Global Variables/Constants and the Preprocessor Directive `define`: Array Case Study

Pedram Pasandide

Introduction

The `#define` directive allows defining macros, which are constants or expressions that the preprocessor replaces before compilation. We have already seen a simple example of defining an integer that is accessible in multiple scopes using the format:

```
#define MACRO_NAME value
```

such as:

```
#define MAX_SIZE 100
```

What are macros? Macros in C are preprocessor directives that define constants or expressions, allowing the compiler to replace occurrences of the macro name with its corresponding value before compilation. Here, `MAX_SIZE` is replaced with 100 wherever it appears in the code. Macros help improve code readability and maintainability.

So, macros are not limited to simple integer values. They can also represent expressions or even functions. The general formats are as follows.

We can define a macro that holds an **expression** within brackets:

```
#define MACRO_NAME (expression)
```

such as:

```
#include <stdio.h>

// Pi with 22/7 and 22.0/7 a reminder to type casting
#define PI_in (22 / 7)
```

```
#define PI_double (22.0 / 7)
#define SQUARE(x) ((x) * (x))

int main() {

 double Cradius = 7;
 double Carea;

 Carea = PI_in * SQUARE(Cradius);
 printf("Area of Circle of PI_in %0.2lf: %0.2lf\n", Cradius,
    Carea);
 Carea = PI_double * SQUARE(Cradius);
 printf("Area of Circle of PI_double %0.2lf: %0.2lf\n", Cradius,
    Carea);

 return 0;
}
```

resulting:

```
Area of Circle of PI_in 7.00: 147.00
Area of Circle of PI_double 7.00: 154.00
```

`PI_in` will replaced by `(22/7)` and the arithmetic operation will be done during execution time. Note that both the numerator and denominator are integers so the result of this division will be `3`. However, `PI_double` which is `(22.0/7)`, is equal to `3.1428571428571428e+00`. `SQUARE(x)` is a macro user defined function with one input argumentation `x`, and every time called it will be replaced by `((x)*(x))`.

Macros can also take multiple arguments, similar to functions:

```
#define MACRO_NAME(ARG1, ARG2,..) (expression)
```

such as:

```
#include <stdio.h>

#define MULTIPLY(a, b) ((a) * (b))

int main() {
 int x = 5, y = 10;
```

```
 printf("Product: %d\n", MULTIPLY(x, y));
 return 0;
}
```

Macros can take arguments, but they lack type safety.

```
#define SQUARE(x) x * x

int main() {
 printf("SQUARE(5+1): %d\n", SQUARE(5+1)); // Expected 36, but
    gets 11!
}
```

Outputs:

```
SQUARE(5+1): 11                ^
```

Use parentheses:

```
#define SQUARE(x) ((x) * (x))
```

Macros can control which parts of the code are compiled.

```
#include <stdio.h>

#define DEBUG 1  // Comment this to disable debug mode

int main() {
 #ifdef DEBUG
 printf("Debug mode is ON\n");
 #else
 printf("Debug mode is OFF\n");
 #endif
 return 0;
}
```

The above code outputs:

```
Debug mode is ON               ^
```

If we remove the line:

```
#define DEBUG 1  // Comment this to disable debug mode
```

Then, we'll have:

```
Debug mode is OFF                    ^
```

You can also use `-D` in command line to pass the macro input such as:

```
gcc -o program -DDEBUG=2025 myprogram.c
```

Now, it will return:

```
Debug mode is ON                    ^
```

C provides built-in macros for debugging and logging.

Table 1: Built-in macros

Macro	Purpose
`__FILE__`	Current file name
`__LINE__`	Current line number
`__DATE__`	Compilation date
`__TIME__`	Compilation time
`__func__`	Current function name

```
#include <stdio.h>

#define LOG_ERROR(msg) printf("Error in %s at line %d: %s\n",
   __FILE__, __LINE__, msg)

int main() {
 LOG_ERROR("Something went wrong");
 return 0;
}
```

Results:

```
Error in main1.c at line 6: Something went wrong                    ^
```

A macro can be undefined using `#undef`.

```c
#define TEMP 100
#undef TEMP
#define TEMP 200

int main() {
 printf("TEMP: %d\n", TEMP);  // Prints 200
 return 0;
}
```

C99 introduced variadic macros, which allow macros to accept a variable number of arguments using `...`.

```c
#include <stdio.h>

#define LOG(format, ...) printf("LOG: " format "\n", __VA_ARGS__)

int main() {
 LOG("Value: %d", 42);
 LOG("Sum: %d + %d = %d", 2, 3, 2+3);
 return 0;
}
```

Outputs:

```
LOG: Value: 42
LOG: Sum: 2 + 3 = 5
```

While macros can replace function calls, inline functions (introduced in C99) are often better because they provide type safety.

```c
#include <stdio.h>

inline int square(int x) { return x * x; }

int main() {
 printf("Square of 6: %d\n", square(6));
 return 0;
}
```

The Difference Between `define` and Global Variables/Constants

Mind you the following code:

```
#include <stdio.h>

#define PreProcessVariable 12

int main() {
 PreProcessVariable = 13;
 printf("result = %d\n", PreProcessVariable);
}
```

results in a compilation error:

```
pedram.c: In function ‘main’:
pedram.c:7:22: error: lvalue required as left operand of assignment
7 |   PreProcessVariable = 13;
  |                      ^
```

Why Does This Happen? The `#define` directive performs text substitution before the code is compiled. When the preprocessor encounters `PreProcessVariable`, it replaces it with `12`, so the compiler effectively sees this:

```
12=13
```

Since `12` is a literal constant and not a variable, it cannot be assigned a new value, leading to the compilation error. This behavior is similar to constant global variables that we have seen before, such as:

```
const int PreProcessVariable = 12;
PreProcessVariable = 13;  // Error: Assignment to a read-only
   variable
```

In both cases, the value is fixed and cannot be modified. However, unlike `const`, which has type checking, a macro is purely a textual replacement.

