We C.
But do we C well?

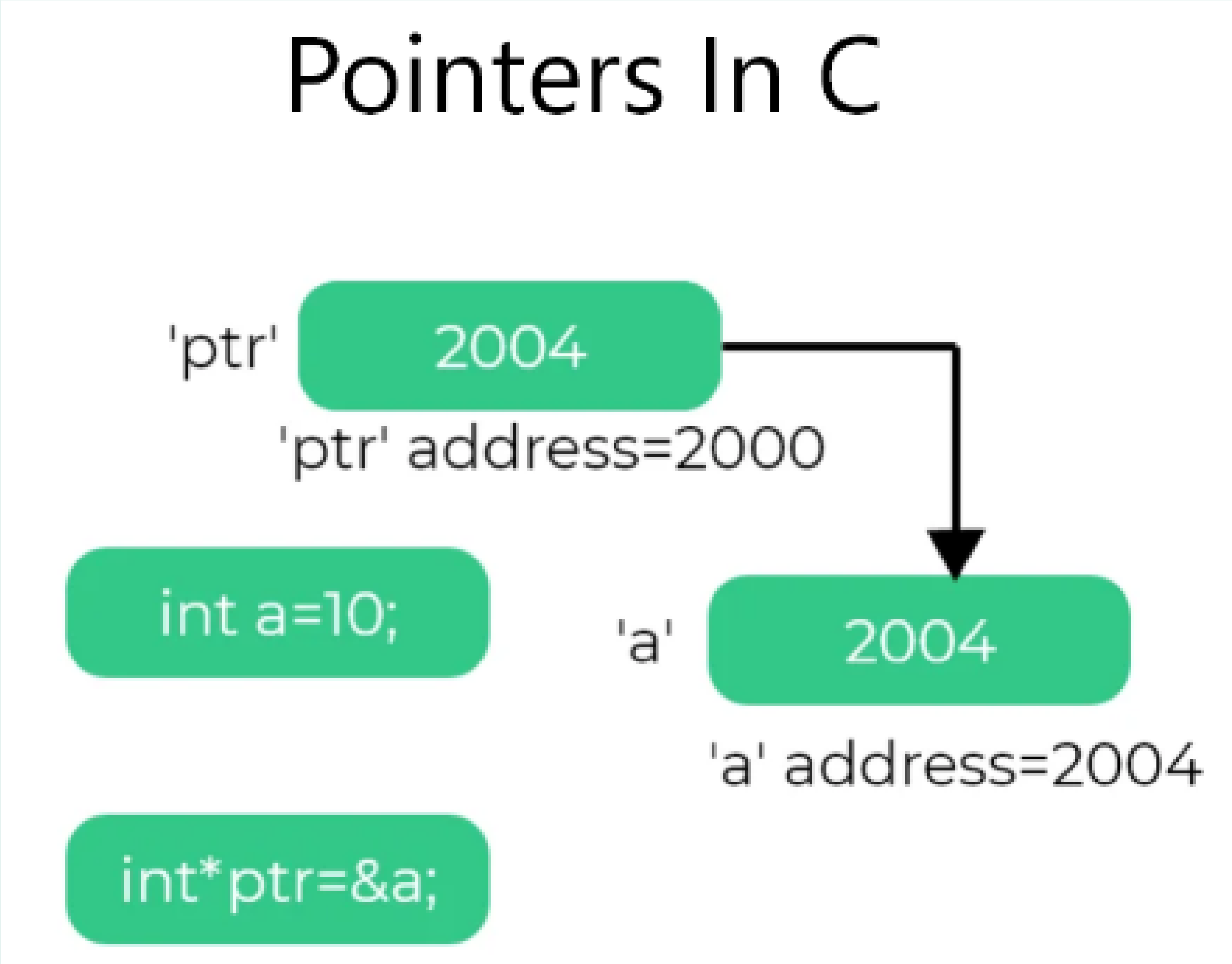
Neat C Techniques

Today's Menu

- Function Pointers
- Callback Functions
- Polymorphism with void pointers
- Discriminated Unions
- Switch vs Function Dispatching
- Coroutines
- Preprocessor and Macros

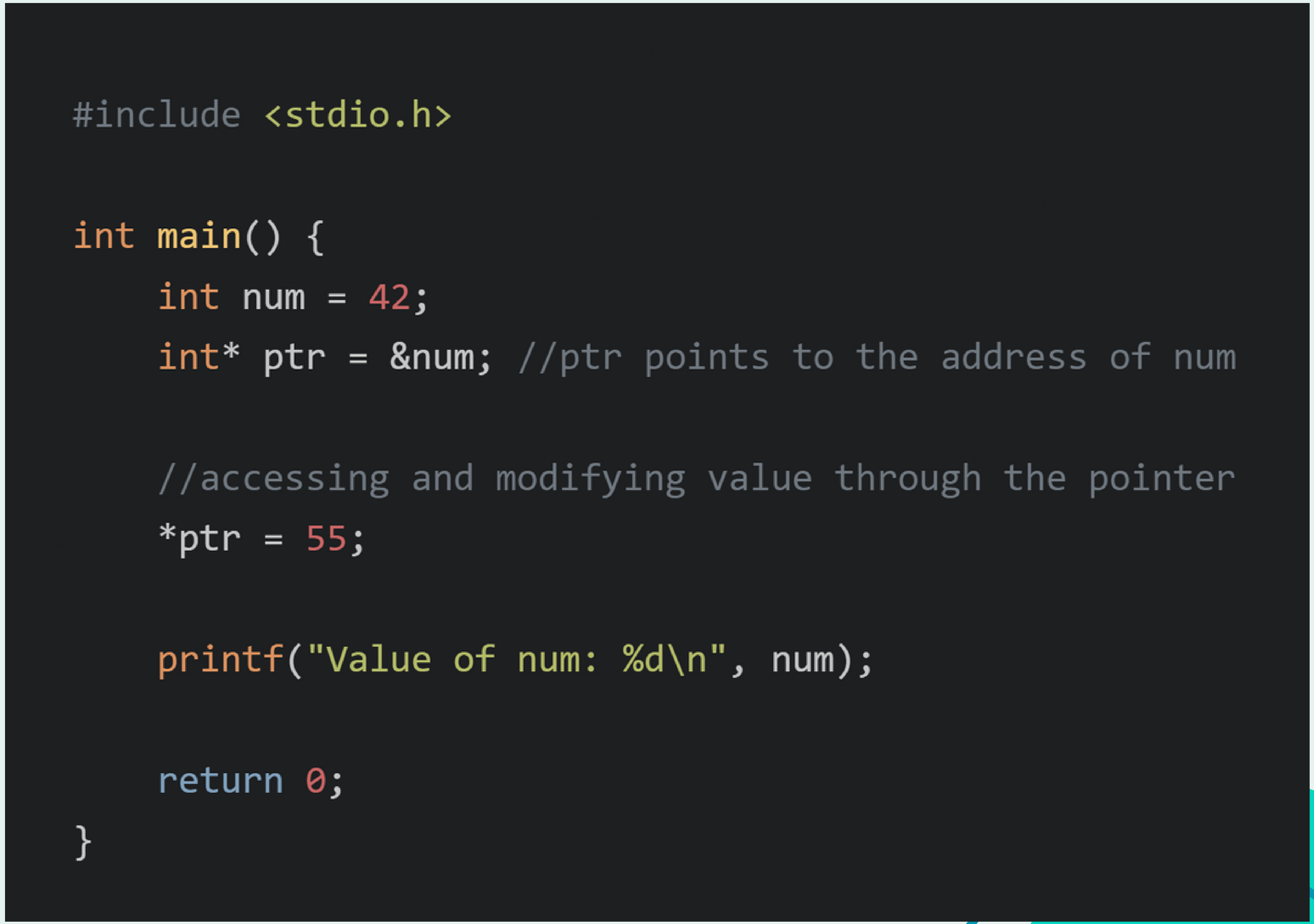
Before Function Pointers

- Regular pointers are variables that store memory addresses of variables.
- They point to specific locations in the computer's memory.



Pointers In C

```
int a=10;
int*ptr=&a;
```



```c
#include <stdio.h>

int main() {
    int num = 42;
    int* ptr = &num; //ptr points to the address of num

    //accessing and modifying value through the pointer
    *ptr = 55;

    printf("Value of num: %d\n", num);

    return 0;
}
```

Function Pointers

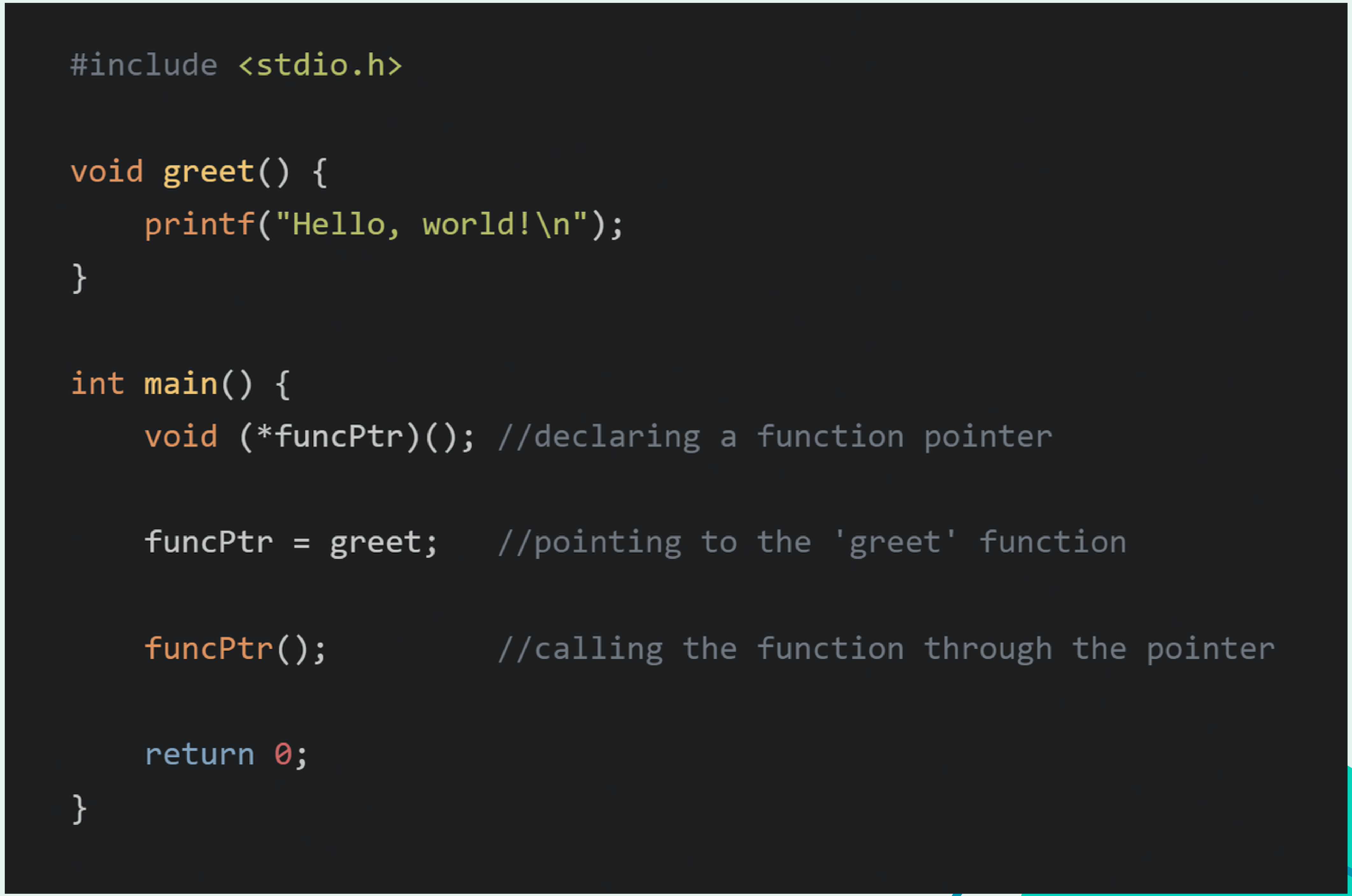
- Function pointers are variables that store the address of functions.
- Useful to pass functions as arguments to other functions, and return functions from subroutine.
- They let your program make choices on the fly, making it more adaptable and versatile.

Functions as arguments??

Function Pointers

- To use function pointers, we declare them just like regular pointers. It's like saying, 'Hey, I have a friend named funcPtr, and it knows the address of a function.’

```
//Declaring a function pointer that can point to a function that takes no arguments.
//The empty parentheses denote the absence of arguments.
     void (*funcPtr)();
```



```c
#include <stdio.h>

void greet() {
    printf("Hello, world!\n");
}

int main() {
    void (*funcPtr)(); //declaring a function pointer

    funcPtr = greet;   //pointing to the 'greet' function

    funcPtr();         //calling the function through the pointer

    return 0;
}
```

