The Preprocessor (2)

Program Design (II)

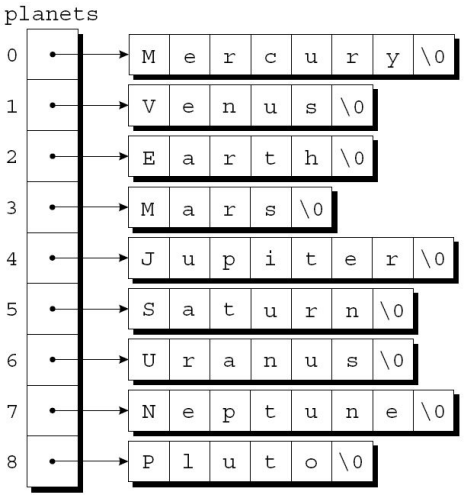
2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Question from last class: `sizeof()`

- How to compute the size of planets?
- What is the `sizeof(planets)/sizeof(planets[0])` actually compute?

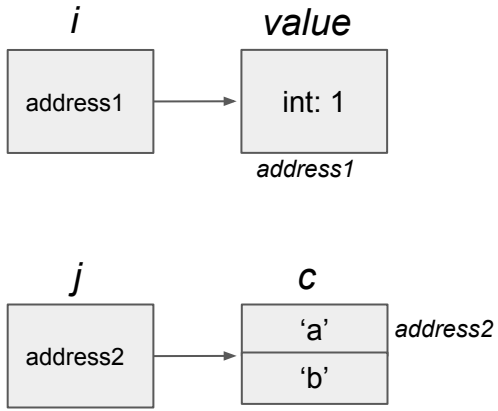
```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```



Question from last class: `sizeof()`

- First, we need to know when compute the size of the following condition
  - a pointer: C use 8 bytes to store memory address
  - a pointer point to array
  - an array of pointer

```
int value = 1;
char c[2] = {'a', 'b'};
int *i = &value; //a pointer point to int obj
char *j = c; // a pointer point to char obj
```

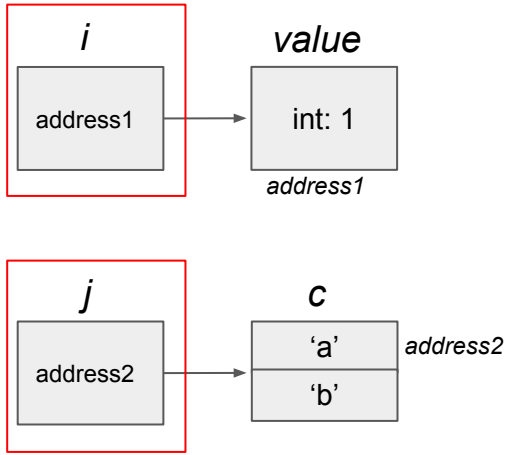


Question from last class: `sizeof()`

- First, we need to know when compute the size of the following condition
  - a pointer: C use 8 bytes to store memory address
  - a pointer point to array
  - an array of pointer

```c
int value = 1;
char c[2] = {'a', 'b'};
int *i = &value; //a pointer point to int obj
char *j = c; // a pointer point to char obj

printf("A pointer has size of %zu bytes\n", sizeof(i));
printf("A pointer has size of %zu bytes\n", sizeof(j));
```

```
A pointer has size of 8 bytes
A pointer has size of 8 bytes
```



slido

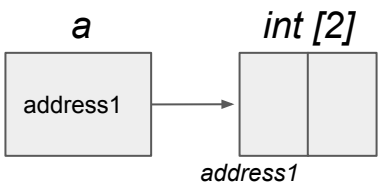
What is the correct printed message?

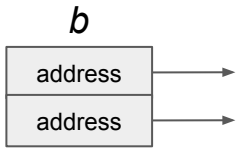
ⓘ Start presenting to display the poll results on this slide.

Question from last class: `sizeof()`

- First, we need to know when compute the size of the following condition
  - a pointer: C use 8 bytes to store memory address
  - a pointer point to array
  - an array of pointer

```
int (*a)[2]; //a pointer point to int array with 2 elements
int *b[2]; // an array of pointers; two pointers in the array b
```

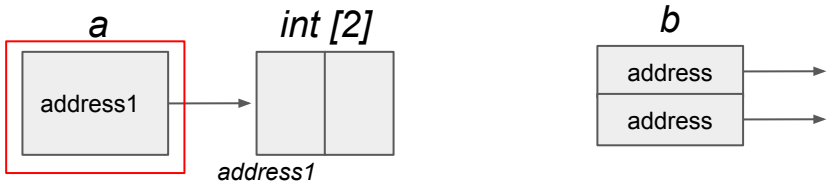




Question from last class: `sizeof()`

- First, we need to know when compute the size of the following condition
  - a pointer: C use 8 bytes to store memory address
  - a pointer point to array
  - an array of pointer

