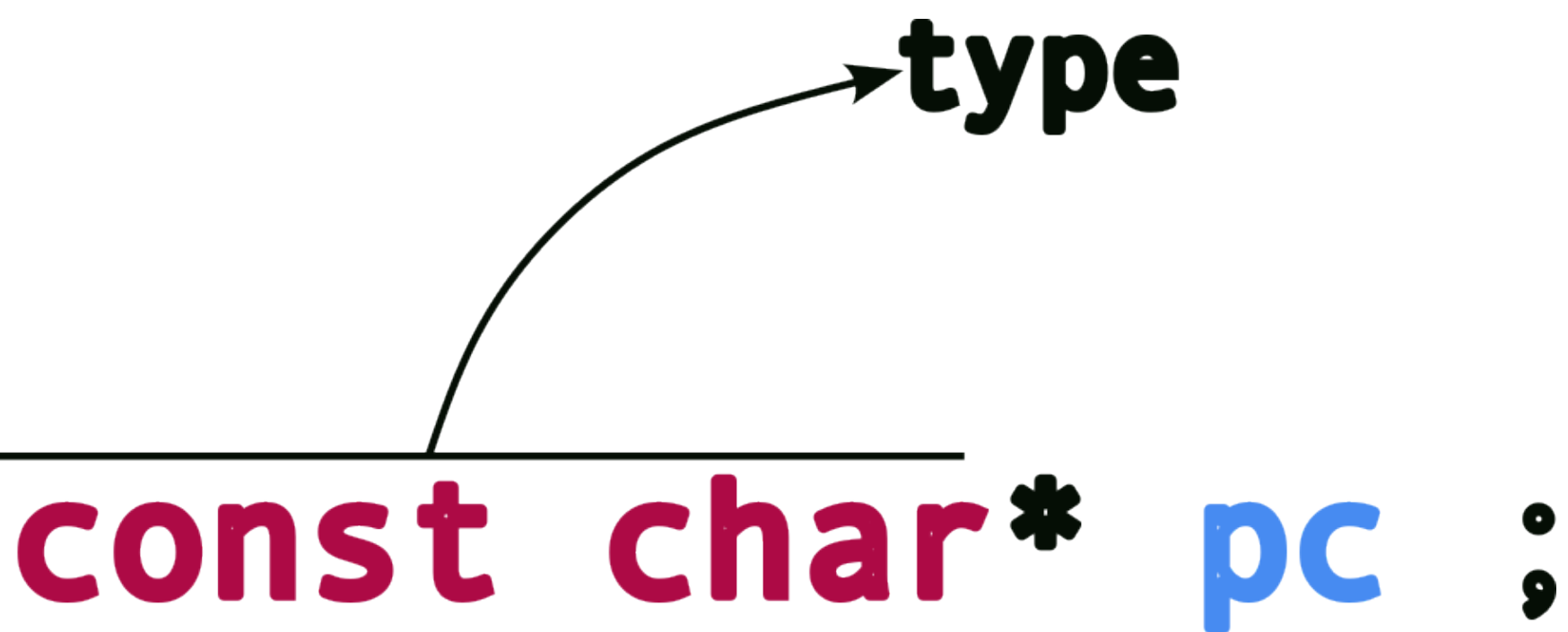
Identifiers and Types

- Most things in C have names, also known as *identifiers*.
- In fact, most things in C have a *name* and a *type*

```
const char* pc ;
```

Identifiers and Types

- Most things in C have names, also known as *identifiers*.
- In fact, most things in C have a *name* and a *type*



const char* **pc** ;

The diagram illustrates the components of a C declaration. A horizontal line is drawn under the text "const char* pc ;". An arrow originates from the line, specifically pointing to the "char*" part, and curves upwards and to the right to point at the word "type".

Identifiers and Types

- Most things in C have names, also known as *identifiers*.
- In fact, most things in C have a *name* and a *type*

The diagram shows the C declaration `const char* pc;`. The text is color-coded: `const` is red, `char` is red, `*` is black, `pc` is blue and enclosed in a black box, and `;` is black. An arrow points from the `char*` part to the word **type**. Another arrow points from the boxed `pc` to the text **name/identifier**.