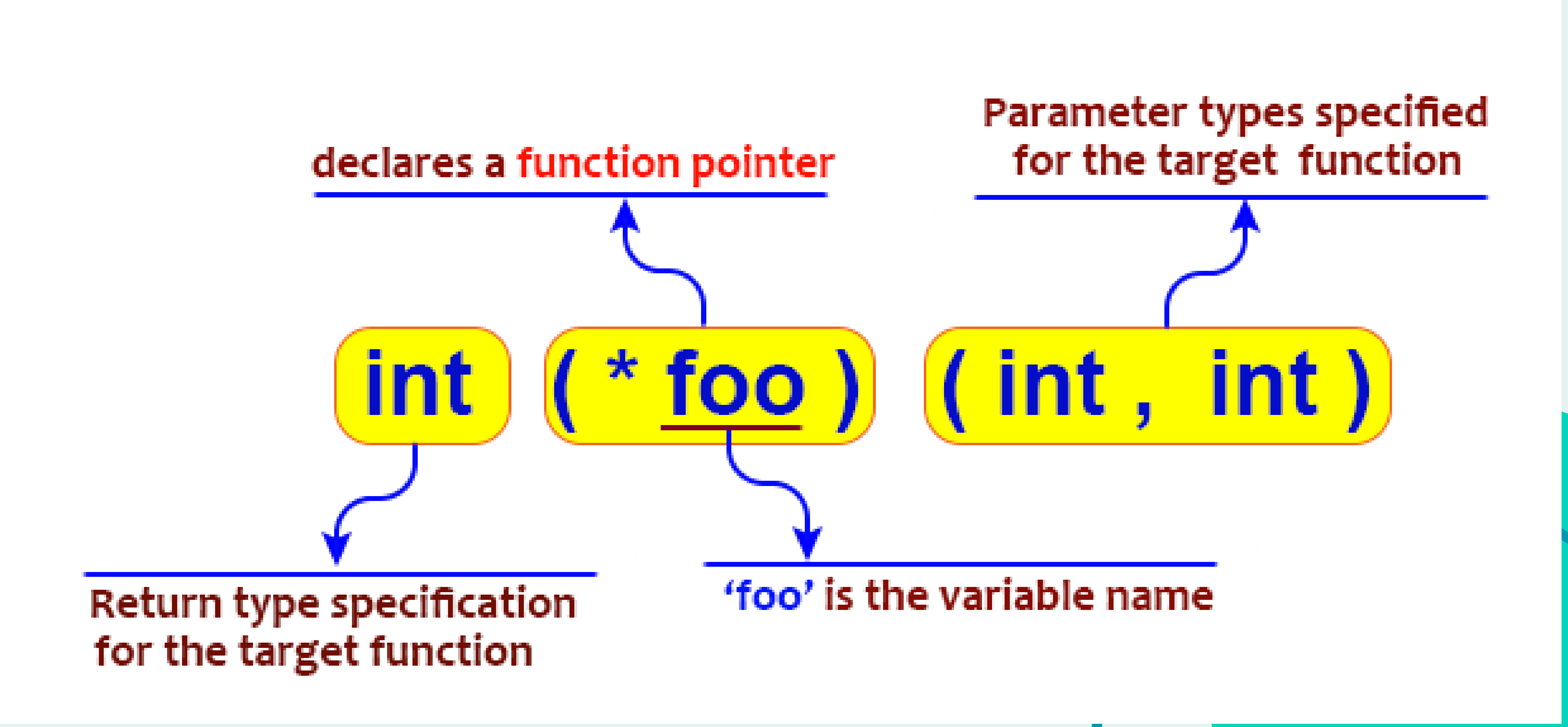
WARNING

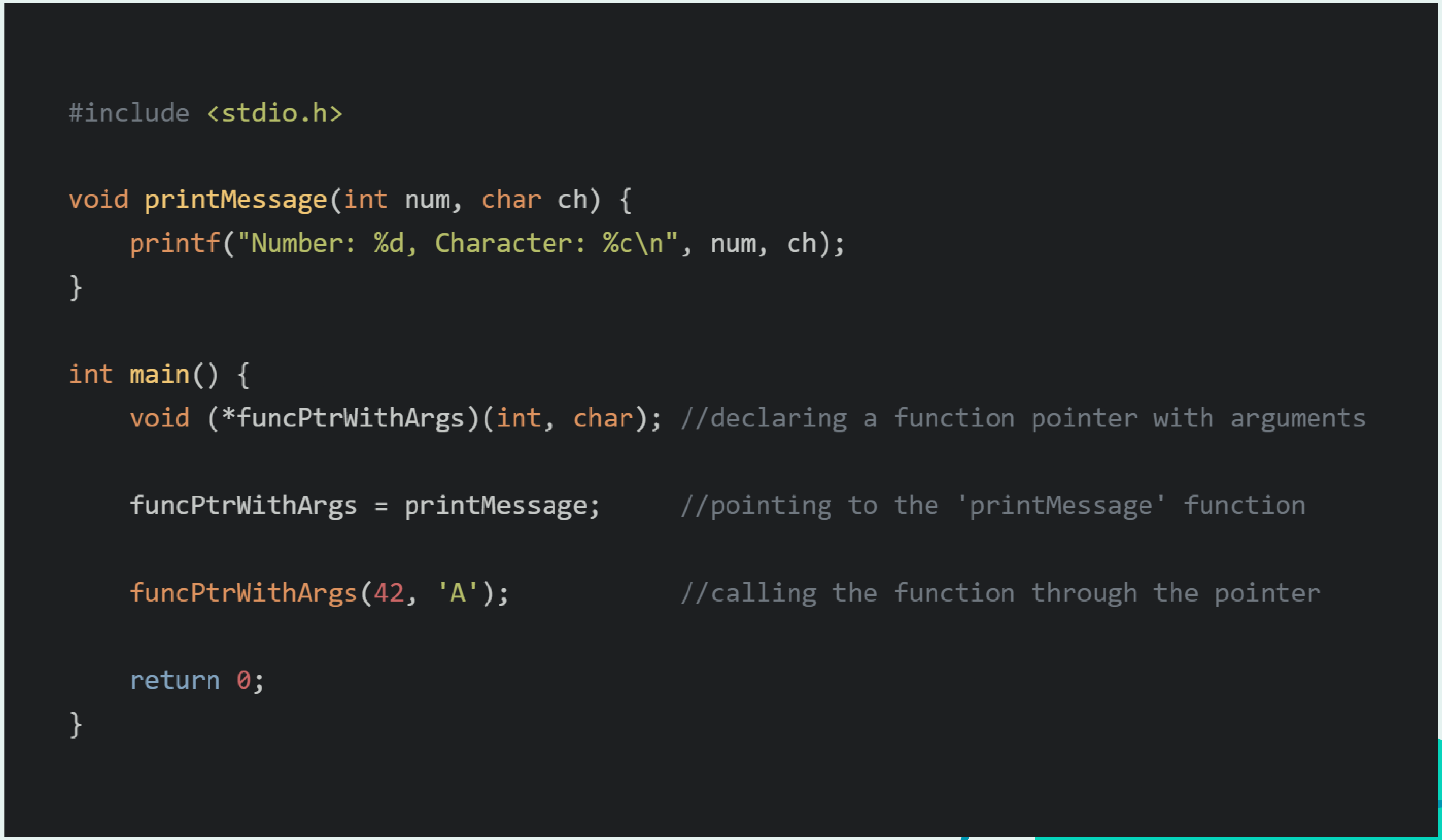
The declaration of a function pointer should have the pointer name in the **(parentheses)** to clarify that it is a pointer to a function rather than a regular function declaration.

```
int * ptr(int, int) //declaration of function named ptr returning an int*

int (*ptr)(int, int) //declaration of function pointer
```

Function Pointers with Arguments

```
//Declaring a function pointer that can point to a function that takes arguments.
//The empty parentheses denote the absence of arguments.
    void (*funcPtrWithArgs)(int, char);
```

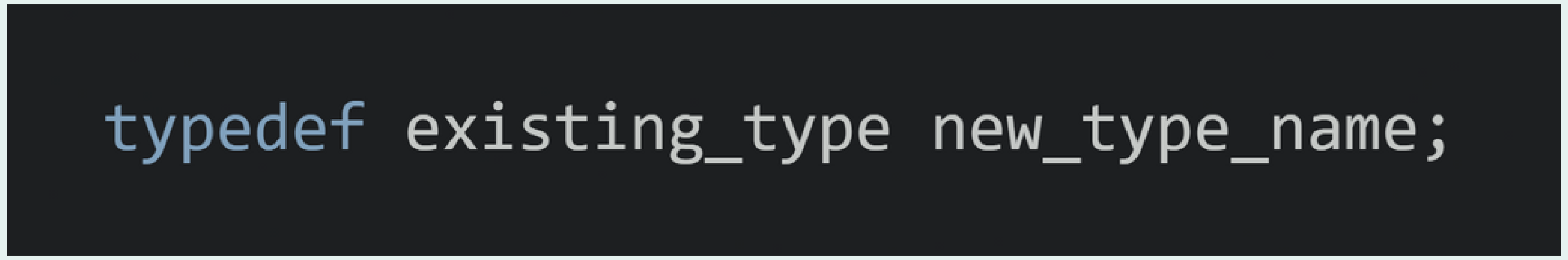

declares a function pointer
Parameter types specified for the target function
int
(* foo)
(int , int)
Return type specification for the target function
'foo' is the variable name



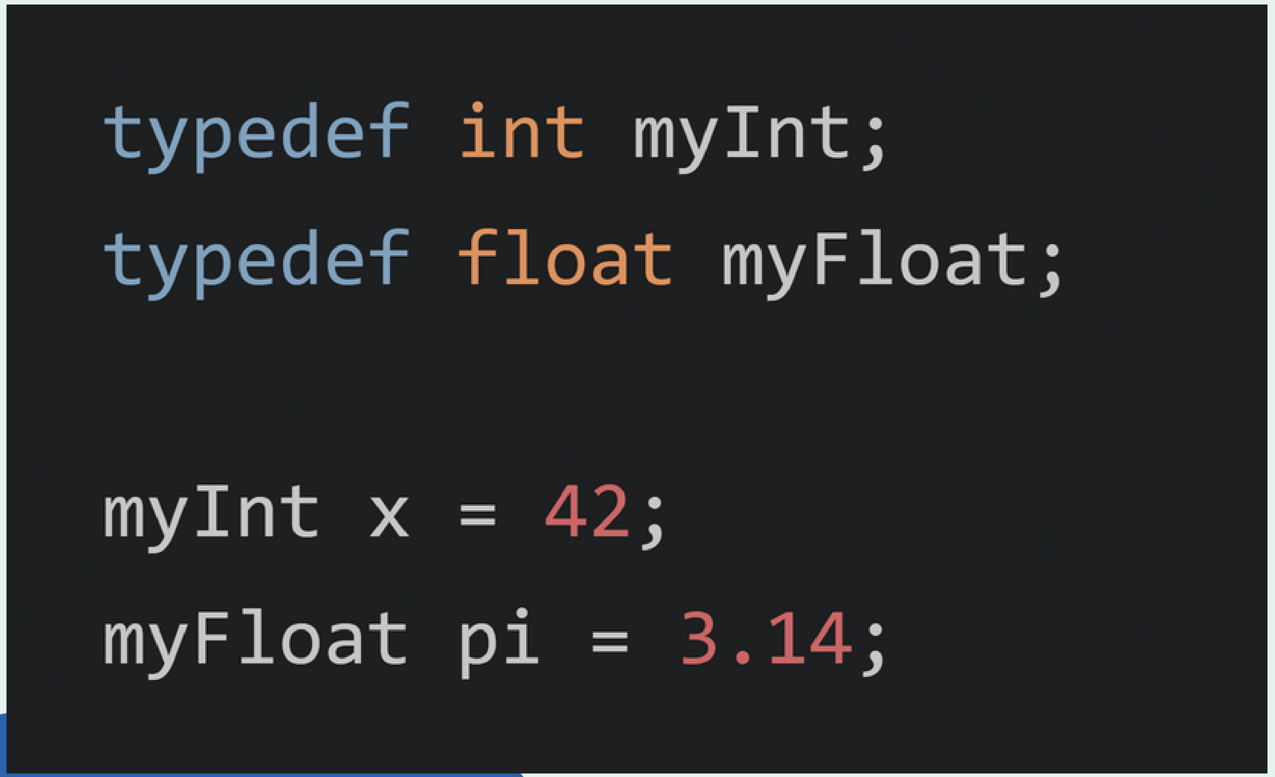
```c
#include <stdio.h>

void printMessage(int num, char ch) {
    printf("Number: %d, Character: %c\n", num, ch);
}

int main() {
    void (*funcPtrWithArgs)(int, char); //declaring a function pointer with arguments

    funcPtrWithArgs = printMessage;     //pointing to the 'printMessage' function

    funcPtrWithArgs(42, 'A');           //calling the function through the pointer

    return 0;
}
```

Recap of typedef in C

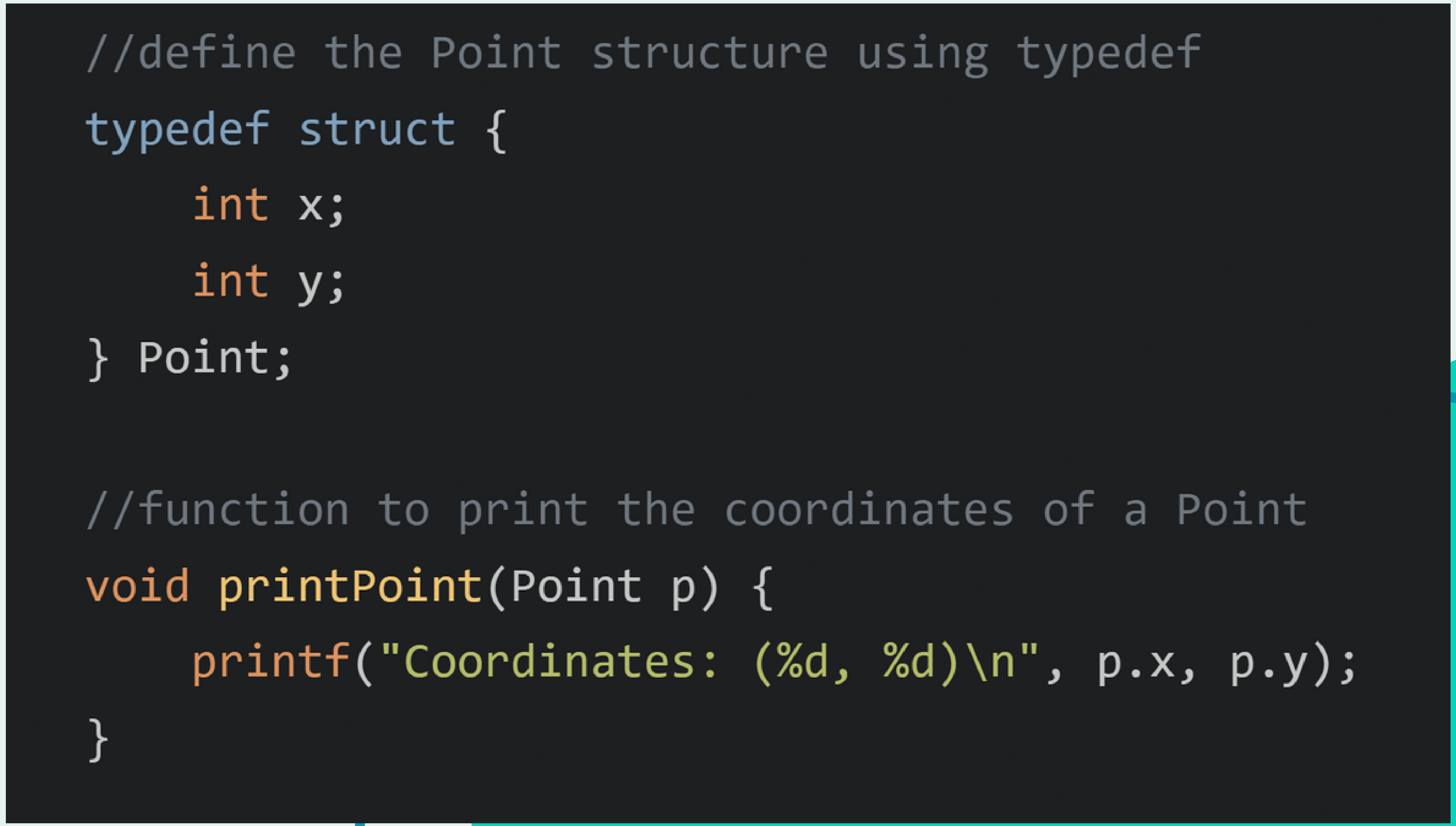
- C language feature for creating user-defined types.
- Provides a way to assign a new name to existing types.



```
typedef existing_type new_type_name;
```