```c
int (*a)[2]; //a pointer point to int array with 2 elements
int *b[2]; // an array of pointers; two pointers in the array b

printf("A pointer has size of %zu bytes\n", sizeof(a));
printf("Two pointers stored in array of pointer b have size of %zu bytes\n", sizeof(b));
```

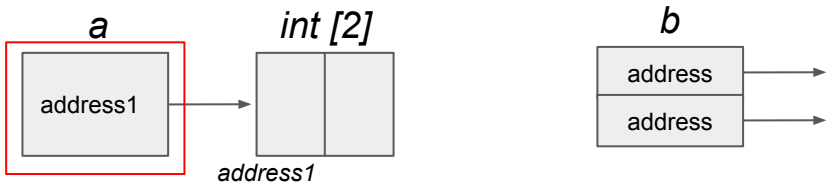


Question from last class: `sizeof()`

- First, we need to know when compute the size of the following condition
  - a pointer: C use 8 bytes to store memory address
  - a pointer point to array
  - an array of pointer

```c
int (*a)[2]; //a pointer point to int array with 2 elements
int *b[2]; // an array of pointers; two pointers in the array b

printf("A pointer has size of %zu bytes\n", sizeof(a));
printf("Two pointers stored in array of pointer b have size of %zu bytes\n", sizeof(b));
```

Where does the `sizeof(b)` compute?

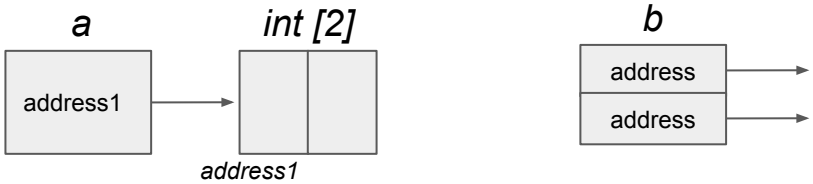


Question from last class: `sizeof()`

- First, we need to know when compute the size of the following condition
  - a pointer: C use 8 bytes to store memory address
  - a pointer point to array
  - an array of pointer

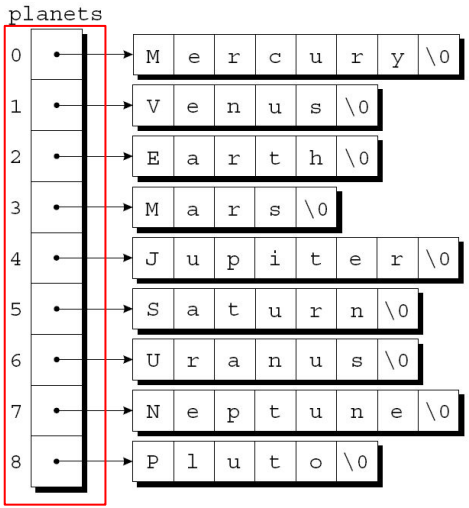
```c
int (*a)[2]; //a pointer point to int array with 2 elements
int *b[2]; // an array of pointers; two pointers in the array b

printf("A pointer has size of %zu bytes\n", sizeof(a));
printf("Two pointers stored in array of pointer b have size of %zu bytes\n", sizeof(b));
```

```
A pointer has size of 8 bytes
Two pointers stored in array of pointer b have size of 16 bytes
```



Question from last class: `sizeof()`

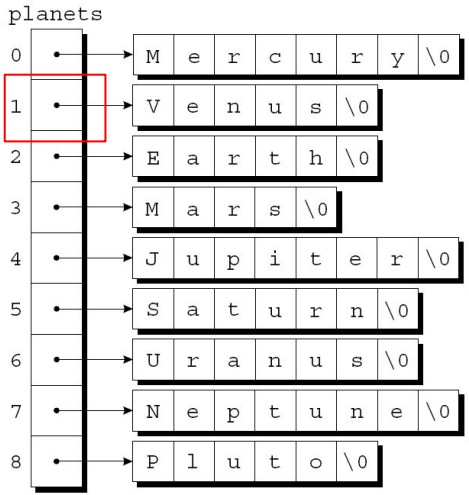
```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};

printf("The array of pointer planets has %zu bytes\n", sizeof(planets)); // 9*8 = 72
```