Case Study: Array Initialization

In C, arrays can be allocated without using dynamic memory allocation. This means that their size must be known at compile time, as the compiler needs to reserve a fixed amount of memory

for them in the program's stack. The most straightforward way to declare an array is to use a fixed integer value or a `#define` directive. However, when using variables or `const` values, unexpected compilation errors can arise.

Take the following code as an example:

```
#include <stdio.h>

int sizeGV = 2;
const int sizeGC = 2;
#define sizeDf 2

int main() {
 int sizeV = 2;
 const int sizeC = 2;

 double array1D_v0[]   = {1,2}; // This works
 double array1D_v1[2]  = {1,2}; // This works

 double array1D_vV[sizeV]  = {1,2}; // 1. Compilation error
 double array1D_vC[sizeC]  = {1,2}; // 2. Sometimes Compilation error

 double array1D_vGV[sizeGV]  = {1,2}; // 3. Compilation error
 double array1D_vGC[sizeGC]  = {1,2}; // 4. Sometimes Compilation error

 double array1D_vDf[sizeDf]  = {1,2}; // This works
}
```

The above code produces the following errors:

```
In function 'main':
variable-sized object may not be initialized except with an empty initializer
14 |     double array1D_vV[sizeV]  = {1,2}; // 1. Compilation error
|                                 ^
error: variable-sized object may not be initialized except with an empty initializer
15 |     double array1D_vC[sizeC]  = {1,2}; // 2. Sometimes Compilation error
|                                 ^
error: variable-sized object may not be initialized except with an empty initializer
17 |     double array1D_vGV[sizeGV]  = {1,2}; // 3. Compilation error
|                                   ^
error: variable-sized object may not be initialized except with an empty initializer
18 |     double array1D_vGC[sizeGC]  = {1,2}; // 4. Sometimes Compilation error
```

This means Variable-Length Arrays (**VLAs**) cannot be initialized. Here, `sizeV` and `sizeC` are

local variables. Even though `sizeC` is declared as `const`, in C (prior to C23), `const` does **not** guarantee **compile-time** evaluation. Instead, `sizeC` is treated as a **runtime variable**, making `array1D_vC` a Variable-Length Array.

In C99 and later, VLAs cannot be initialized with values. Their size is determined at runtime, and they must be assigned values separately after declaration. This explains why the compiler gives an error stating that "variable-sized object may not be initialized except with an empty initializer."

`sizeGV` is a global variable, meaning its value is not known at compile time. The compiler treats `sizeGV` as a runtime-determined value, making `array1D_vGV` a VLA, which again cannot be initialized.

`sizeGC` is a global `const` variable. However, whether it is treated as a compile-time constant depends on the compiler. Some compilers might replace `sizeGC` with `2` during compilation, while others might still treat it as a runtime variable.

Thus, while `array1D_vGC` may compile on some systems, it does not have guaranteed behavior across all C compilers. Visual Studio Code (VSCode) might not show an error because the compiler it uses (MSVC or Clang) could be optimizing `sizeGC` as a constant, while GCC on Linux may not.

Macros (`#define`) are purely text replacements done by the preprocessor before the compiler even starts analyzing the code. When the compiler processes this line, **it sees**:

```
double array1D_vDf[2] = {1,2};
```

Since `2` is a **compile-time constant**, this works perfectly fine. Unlike `const` variables, **macros are always resolved at compile time**.

The same concepts we discussed for one-dimensional arrays apply to multidimensional arrays as well. Take the following example

```
#include <stdio.h>

#define sizeDf 2

int main() {
 double array2Dv0[][sizeDf] = {{1,2}, {2,3}};
 double array2Dv1[sizeDf][] = {{1,2}, {2,3}}; // Compilation error
 double array2Dv2[sizeDf][sizeDf]= {{1,2}, {2,3}};
}
```

This code produces the following error:

```
In function ‘main’:
error: array type has incomplete element type ‘double[]’
7 |     double array2Dv1[sizeDf][]       = {{1,2}, {2,3}};  // Compilation error
|            ^~~~~~~~~
note: declaration of ‘array2Dv1’ as multidimensional array must have bounds for all
dimensions except the first
```

The line with `array2Dv1[sizeDf][] = 1,2, 2,3` causes an error because in C, when declaring a multidimensional array, the size of all but the first dimension must be explicitly specified. `array2Dv0` works because the first dimension is left empty, and the compiler **infers** its size from the number of initializer lists provided (`{{1,2}, {2,3}}` contains two rows). `array2Dv2` works because both dimensions are explicitly specified (`[sizeDf][sizeDf]`). `array2Dv1` **fails** because the second dimension is unspecified, and **C does not allow implicit size deduction for any dimension except the first**.

This restriction exists because **without knowing the size of inner arrays, the compiler cannot correctly calculate memory offsets for elements** in a multidimensional array. This example serves as a reminder of the fundamental array rules we covered in the previous section. Just like one-dimensional arrays:

1. Macros (`#define`) work fine because they are replaced at compile time.
2. Multidimensional arrays must have fixed sizes, except for the first dimension.
3. The compiler must know all array sizes except the first to correctly allocate memory.

What should we do if we don’t know the array size at compile time? For example, if we want to create a multidimensional array based on user input? This is where **dynamic memory allocation** comes into play, which allows us to allocate memory during runtime instead of compile time. Dynamic memory allocation will be covered in the next chapter.