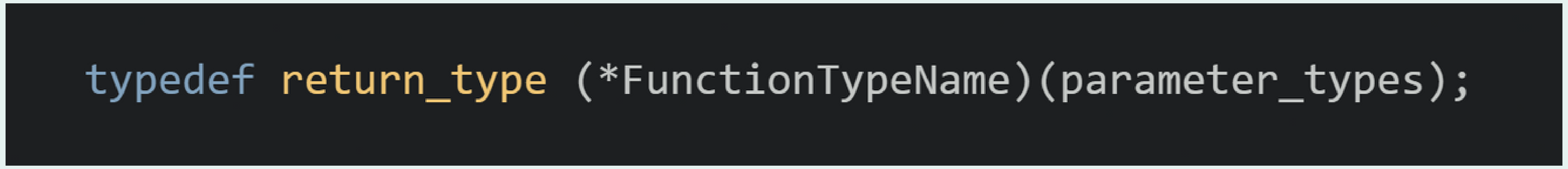
```
typedef int myInt;
typedef float myFloat;

myInt x = 42;
myFloat pi = 3.14;
```



```
//define the Point structure using typedef
typedef struct {
    int x;
    int y;
} Point;

//function to print the coordinates of a Point
void printPoint(Point p) {
    printf("Coordinates: (%d, %d)\n", p.x, p.y);
}
```

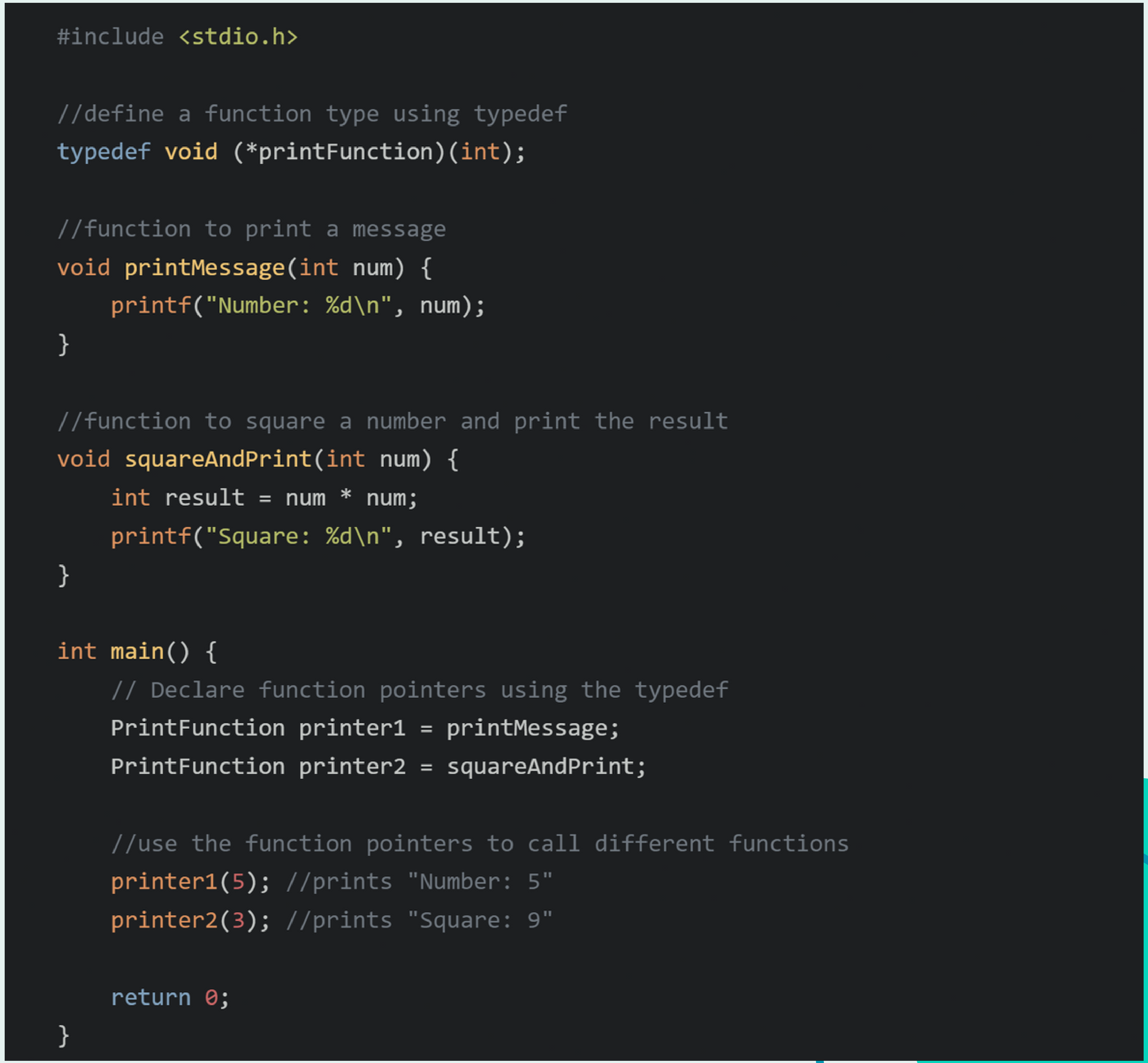
Typedef for function pointers

- Provides a meaningful name to a type associated with a specific function signature.



```
typedef return_type (*FunctionTypeName)(parameter_types);
```

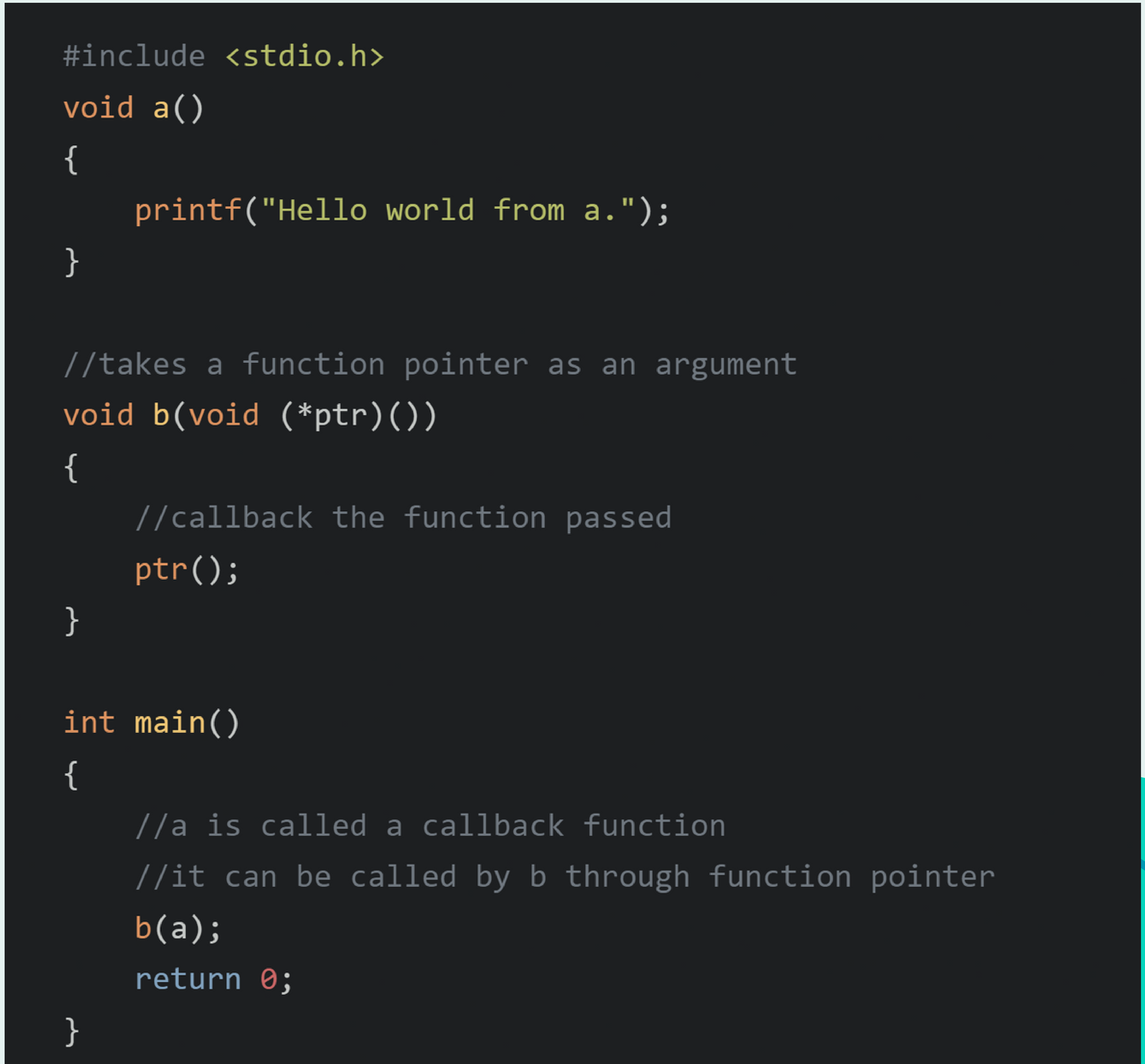
- The FunctionTypeName now becomes an alias for the function pointer type return_type (*)(parameter_types).



```c
#include <stdio.h>

//define a function type using typedef
typedef void (*printFunction)(int);

//function to print a message
void printMessage(int num) {
    printf("Number: %d\n", num);
}

//function to square a number and print the result
void squareAndPrint(int num) {
    int result = num * num;
    printf("Square: %d\n", result);
}

int main() {
    // Declare function pointers using the typedef
    PrintFunction printer1 = printMessage;
    PrintFunction printer2 = squareAndPrint;

    //use the function pointers to call different functions
    printer1(5); //prints "Number: 5"
    printer2(3); //prints "Square: 9"

    return 0;
}
```

Callbacks

- Function pointers can be passed as arguments to functions.
- A callback function is simply a function pointer that is passed to another function as a parameter. In most instances, a callback will contain three pieces.
  - The callback function
  - A callback registration
  - Callback execution

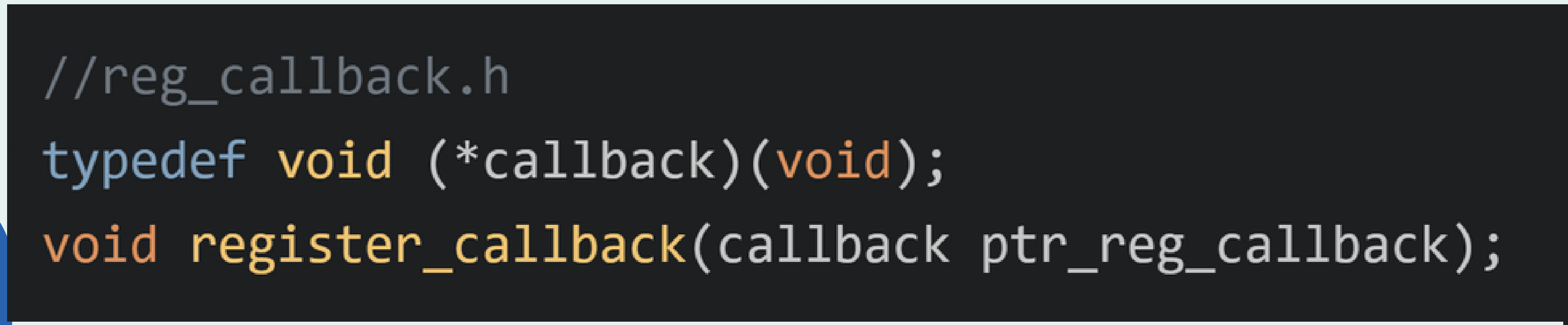


```c
#include <stdio.h>
void a()
{
    printf("Hello world from a.");
}

//takes a function pointer as an argument
void b(void (*ptr)())
{
    //callback the function passed
    ptr();
}

int main()
{
    //a is called a callback function
    //it can be called by b through function pointer
    b(a);
    return 0;
}
```

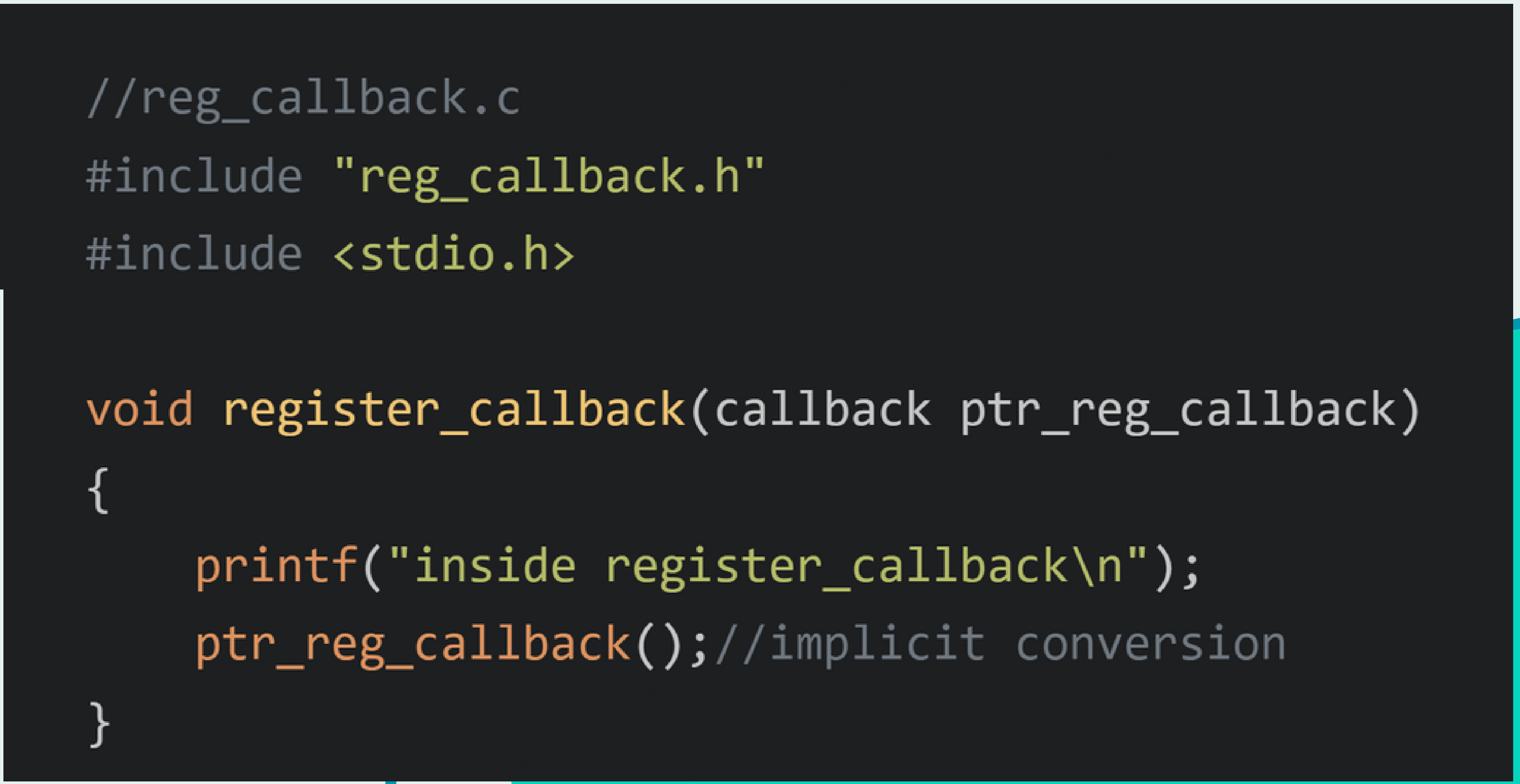
Callback function, Callback registration, and Callback execution

The callback mechanism allows the lower layer function to call the higher layer function through a pointer to a callback function.