```
The array of pointer planets has 72 bytes
```

Question from last class: `sizeof()`

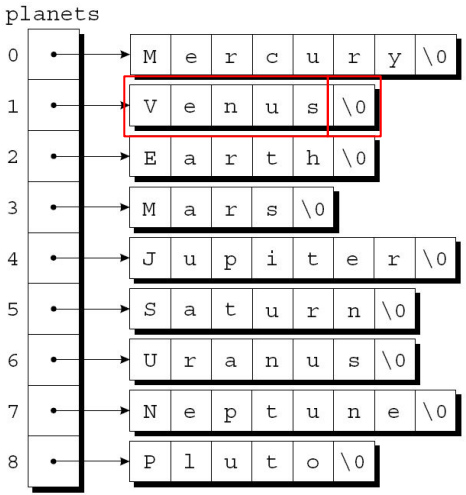
```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};

printf("The array of pointer planets has %zu bytes\n", sizeof(planets)); // 9*8 = 72
printf("One pointer has %zu bytes\n", sizeof(planets[1]));
```




```
The array of pointer planets has 72 bytes
One pointer has 8 bytes
```

Question from last class: `sizeof()`

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};

printf("The array of pointer planets has %zu bytes\n", sizeof(planets)); // 9*8 = 72
printf("One pointer has %zu bytes\n", sizeof(planets[1]));
printf("The second string %s has %zu bytes\n", (planets[1]), strlen(planets[1]) + 1);
```




```
The array of pointer planets has 72 bytes
One pointer has 8 bytes
The second string Venus has 6 bytes
```

extra reading: why can't we use `sizeof()` to find the size of string pointed by `planets[1]`?
https://stackoverflow.com/questions/492384/how-to-find-the-sizeof-a-pointer-pointing-to-an-array

Outline

- Macro Definitions
- Predefined Macros
- Conditional Compilation

Macro Definitions

- The macros that we've been using a lots are known as *simple* macros, because they have no parameters.
- The preprocessor also supports *parameterized* macros.

```
#define PI 3.14 //simple macros
#define IS_EVEN(n) ((n)%2==0) //parameterized macros
```

Simple Macros

- Definition of a ***simple macro*** (or ***object-like macro***)
- The replacement list may include identifiers, keywords, numeric constants, character constants, string literals, operators, and punctuation.
- Wherever *identifier* appears later in the file, the preprocessor replace *identifier* by *replacement-list*.

```
#define identifier replacement-list
```

Simple Macros

- Example of using operator + as the *replacement-list*.

```
#include <stdio.h>
#define ADD +

int main(void){
  printf("%d", 1 ADD 1);
  return 0;
}
```

```
content from stdio.h


int main(void){
  printf("%d", 1 + 1);
  return 0;
}
```

Simple Macros

- Any extra symbols in a macro definition will become part of the replacement list.
- Putting the = symbol in a macro definition is a common error:
- Ending a macro definition with a semicolon is another popular mistake:

```
#define N = 100  /*** WRONG ***/
…
int a[N]; /* int a[= 100]; */
```

```
#define N 100;  /*** WRONG ***/
…
int a[N]; /* int a[100;]; */
```

Simple Macros

- Simple macros are primarily used for defining “manifest constants”—names that represent numeric, character, and string values:

```
#define STR_LEN 80
#define TRUE    1
#define FALSE   0
#define PI      3.14159
#define EOS     '\0'
#define MEM_ERR "Error: not enough memory"
```

Simple Macros

- Advantages of using `#define` to create names for constants:
- ***It makes programs easier to read.*** The name of the macro can help the reader understand the meaning of the constant.

```
int main(){
    int n = 100;
    …
    return 0;
}
```

```
#define NUM_STUDENT 100

int main(){
    int n = NUM_STUDENT;
    …
    return 0;
}
```

Simple Macros

- Advantages of using `#define` to create names for constants:
- ***It makes programs easier to modify.*** We can change the value of a constant throughout a program by modifying a single macro definition.

```
int main(){
    int n = 100;
    …
    int j = 100/2;
    return 0;
}
```

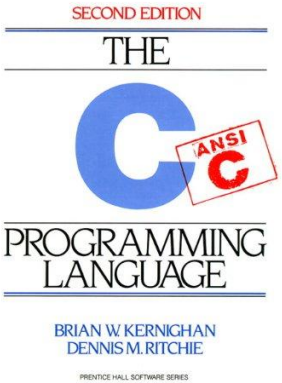
```
#define NUM_STUDENT 100

int main(){
    int n = NUM_STUDENT;
    …
    int j = NUM_STUDENT/2;
    return 0;
}
```

Simple Macros

- Advantages of using `#define` to create names for constants:
- ***It helps avoid inconsistencies and typographical errors.*** If a numerical constant like 3.14159 appears many times in a program, chances are it will occasionally be written 3.1416 or 3.14195 by accident.