Identifiers and Types

- Most things in C have names, also known as *identifiers*.
- In fact, most things in C have a *name* and a *type*
- Typical things that have names:
 - variable names
 - function names
 - file names
 - other resources, e.g., network socket
- Every identifier/name in C has a *scope*, i.e., where that identifier is valid.

Declarations and Definitions

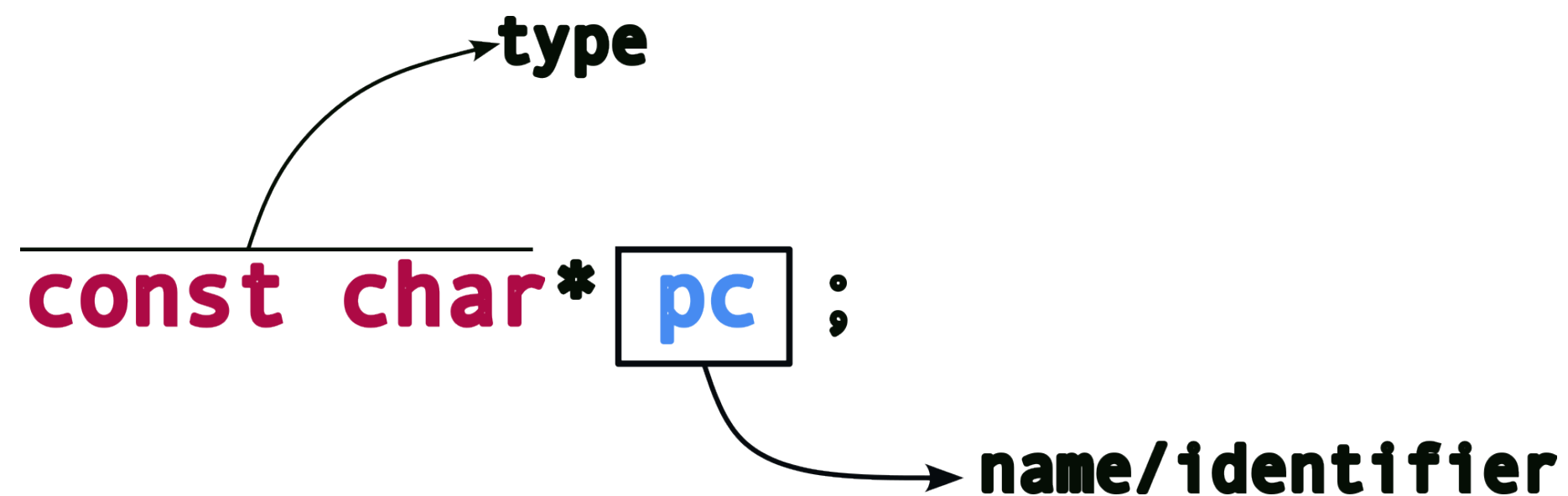
Declaration

Associates a **name** with a **type**

Definition

Declaration + allocate resources,
e.g., reserves memory or defines
a function body

- Most creation of identifiers are definition, e.g.,


The diagram shows the declaration `const char* pc;`. An arrow points from the text `const char*` to the word **type**. Another arrow points from the identifier `pc` (which is enclosed in a box) to the text **name/identifier**.



Why does it need to separate declarations and definitions?

Declaration or Definition?

Statement	Type
<code>int i;</code>	definition
<code>double * pd;</code>	definition
<code>char name[128];</code>	definition
<code>struct student *student_records[100];</code>	definition

- But what about this:

```
// declaration or definition?  
double foo(double temp);
```

- It is a declaration of a function.
 - Declares a function that takes a double argument and returns a double.
 - Also known as **function signature**.

Function Definition

```
// declaration or definition?  
double foo(double temp); {  
    // code here -- the "resources" for function foo()  
    // hence, this is a definition!  
}
```

- The “resource” allocated here is *code*.

Local Scope

- The identifier/name is only valid from its declaration/definition to its enclosing **block**, delimited by curly braces {...}.

```
void foo(double a) {  
    double temperature = a;  
    ...  
}
```

- Variable name temperature is only valid inside function foo().

```
void bar() {  
    temperature = 200;  
    ...  
}
```

This will fail compilation!