An Example:



```c
//reg_callback.h
typedef void (*callback)(void);
void register_callback(callback ptr_reg_callback);
```



```c
//reg_callback.c
#include "reg_callback.h"
#include <stdio.h>

void register_callback(callback ptr_reg_callback)
{
    printf("inside register_callback\n");
    ptr_reg_callback();//implicit conversion
}
```

- When the lower layer function is executed, it performs its operations and, if a callback is registered, calls back to the higher layer by invoking the registered callback.

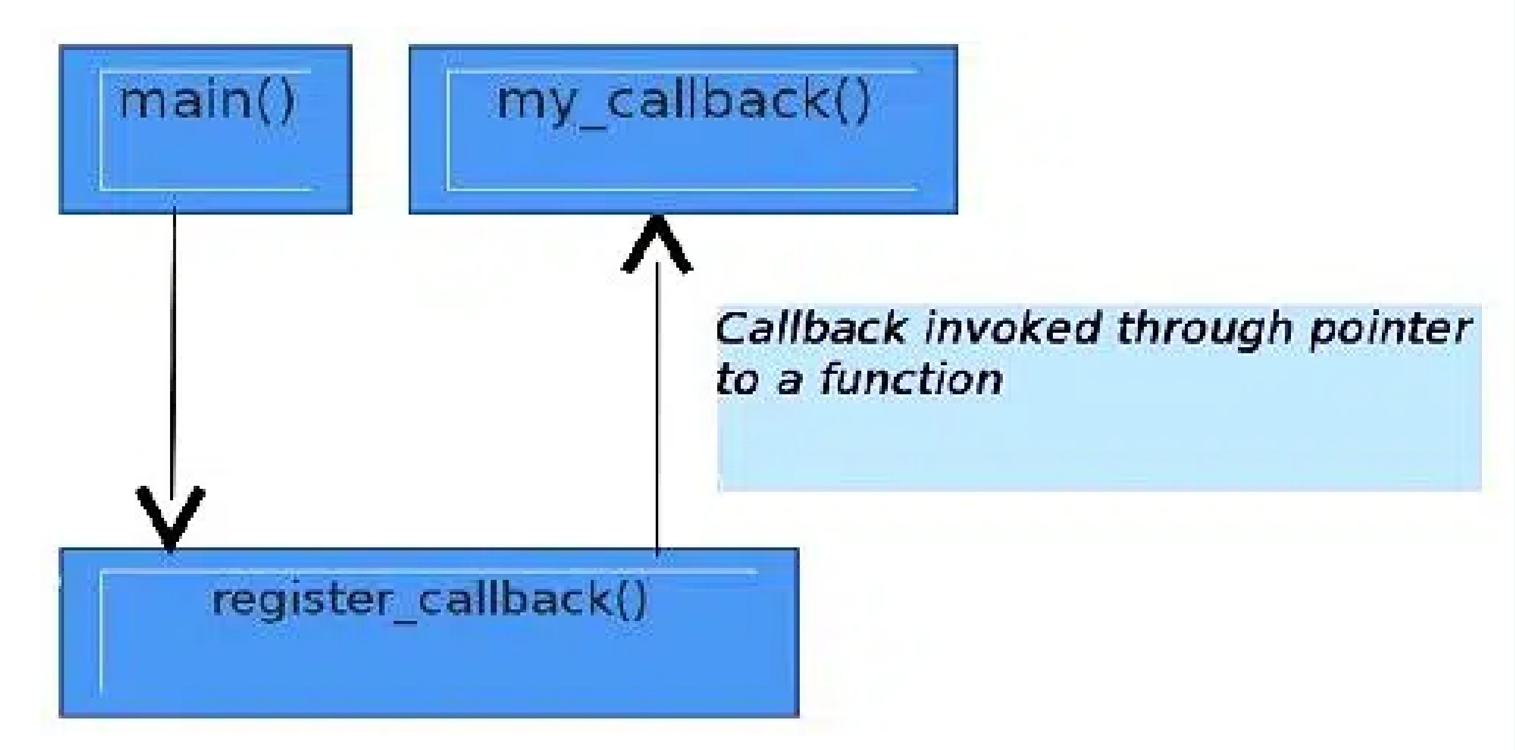
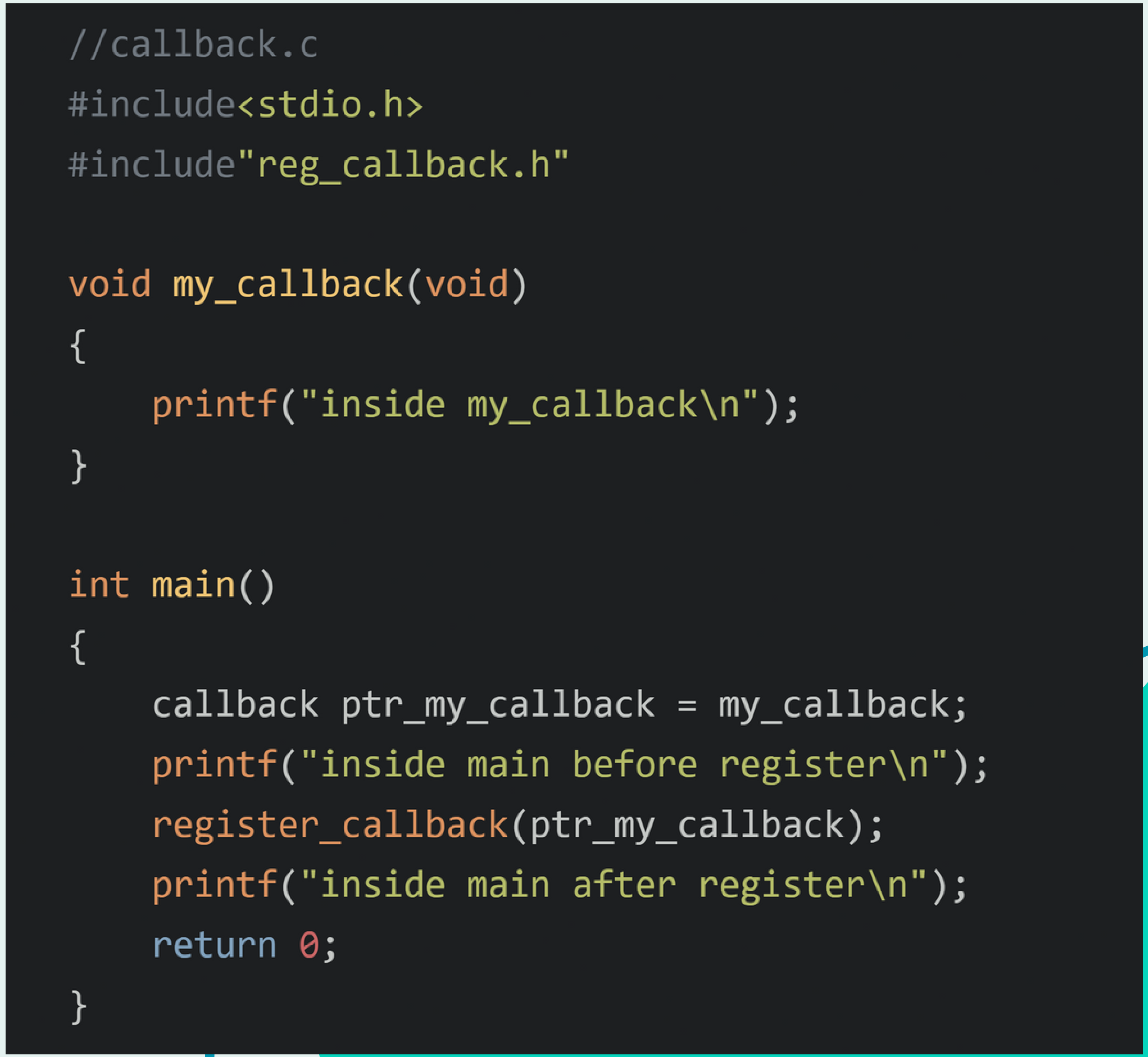
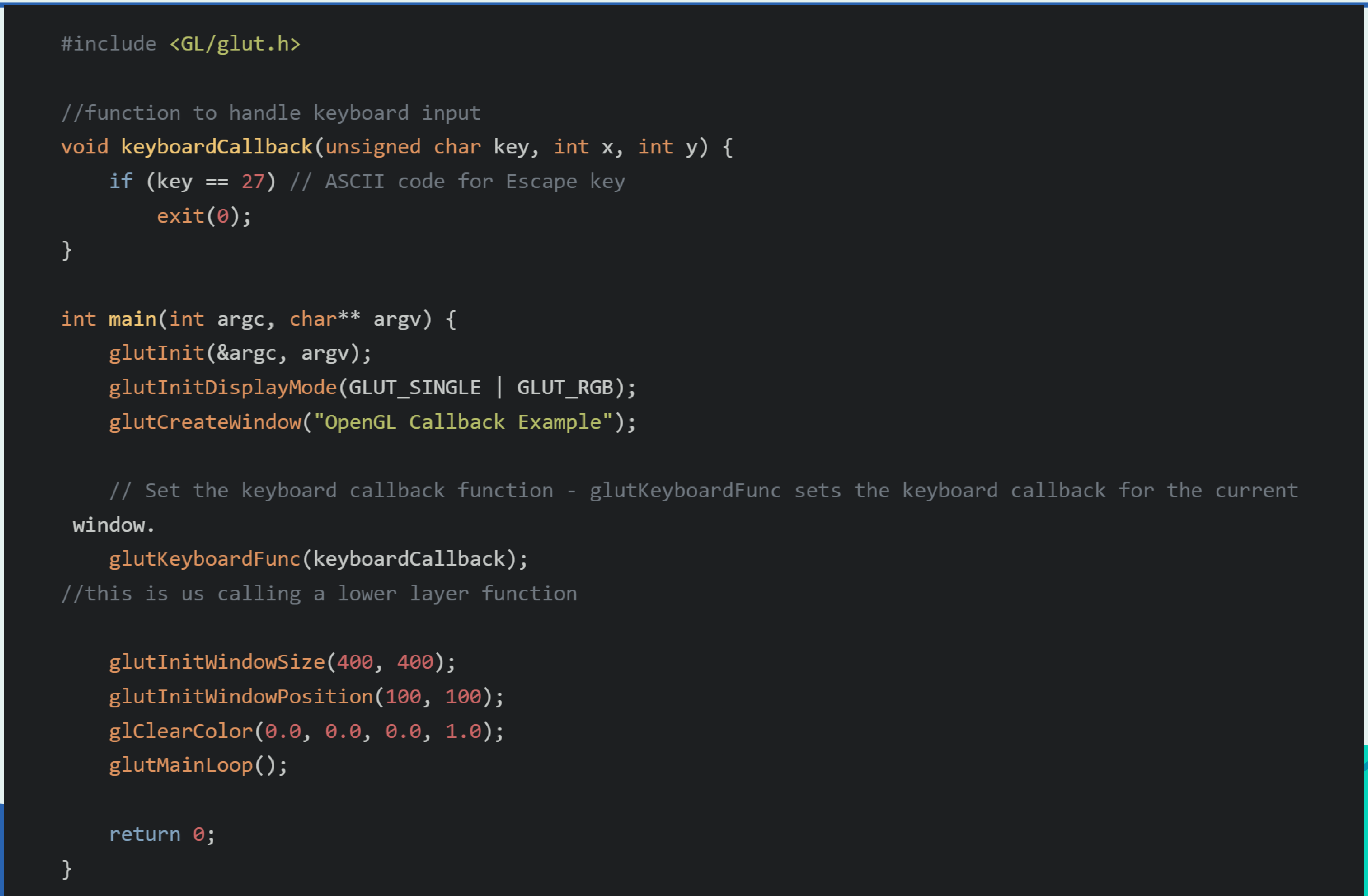


Figure 1: Program flow



```c
//callback.c
#include<stdio.h>
#include"reg_callback.h"

void my_callback(void)
{
    printf("inside my_callback\n");
}

int main()
{
    callback ptr_my_callback = my_callback;
    printf("inside main before register\n");
    register_callback(ptr_my_callback);
    printf("inside main after register\n");
    return 0;
}
```

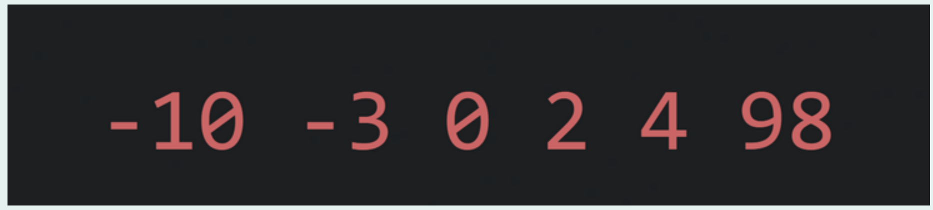


```c
#include <GL/glut.h>

//function to handle keyboard input
void keyboardCallback(unsigned char key, int x, int y) {
    if (key == 27) // ASCII code for Escape key
        exit(0);
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("OpenGL Callback Example");

    // Set the keyboard callback function - glutKeyboardFunc sets the keyboard callback for the current
 window.
    glutKeyboardFunc(keyboardCallback);
//this is us calling a lower layer function

    glutInitWindowSize(400, 400);
    glutInitWindowPosition(100, 100);
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glutMainLoop();

    return 0;
}
```