Simple Macros

- When macros are used as constants, C programmers customarily capitalize all letters in their names.
- However, there's no consensus as to how to capitalize macros used for other purposes.
  - Some programmers like to draw attention to macros by using all upper-case letters in their names.
  - Others prefer lower-case names, following the style of K&R.



Parameterized Macros

- Definition of a ***parameterized macro*** (also known as a ***function-like macro***):
- $x_1, x_2, \ldots, x_n$ are identifiers (the macro's ***parameters***).
- The parameters may appear as many times as desired in the *replacement list*.

```
#define identifier(x1 , x2 , ... , xn ) replacement-list
#define IS_EVEN(n) ((n)%2==0)
#define POWER(i) ((i)*(i))
```

Parameterized Macros

- There must be *no space* between the macro name and the left parenthesis.
- If space is left, the preprocessor will treat $(x_1, x_2, \ldots, x_n)$ as part of the replacement list.

```
#define identifier (x1 , x2 , ... , xn ) replacement-list
#define IS_EVEN (n) ((n)%2==0)
#define POWER (i) ((i)*(i))
```

Parameterized Macros

- When the preprocessor encounters the definition of a parameterized macro, it stores the definition away for later use.
- Wherever a macro ***invocation*** of the form *identifier* $(y_1, y_2, \ldots, y_n)$ appears later in the program, the preprocessor replaces it with *replacement-list*, substituting y_1 for x_1, y_2 for x_2, ...

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define IS_EVEN(n)  ((n)%2==0)

i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

Parameterized Macros

- A more complicated function-like macro:
- A parameterized macro may have an empty parameter list
  - The empty parameter list isn't really needed, but it makes `PRINT_MSG()` resemble a function.

```
#define TOUPPER(c) \
     ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
#define PRINT_MSG() printf("Hello World")
```

Parameterized Macros

- Using a parameterized macro instead of a true function has a couple of advantages:
  - ***The program may be slightly faster.*** A function call usually requires some overhead during program execution, but a macro does not.
  - ***Macros are "generic."*** A macro can accept arguments of any type, provided that the resulting program is valid.

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
…
MAX(1, 2); //find the larger of two int values
MAX(1.1, 1.2); //find the larger of two float values
```

Parameterized Macros

- Parameterized macros also have disadvantages.
- ***The compiled code will often be larger.*** Each macro invocation increases the size of the source program.

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
int n = MAX(i, MAX(j, k));
// after preprocessing:
//  n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```

Parameterized Macros

- Parameterized macros also have disadvantages.
- ***Arguments aren't type-checked.*** When a function is called, the compiler checks each argument to see if it has the appropriate type. Macro arguments aren't checked by the preprocessor, nor are they converted.

```
#define PRINT_INT(x) printf("%d\n", x)
PRINT_INT(0.1);
```

Let’s take a break

General Properties of Macros

- Several rules apply to both simple and parameterized macros.
- ***A macro's replacement list may contain invocations of other macros.***
- When it encounters `TWO_PI` later in the program, the preprocessor replaces it by `(2*PI)`.
- The preprocessor ***rescans*** the replacement list to see if it contains invocations of **other** macros.

```
#define PI     3.14159
#define TWO_PI (2*PI)
```

General Properties of Macros

- ***A macro definition normally remains in effect until the end of the file in which it appears.***
- Macros don't obey normal **scope** rules.
- A macro defined inside the body of a function isn't local to that function; it remains defined until the end of the file.

```
void f(){
    #define PI      3.14159
}

int main(){
    float i = PI / 2.0f;
    return 0;
}
```

General Properties of Macros

- ***A macro may not be defined twice unless the new definition is identical to the old one.*** Differences in spacing are allowed, but the tokens in the macro's replacement list must be the same.

```
#define PI 3.14159
#define PI 3       //WRONG!
#define PI 3.14159
```

General Properties of Macros

- ***Macros may be "undefined" by the `#undef` directive.***
- One use of `#undef` is to remove the existing definition of a macro so that it can be given a new definition.