Local Scope

```
double foo(double temperature*) {  
    for( unsigned int i = 0 ; i < 10 ; ++i ) {  
        double fahrenheit = (temperature[i] * (9/5) + 32 ) ;  
        printf( "temperature = %f\t in fahrenheit = %f\n", temperature, fahrenheit );  
    }  
}
```

- Variable `fahrenheit` is *local* to the for block.

```
double foo(double temperature*) {  
    for( unsigned int i = 0 ; i < 10 ; ++i ) {  
        double fahrenheit = (temperature * (9/5) + 32 ) ;  
        printf( "temperature = %f\t in fahrenheit = %f\n", temperature, fahrenheit );  
    }  
    fahrenheit = 98.5 ;  
}
```

- What about `i` in the example?

Local Scope

- A *block* can mean many things
 - function
 - loop (for, while)
 - conditional (if ... else)
 - even just a pair of braces {...}
- Can have the same names in different scopes.

```
int a = 100 ;
```

```
if(...) {
```

```
    int b = a*100 ;
```

```
    int a = b ;
```

```
    printf( "a inside if = %d\n", a ) ;
```

```
}
```

```
printf( "a outside if = %d\n", a ) ;
```

A new *a* shadows the old *a*.



Local Scope

- A *block* can mean many things
 - function
 - loop (for, while)
 - conditional (if ... else)
 - even just a pair of braces {...}
- Can even have a standalone block.

```
{  
    int a = 9038 ;  
    {  
        int a = 999999 ;  
    }  
}
```

Global Scope

- Things defined *outside* of any scope, i.e., outside of any function or block.

```
double celsius_to_fahrenheit = 9.0/5.0 ;

double foo( double a ) {
    double temperature = a ;

    for(unsigned int i = 0 ; i < 10 ; ++i ) {
        double fahrenheit = (temperature * celsius_to_fahrenheit) + 32 ; // using the global
        char name[124] ;
    }
    ...
}

void bar() {
    // can CHANGE the global ANYWHERE!
    celsius_to_fahrenheit = 5.0/9.0 ;
}
```

- `foo()` and `bar()` are also defined in the global scope.

Global Scope

- Global identifiers can be accessed in *other files*.
- For global variables, use the `extern` keyword to mean this variable is *defined* elsewhere.
- E.g., in file `scope1.c`, we have
 - ▶ `double pi = 3.14159;`
- In `scope2.c`, we have
 - ▶ `extern double pi;`
- Then in `scope2.c`, we can do

```
void bar( double radius ) {  
    // using the externally declared variable "pi"  
    double area = pi * radius * radius ;  
}
```

Global Scope

- Global identifiers can be accessed in *other files*
- For functions, *declare* the functions defined in another source file.
- E.g., in file `scope1.c`, we have
 - ▶ `double foo(double d) {... // function definition;}`
- In `scope2.c`, if we want to use `foo`, we can just declare it
 - ▶ `double foo(double d);`
- Or, declare it in a header file and include this header file.

Note, each global identifier can only be defined once!

What is Different From Java?

- No object-oriented programming. C is *procedural*.
- No function overloading
- No classes
- But C has `struct` instead, which is similar but also different.
- Compiled not interpreted
- No public/private
 - By default, everything is public.
- Pointers!!
 - Source of superpower but also root causes for disastrous bugs.

Declaration and Definition

```
double pi = 3.14159;
```

a name, pi

associated with a type, double

resources allocated: memory to store
a double

also an initialization

```
extern double pi;
```

a name, pi

associated with a type, double

defined elsewhere; no
additional memory allocated



What if we want an identifier global to its file but invisible to other files

Type Modifiers

- `unsigned`: variables that cannot be negative
- `signed`: signed variable. `char`, `int`, `long` all default to signed.
- `static`: this variable/function are only accessible inside its source file.
- `const`: an immutable value

File Scope

- Use the static keyword to restrict the identifier to its file scope.

```
// GLOBAL but only for US  
static double celsius_to_fahrenheit = 9.0/5.0 ;
```

- `celsius_to_fahrenheit` can only be accessed in the file that defines it.
- `static` can be used for both global variables and functions.