Uses of Callback Functions

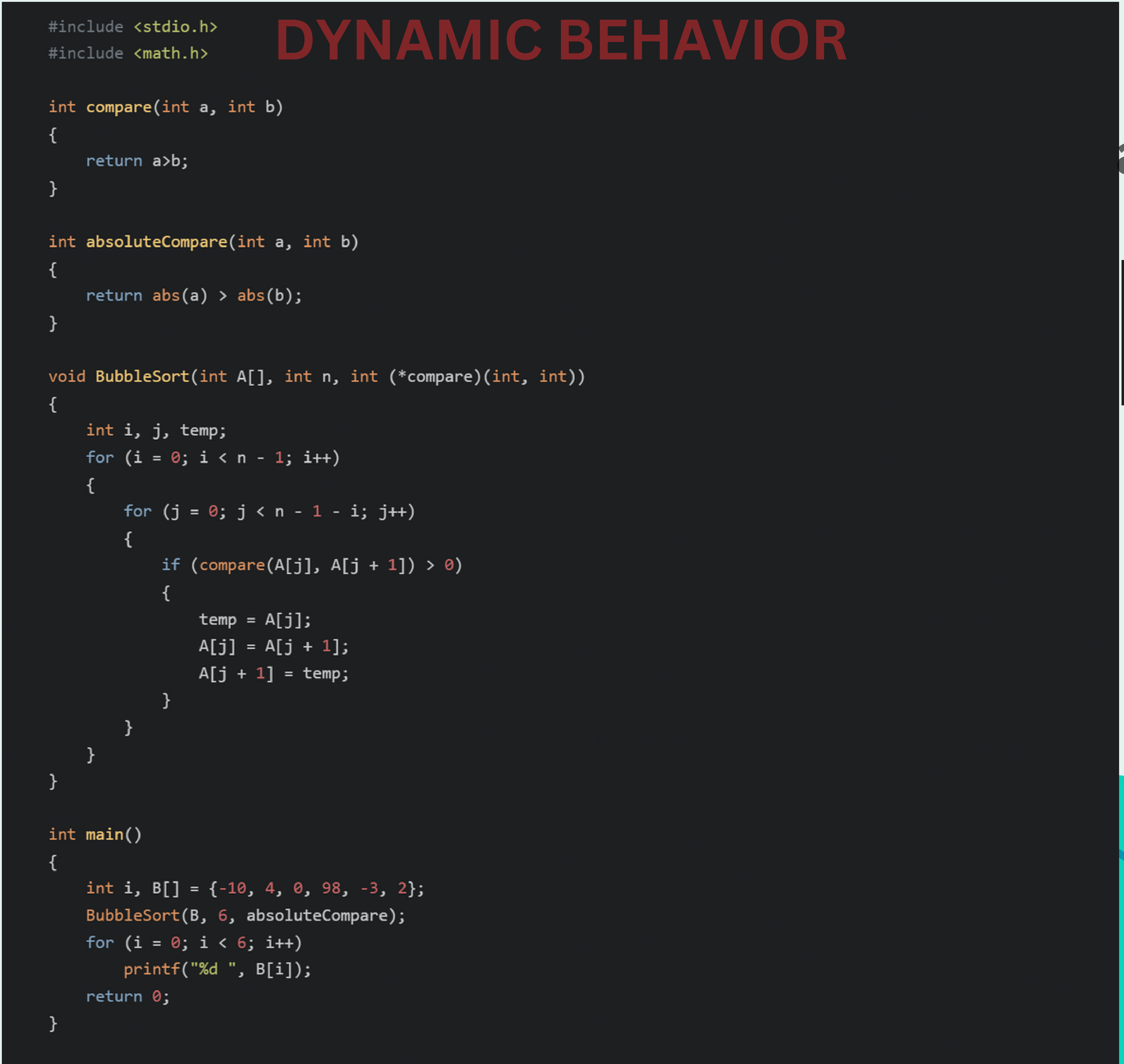
- **Event-driven Programming** (a button click might trigger a callback function)
- **Dynamic Behavior** (simply registering a different callback function can change the behavior of register_callback)
- **Modularity** (the function that accepts the callback doesn't need to know the callback's implementation)

With compare



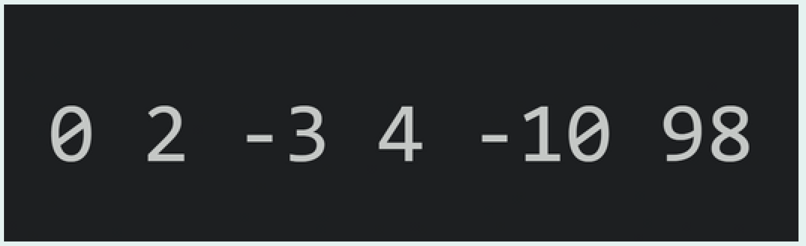
```
-10 -3 0 2 4 98
```

DYNAMIC BEHAVIOR

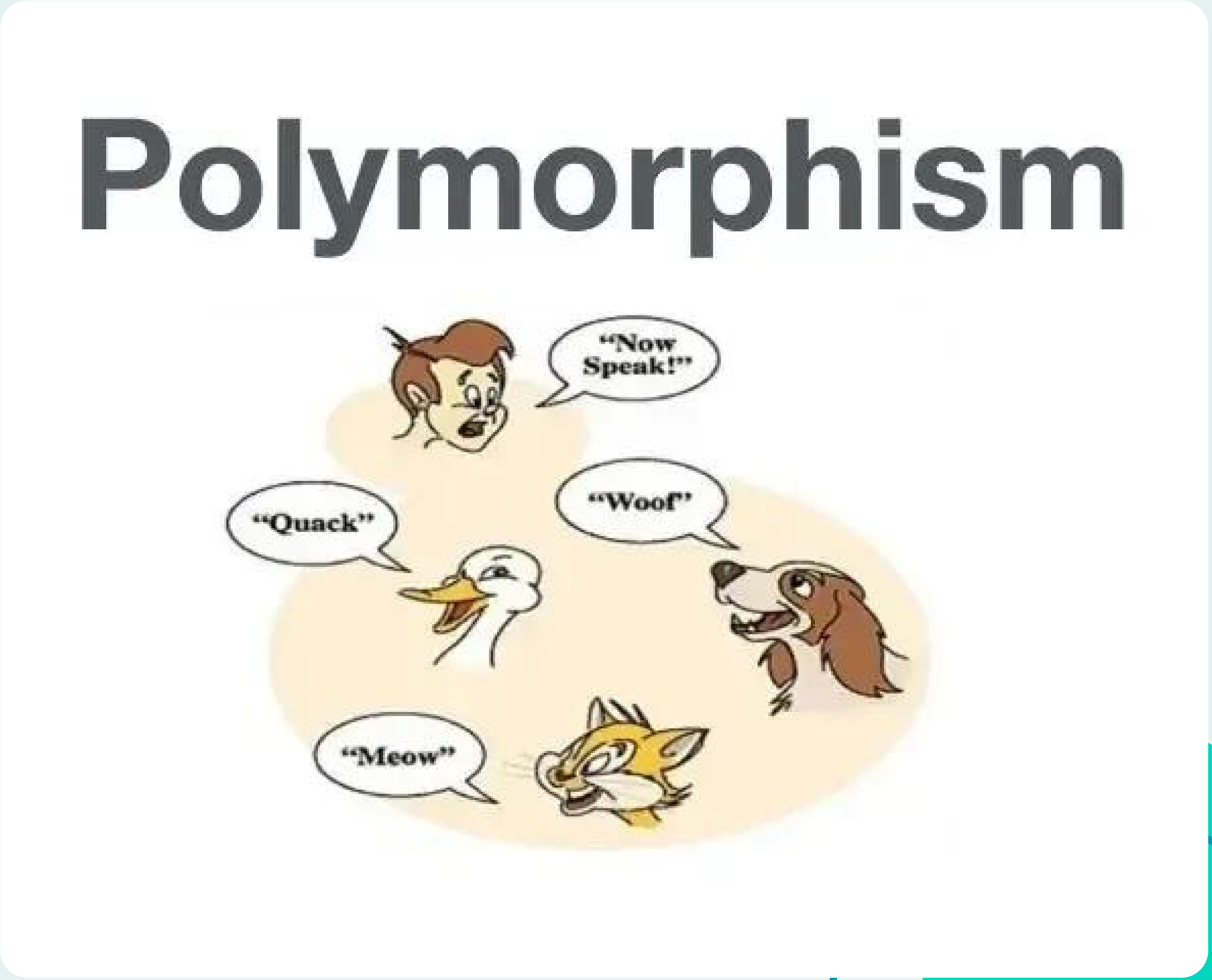


```c
#include <stdio.h>
#include <math.h>

int compare(int a, int b)
{
    return a>b;
}

int absoluteCompare(int a, int b)
{
    return abs(a) > abs(b);
}

void BubbleSort(int A[], int n, int (*compare)(int, int))
{
    int i, j, temp;
    for (i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n - 1 - i; j++)
        {
            if (compare(A[j], A[j + 1]) > 0)
            {
                temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
            }
        }
    }
}

int main()
{
    int i, B[] = {-10, 4, 0, 98, -3, 2};
    BubbleSort(B, 6, absoluteCompare);
    for (i = 0; i < 6; i++)
        printf("%d ", B[i]);
    return 0;
}
```

With absoluteCompare

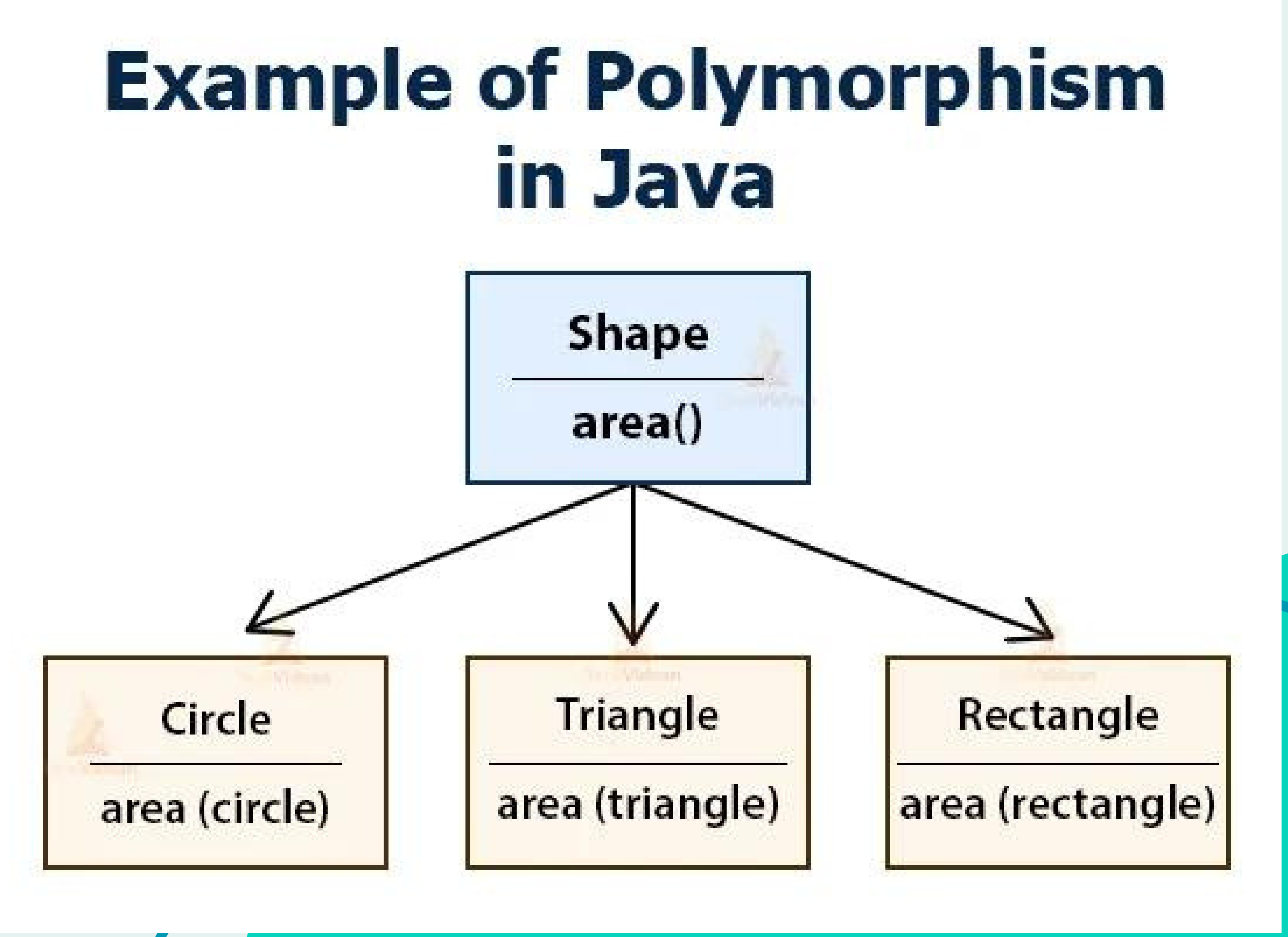


```
0 2 -3 4 -10 98
```


Polymorphism
"Now Speak!"
"Quack"
"Woof"
"Meow"

Polymorphism - "having multiple forms."