```
#define PI 3.14159
#undef PI
#define PI 3        //CORRECT!
```

Parentheses in Macro Definitions

- The replacement lists in macro definitions often **require** parentheses in order to avoid unexpected results.
- If the macro's replacement list contains an **operator**, always enclose the replacement list in parentheses
- Also, put parentheses around each **parameter** every time it appears in the replacement list

```
#define TWO_PI (2*3.14159)
#define SCALE(x) ((x)*10)
```

Parentheses in Macro Definitions

- Without the parentheses, we can't guarantee that the compiler will treat replacement lists and arguments as whole expressions.
- For example, …
  - The division will be performed before the multiplication.

```
// instead of #define TWO_PI (2*3.14159)
#define TWO_PI 2*3.14159

int conversion_factor = 360/TWO_PI;
// will become: conversion_factor = 360/2*3.14159;
```

Predefined Macros of C

Macro	Value
`__DATE__`	A string containing the current date
`__FILE__`	A string containing the file name
`__LINE__`	An integer representing the current line number
`__STDC__`	If follows ANSI standard C, then the value is a nonzero integer
`__TIME__`	A string containing the current date.

https://www.programiz.com/c-programming/c-preprocessor-macros

Conditional Compilation

- The C preprocessor recognizes a number of directives that support ***conditional compilation.***
- This feature permits the inclusion or exclusion of a section of program text depending on the outcome of a test performed by the preprocessor.
- Namely, we can use directives to control if this section of program will be included or removed after preprocessing

The `#if` and `#endif` Directives

- Suppose we're in the process of debugging a program.
- We'd like the program to print the values of certain variables, so we put calls of `printf` in critical parts of the program.
- Once we've located the bugs, it's often a good idea to let the `printf` calls remain, just in case we need them later.
- How to use conditional compilation to achieve this?

The `#if` and `#endif` Directives

- The first step is to define a macro and give it a nonzero value
- Next, we'll surround each group of `printf` calls by an `#if-#endif` pair

```
#define DEBUG 1
…
int main(){
…
    #if DEBUG
        printf("i: %d\n", i);
        printf("j: %d\n", j);
    #endif
…
}
```

The `#if` and `#endif` Directives

- During preprocessing, the `#if` directive will test the value of `DEBUG`.
- Since its value isn't zero, the preprocessor will leave the two calls of `printf` in the program.

```
#define DEBUG 1
…
int main(){
…
    #if DEBUG
        printf("i: %d\n", i);
        printf("j: %d\n", j);
    #endif
…
}
```

The `#if` and `#endif` Directives

- If we change the value of `DEBUG` to zero and recompile the program, the preprocessor will remove all four lines from the program.

```
#define DEBUG 0
…
int main(){
…
    #if DEBUG
        printf("i: %d\n", i);
        printf("j: %d\n", j);
    #endif
…
}
```

The `#if` and `#endif` Directives

- General form of the `#if` and `#endif` directives:
- When the preprocessor encounters the `#if` directive, it evaluates the constant expression.
- If the value of the expression is **zero**, the lines between `#if` and `#endif` will be removed from the program during preprocessing.
- Otherwise (not zero), the lines between `#if` and `#endif` will remain.

```
#if constant-expression
lines
#endif
```

The `#if` and `#endif` Directives

- The `#if` directive treats undefined identifiers as macros that have the value 0.
- If we neglect to define `DEBUG`, *lines1* will be excluded from the program
- but *lines2* will be included in the program

```
#if DEBUG
lines1
#endif

#if !DEBUG
lines2
#endif
```

The `#elif` and `#else` Directives

- Just like `if` statement, `#elif` and `#else` can be used in conjunction with `#if` to test a series of conditions
- Any number of `#elif` directives—but at most one `#else`—may appear between `#if` and `#endif`.

```
#if expr1
Lines to be included if expr1 is nonzero
#elif expr2
Lines to be included if expr1 is zero but expr2 is nonzero
#else
Lines to be included otherwise
#endif
```

Uses of Conditional Compilation

- Conditional compilation has other uses besides debugging.
- ***Writing programs that are portable to several machines or operating systems.***

```
#if WIN32
…
#elif MAC_OS
…
#elif LINUX
…
#endif
```

Uses of Conditional Compilation

- ***Providing a default definition for a macro.***
- Conditional compilation makes it possible to check whether a macro is currently defined and, if not, give it a default definition

```
#if BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

Summary

- Macro Definitions
  - Simple Macros
  - Parameterized Macros
  - General Properties of Macros
  - Parentheses in Macro Definitions
- Predefined Macros
- Conditional Compilation
  - The `#if` and `#endif` Directives
  - The `#elif` and `#else` Directives