- A popular concept in object-oriented programming (OOP).
- A programming language can **present the same interface for several underlying data types and objects to respond uniquely to the same message.**



Example of Polymorphism in Java

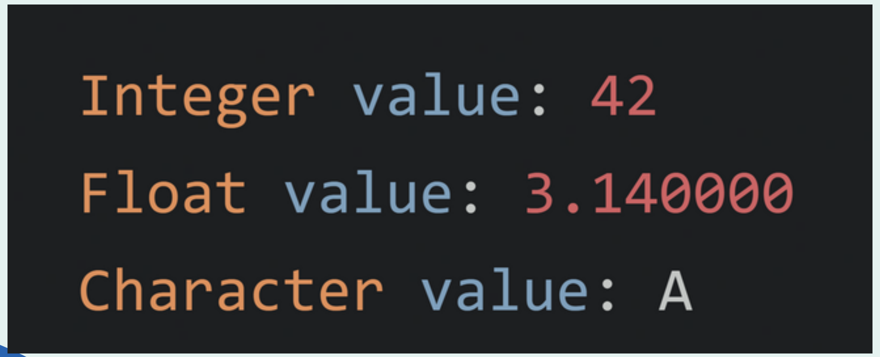
Polymorphic behavior in C

- Even though true polymorphism, as seen in languages like C++ and Java, is not directly supported in C, it can be achieved through the use of function pointers and void pointers.
- void* is a generic pointer type in C that can point to data of any type.
- int* or float* point to integers or floats respectively
- but void* is not tied to any specific type

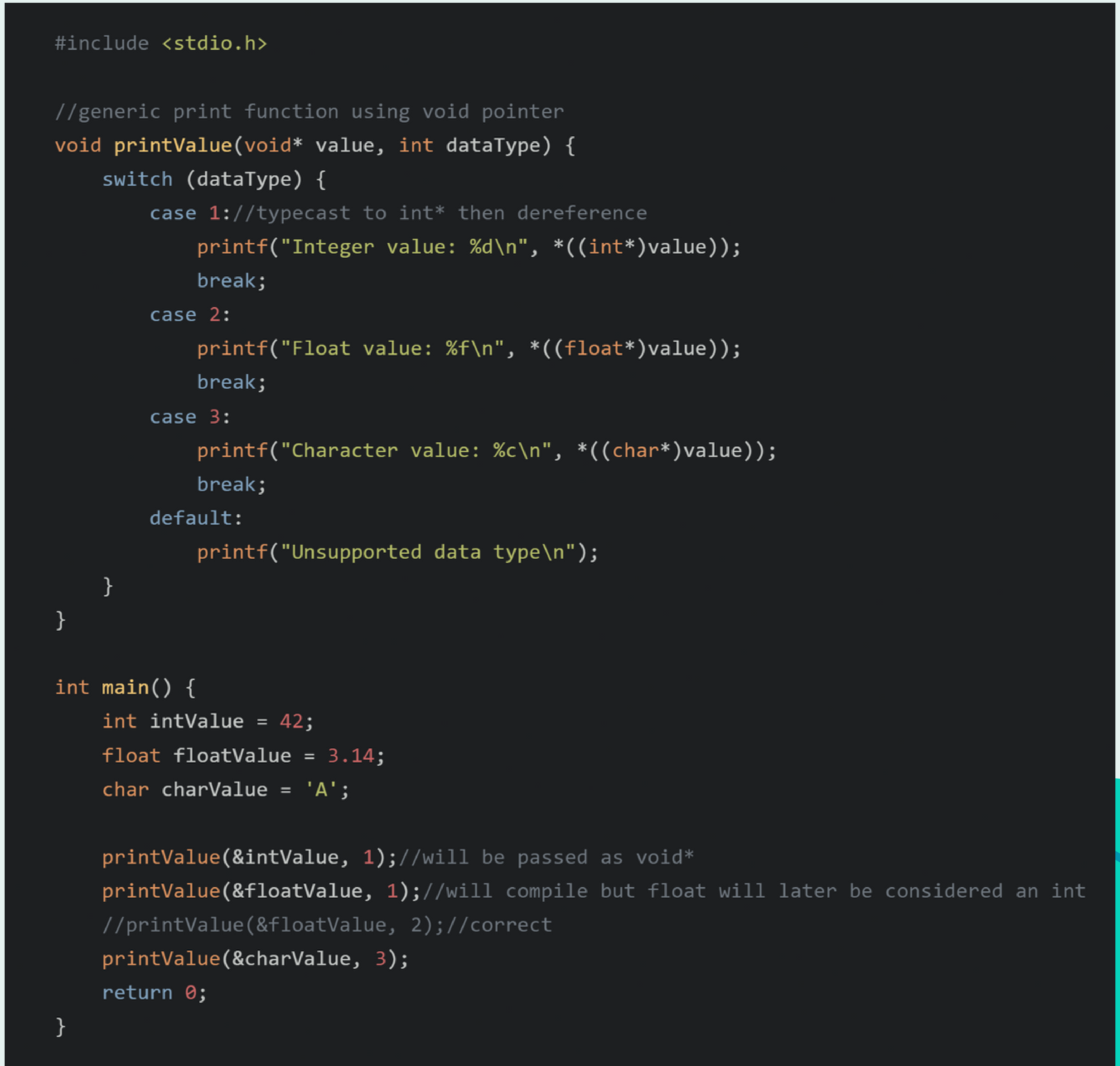
While void* offers flexibility, it also comes with the responsibility of ensuring type safety during usage.

BE CAREFUL

Correctly handled void*

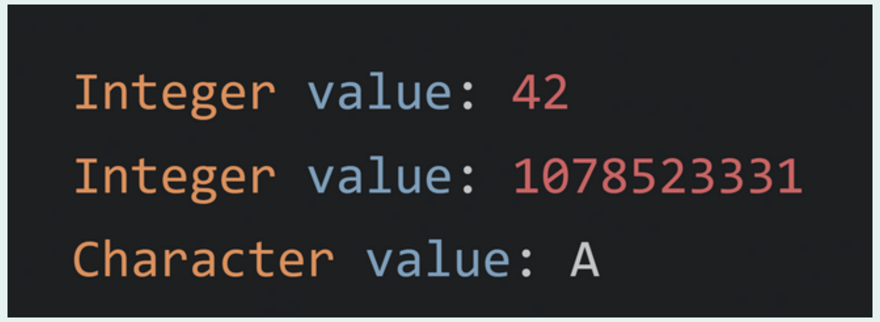


```
Integer value: 42
Float value: 3.140000
Character value: A
```

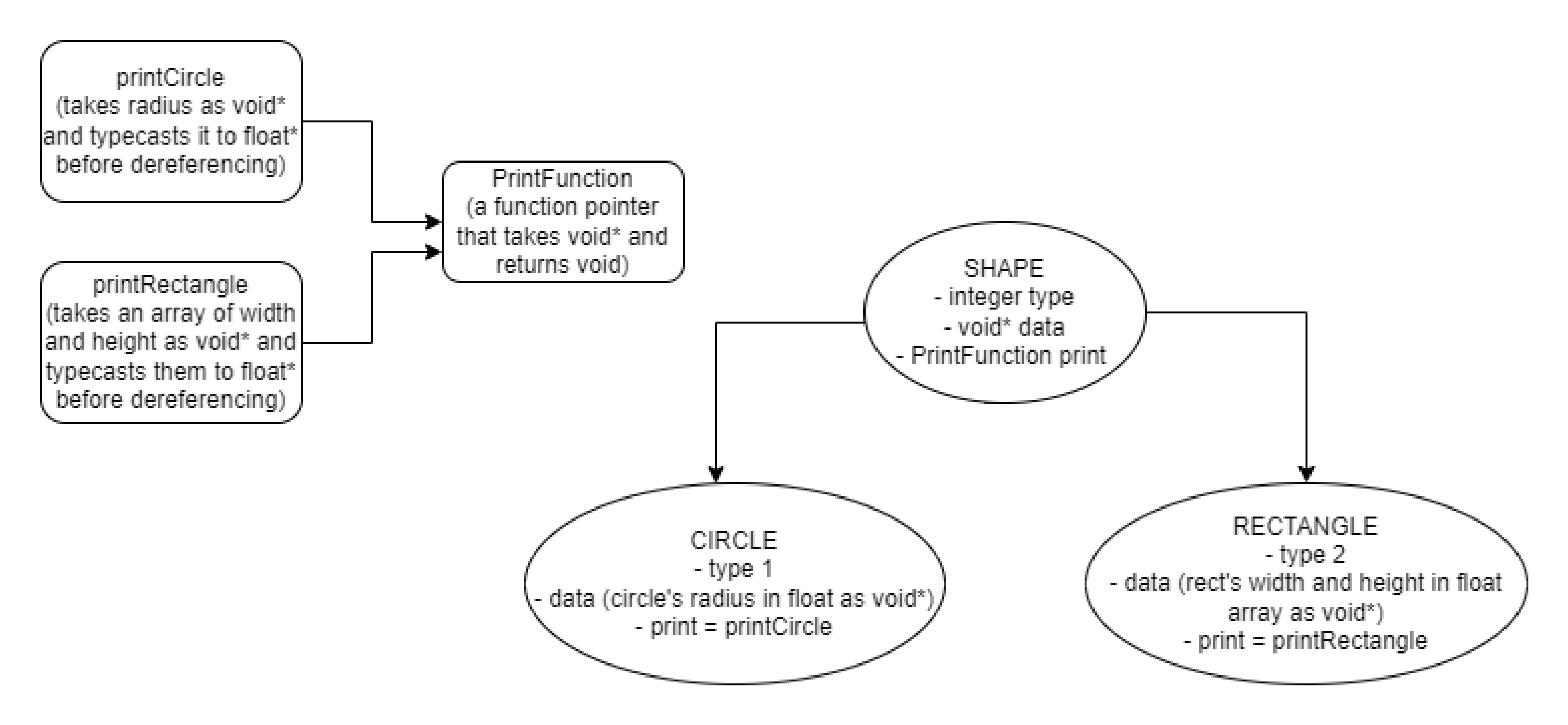


```
#include <stdio.h>

//generic print function using void pointer
void printValue(void* value, int dataType) {
    switch (dataType) {
        case 1://typecast to int* then dereference
            printf("Integer value: %d\n", *((int*)value));
            break;
        case 2:
            printf("Float value: %f\n", *((float*)value));
            break;
        case 3:
            printf("Character value: %c\n", *((char*)value));
            break;
        default:
            printf("Unsupported data type\n");
    }
}

int main() {
    int intValue = 42;
    float floatValue = 3.14;
    char charValue = 'A';

    printValue(&intValue, 1);//will be passed as void*
    printValue(&floatValue, 1);//will compile but float will later be considered an int
    //printValue(&floatValue, 2);//correct
    printValue(&charValue, 3);
    return 0;
}
```

Incorrectly handled void*



```
Integer value: 42
Integer value: 1078523331
Character value: A
```



printCircle
(takes radius as void* and typecasts it to float* before dereferencing)

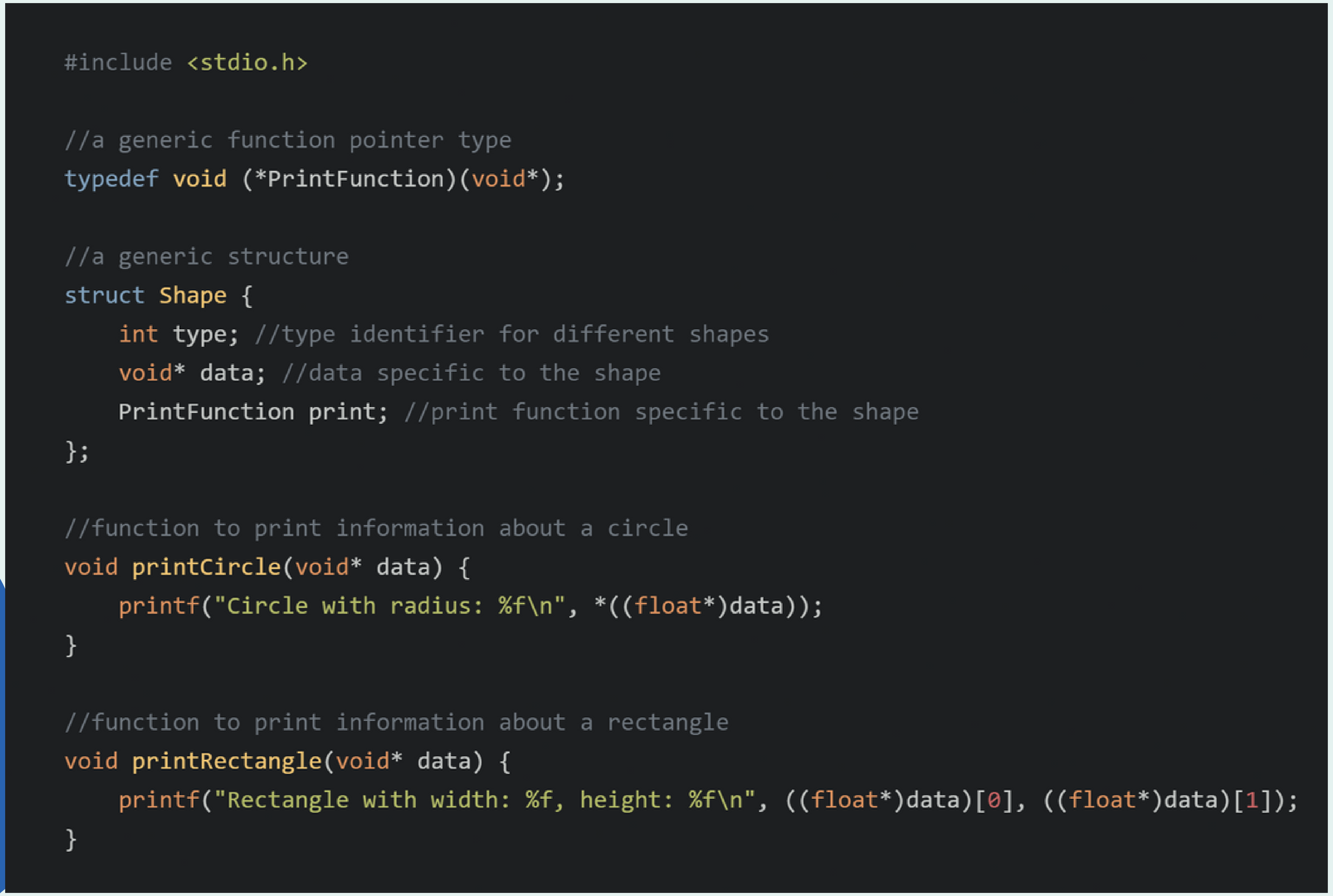
printRectangle
(takes an array of width and height as void* and typecasts them to float* before dereferencing)

PrintFunction
(a function pointer that takes void* and returns void)

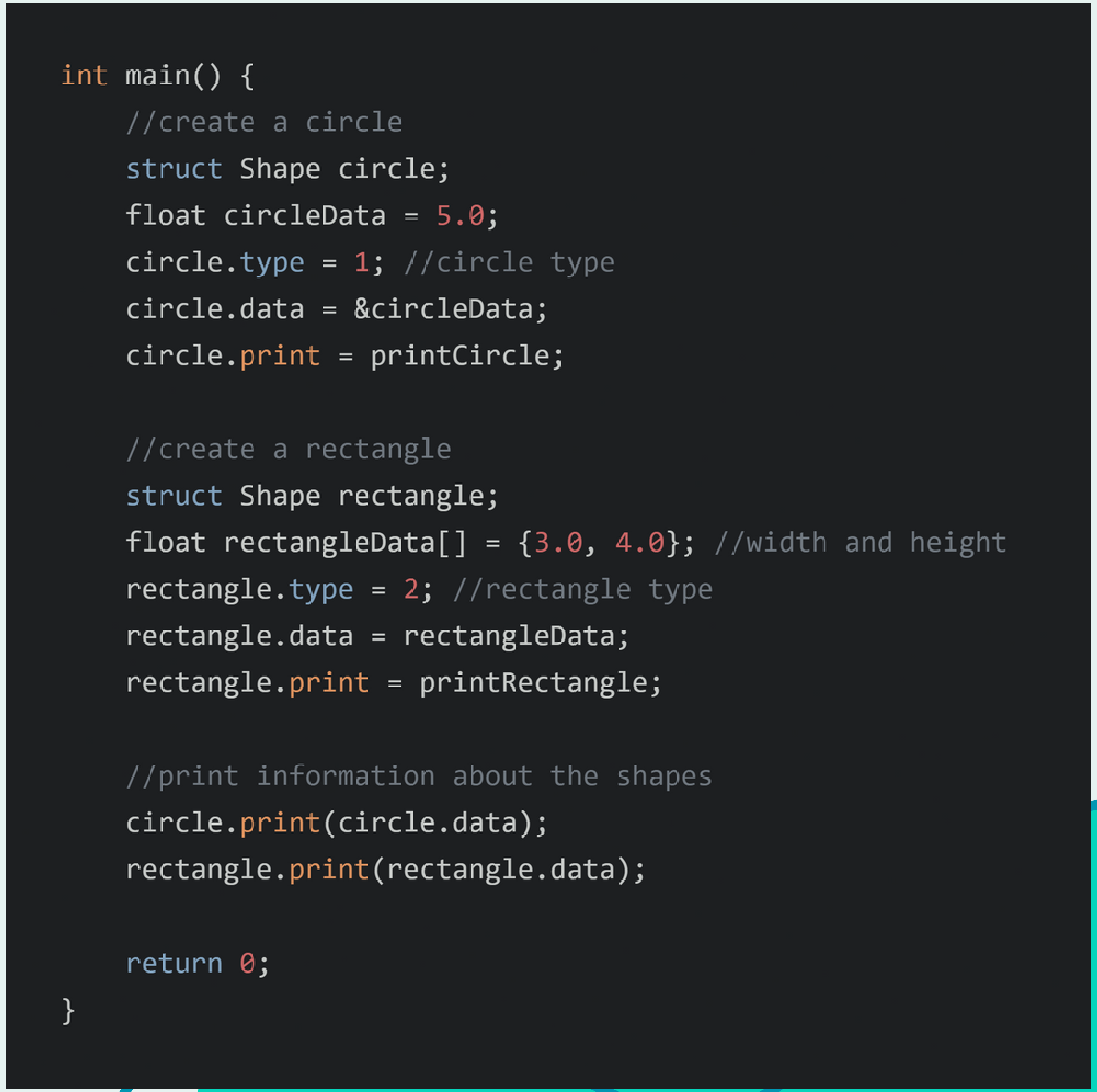
SHAPE
- integer type
- void* data
- PrintFunction print

CIRCLE
- type 1
- data (circle's radius in float as void*)
- print = printCircle

RECTANGLE
- type 2
- data (rect's width and height in float array as void*)
- print = printRectangle



```c
#include <stdio.h>

//a generic function pointer type
typedef void (*PrintFunction)(void*);

//a generic structure
struct Shape {
    int type; //type identifier for different shapes
    void* data; //data specific to the shape
    PrintFunction print; //print function specific to the shape
};

//function to print information about a circle
void printCircle(void* data) {
    printf("Circle with radius: %f\n", *((float*)data));
}

//function to print information about a rectangle
void printRectangle(void* data) {
    printf("Rectangle with width: %f, height: %f\n", ((float*)data)[0], ((float*)data)[1]);
}
```



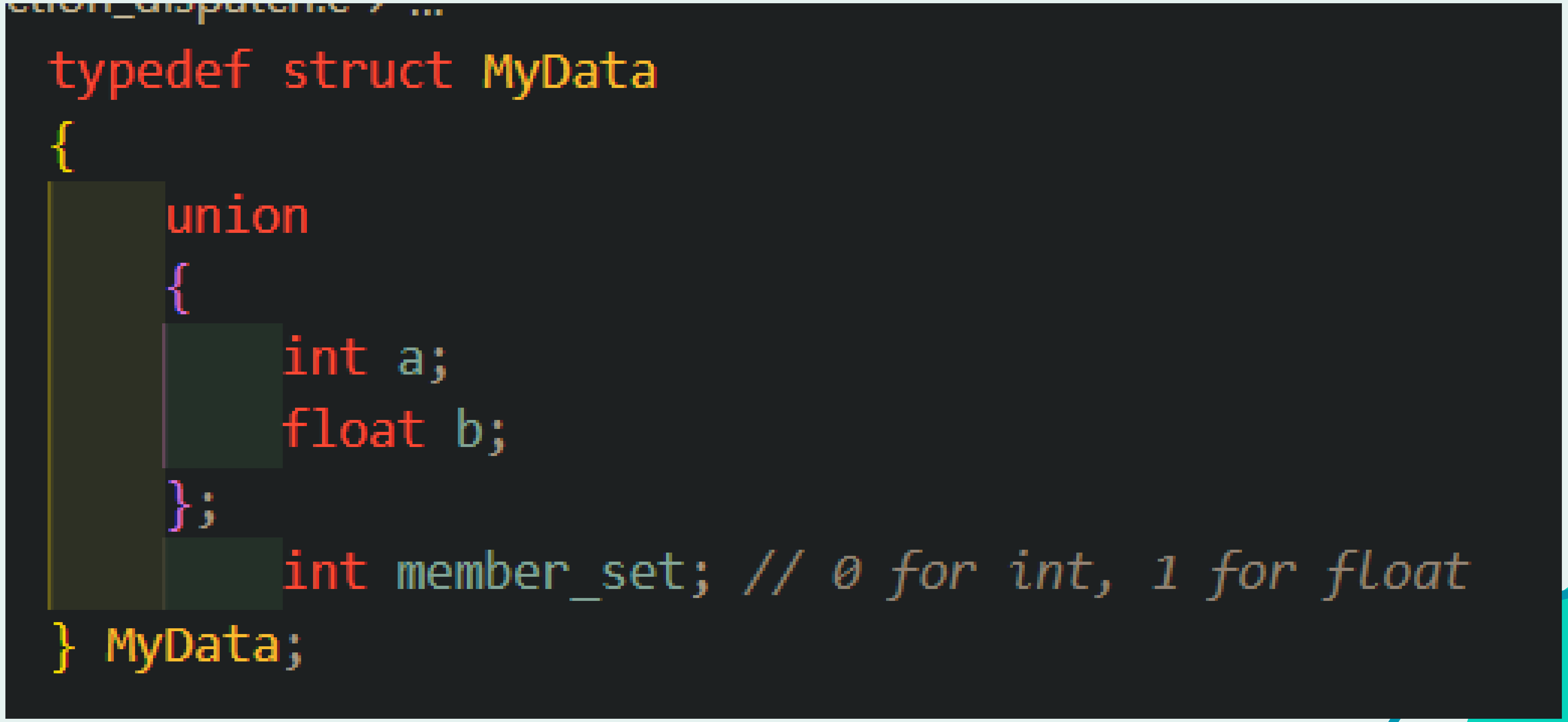
```c
int main() {
    //create a circle
    struct Shape circle;
    float circleData = 5.0;
    circle.type = 1; //circle type
    circle.data = &circleData;
    circle.print = printCircle;

    //create a rectangle
    struct Shape rectangle;
    float rectangleData[] = {3.0, 4.0}; //width and height
    rectangle.type = 2; //rectangle type
    rectangle.data = rectangleData;
    rectangle.print = printRectangle;

    //print information about the shapes
    circle.print(circle.data);
    rectangle.print(rectangle.data);

    return 0;
}
```

Unions

- Unions are similar to structs except that they only have enough memory to hold one member at a time.
- Often, unions provide flexibility to the programmer for different implementations of the same object, while also aiding memory utilization.

Tagged Unions

- Unions but with tags to indicate the type of data they hold.



```
typedef struct MyData
{
    union
    {
        int a;
        float b;
    };
    int member_set; // 0 for int, 1 for float
} MyData;
```

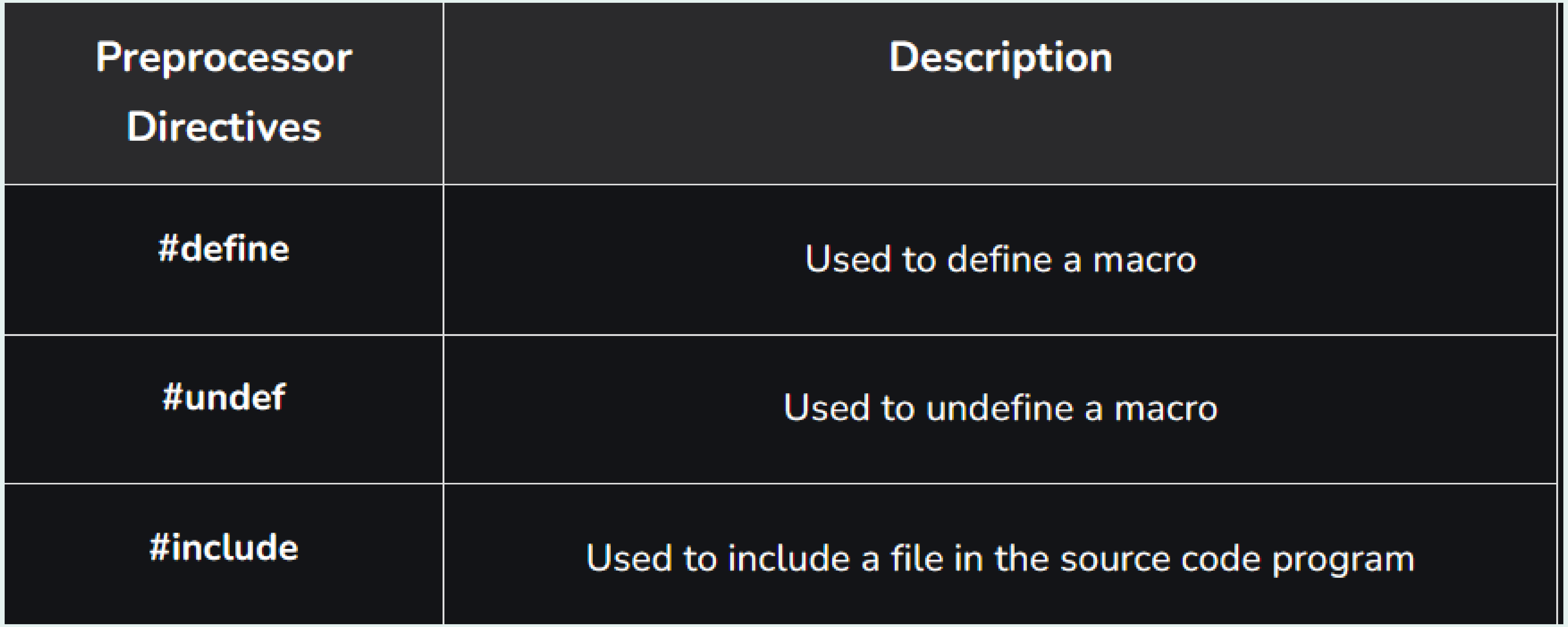
Switch Vs Function Dispatch

Coroutines

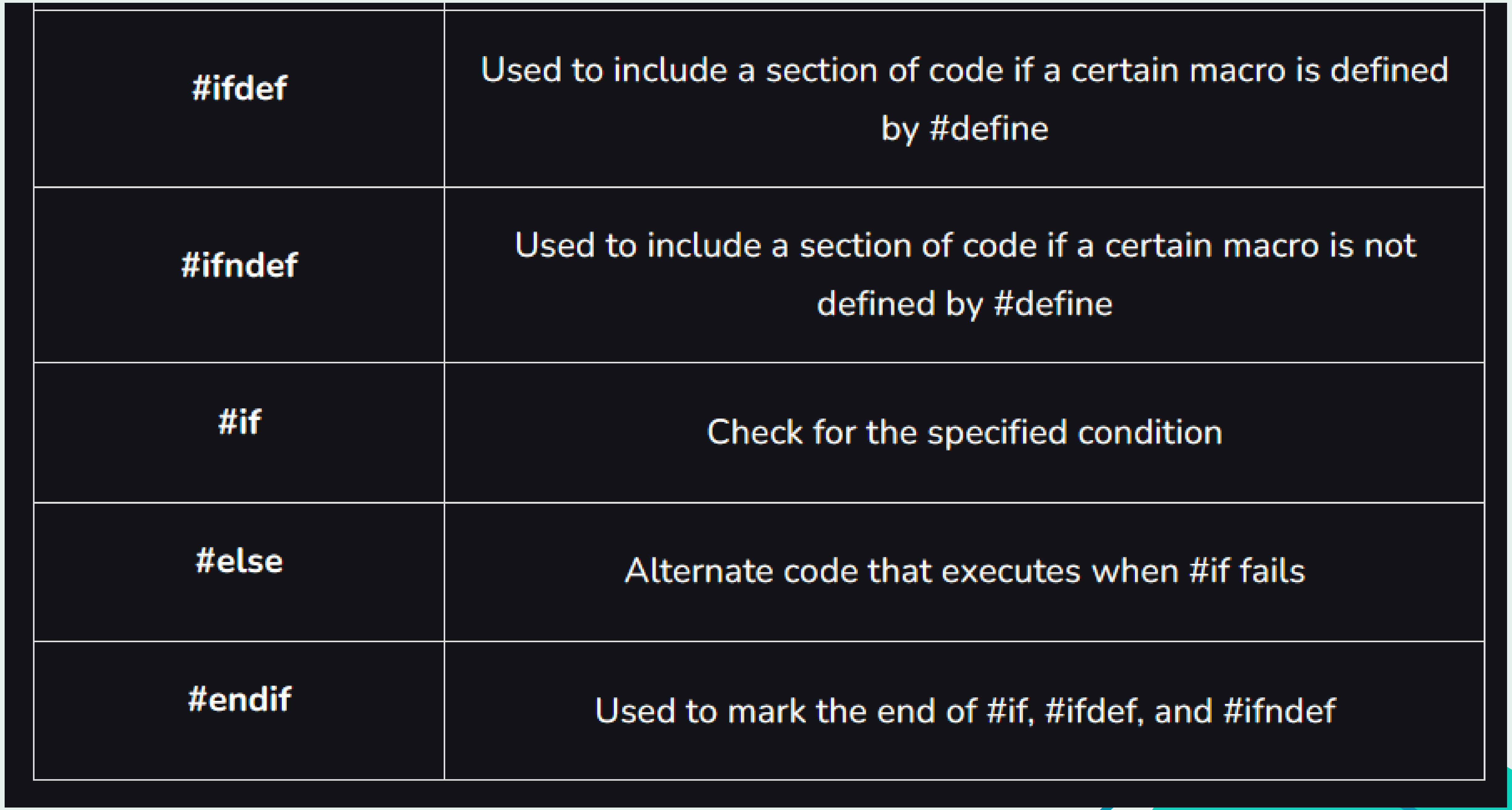
Static Variables

- A static variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.

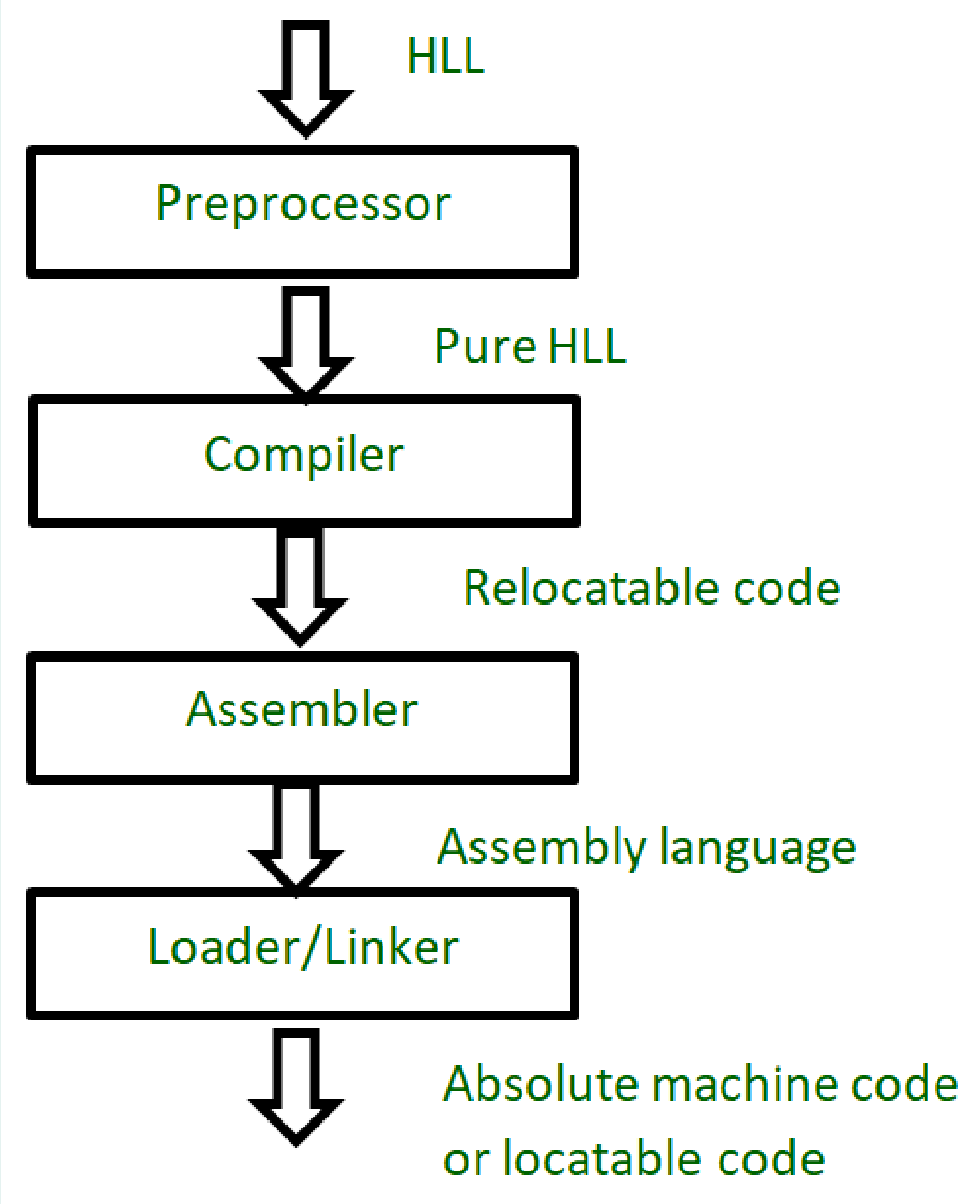
Preprocessor Directives



Preprocessor Directives	Description
#define	Used to define a macro
#undef	Used to undefine a macro
#include	Used to include a file in the source code program

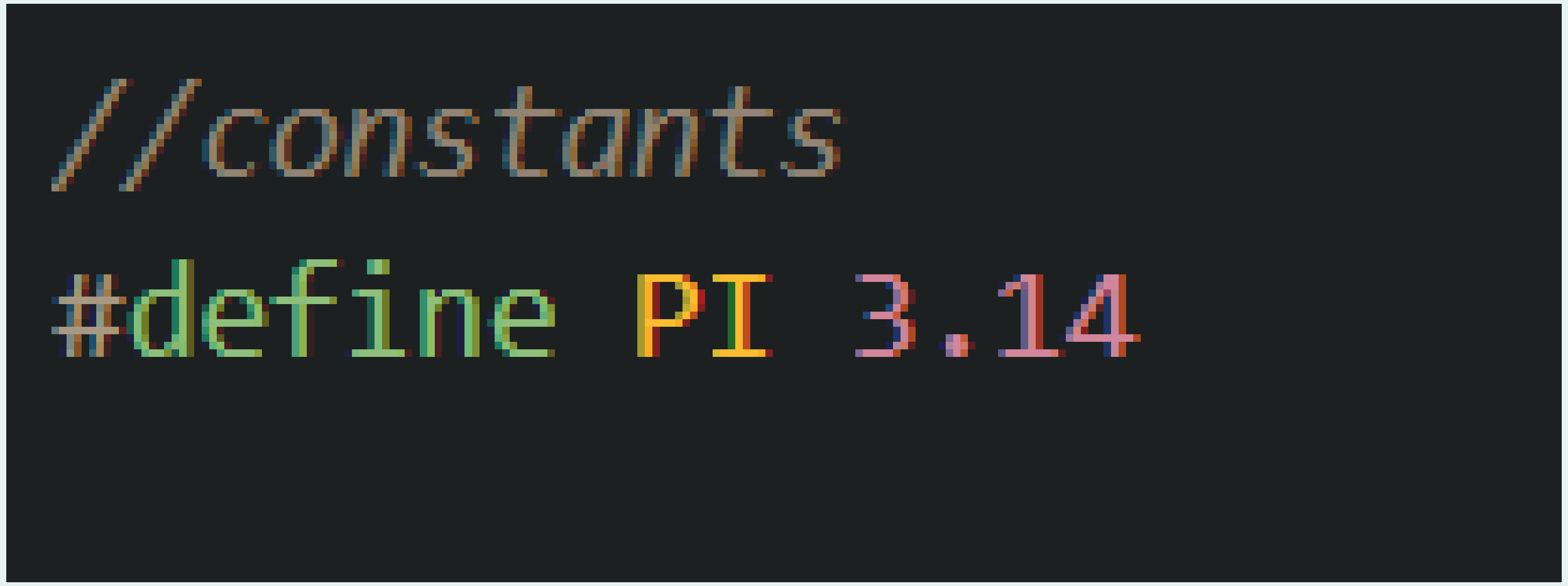


#ifdef	Used to include a section of code if a certain macro is defined by #define
#ifndef	Used to include a section of code if a certain macro is not defined by #define
#if	Check for the specified condition
#else	Alternate code that executes when #if fails
#endif	Used to mark the end of #if, #ifdef, and #ifndef


HLL
Preprocessor
Pure HLL
Compiler
Relocatable code
Assembler
Assembly language
Loader/Linker
Absolute machine code
or locatable code

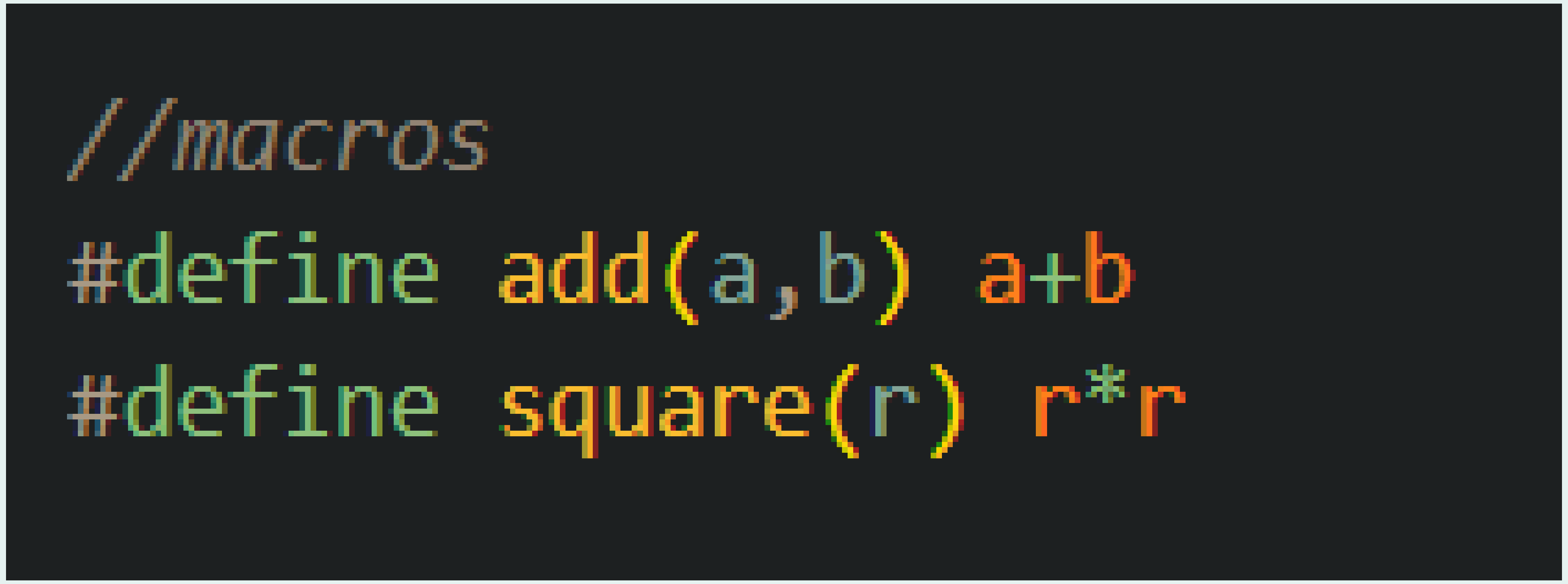
#defines

Constants



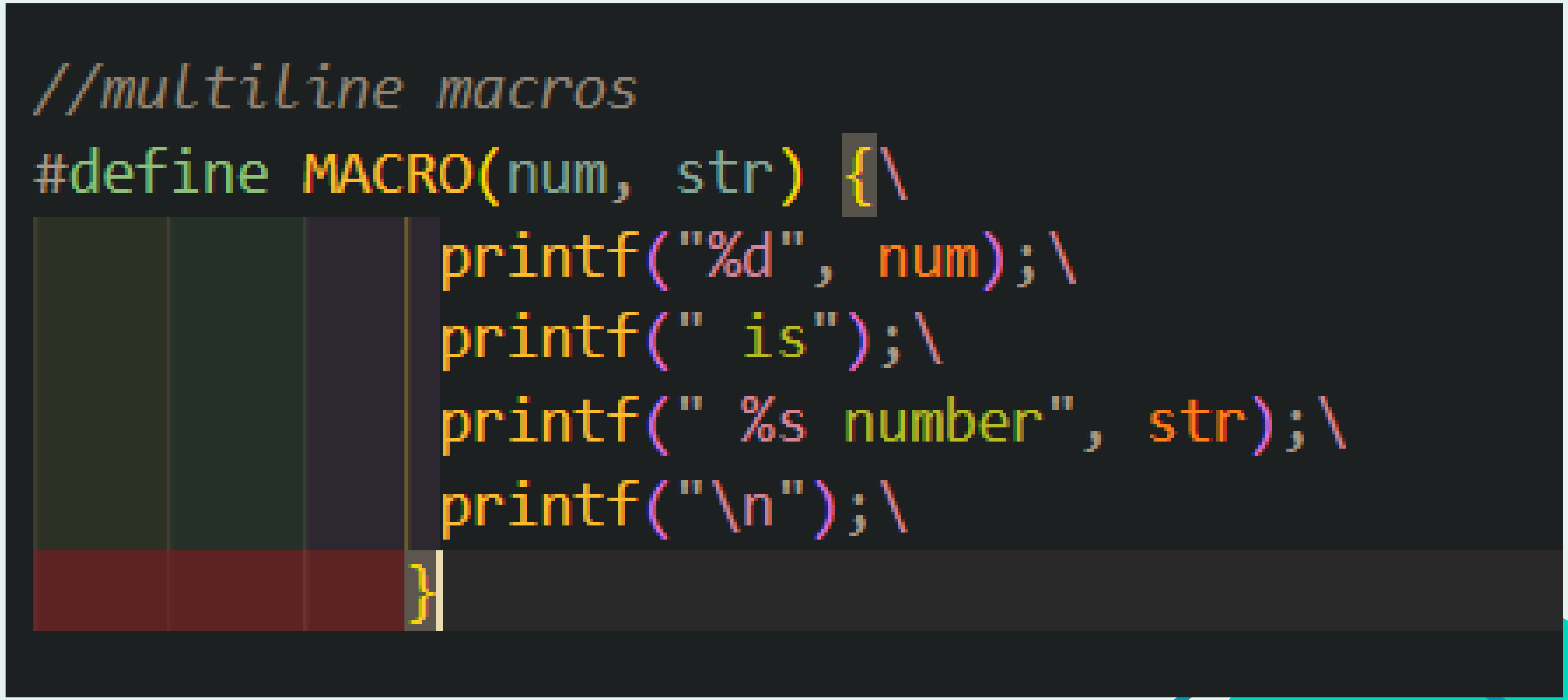
```
//constants
#define PI 3.14
```

Macros



```c
//macros
#define add(a,b) a+b
#define square(r) r*r
```

Multiline Macros



```
//multiline macros
#define MACRO(num, str) {\
            printf("%d", num);\
            printf(" is");\
            printf(" %s number", str);\
            printf("\n");\
           }
```

Thank You!