CSE105

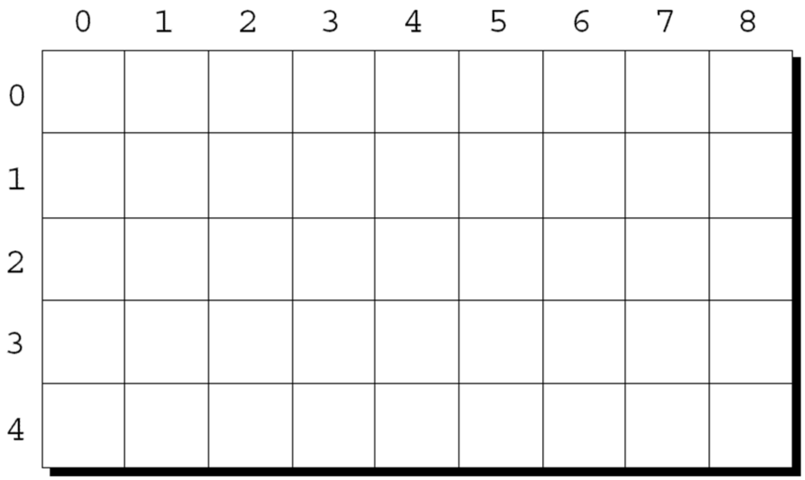
ARRAY (PART 2)

Multidimensional Arrays

- An array may have any number of dimensions.
- The following declaration creates a two-dimensional array (a *matrix*, in mathematical terminology):

```
int m[5][9];
```

- `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0:

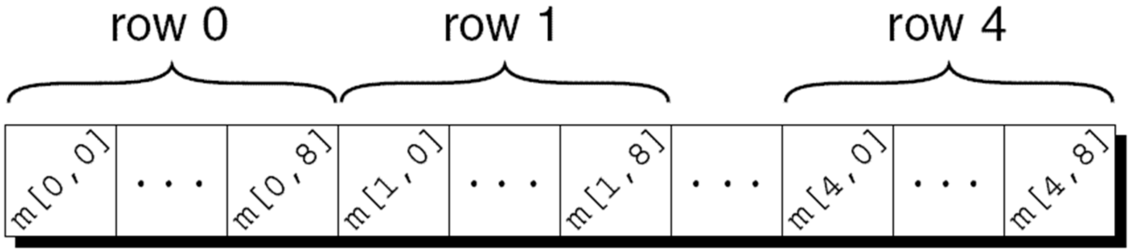


Multidimensional Arrays

- To access the element of `m` in row `i`, column `j`, we must write `m[i][j]`.
- The expression `m[i]` designates row `i` of `m`, and `m[i][j]` then selects element `j` in this row.

Multidimensional Arrays

- Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory.
- C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth.
- How the `m` array is stored:



Multidimensional Arrays

- Nested `for` loops are ideal for processing multidimensional arrays.
- Consider the problem of initializing an array for use as an identity matrix. A pair of nested `for` loops is perfect:

```
double ident[3][3];
int row, col, N=3;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

Initializing a Multidimensional Array

- We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- Initializers for higher-dimensional arrays are constructed in a similar fashion.
- C provides a variety of ways to abbreviate initializers for multidimensional arrays

Initializing a Multidimensional Array

- If an initializer isn’t large enough to fill a multidimensional array, the remaining elements are given the value 0.
- The following initializer fills only the first three rows of `m`; the last two rows will contain zeros:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

Initializing a Multidimensional Array

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1},
               {0, 1, 0, 1, 1, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

Initializing a Multidimensional Array

- We can even omit the inner braces:

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

  Once the compiler has seen enough values to fill one row, it begins filling the next.

Try this

- Create a program that contains the marks of 5 courses of 7 students.
- Find the highest marks obtained by each student.
- Find the class highest marks of each course.
- Take two 5x5 matrices A and B as input and perform C = A+B (using matrix addition rule)
- Take two 5x5 matrices A and B as input and perform C = A X B (using matrix multiplication rule)

FUNCTION (PART 1)

Introduction

- A function is a series of statements that have been grouped together and given a name.
- Each function is essentially a small program, with its own declarations and statements.
- Advantages of functions:
  - A program can be divided into small pieces that are easier to understand and modify.
  - We can avoid duplicating code that's used more than once.
  - A function that was originally part of one program can be reused in other programs.

Defining and Calling Functions

- Before we go over the formal rules for defining a function, let's look at three simple programs that define functions.

Program: Computing Averages

- A function named `average` that computes the average of two `double` values:

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

- The word **`double`** at the beginning is the ***return type*** of `average`.
- The identifiers `a` and `b` (the function's ***parameters***) represent the numbers that will be supplied when `average` is called.

Program: Computing Averages

- Every function has an executable part, called the ***body***, which is enclosed in braces.
- The body of `average` consists of a single `return` statement.
- Executing this statement causes the function to "return" to the place from which it was called; the value of `(a + b) / 2` will be the value returned by the function.

Program: Computing Averages

- A function call consists of a function name followed by a list of ***arguments*.**
  - `average(x, y)` is a call of the `average` function.
- Arguments are used to supply information to a function.
  - The call `average(x, y)` causes the values of `x` and `y` to be copied into the parameters `a` and `b`.
- An argument doesn't have to be a variable; any expression of a compatible type will do.
  - `average(5.1, 8.9)` and `average(x/2, y/3)` are legal.

Program: Computing Averages

- We'll put the call of `average` in the place where we need to use the return value.
- A statement that prints the average of `x` and `y`:

  ```
  printf("Average: %g\n", average(x, y));
  ```

  The return value of `average` isn't saved; the program prints it and then discards it.
- If we had needed the return value later in the program, we could have captured it in a variable:

  ```
  avg = average(x, y);
  ```

Program: Computing Averages

- The `average.c` program reads three numbers and uses the `average` function to compute their averages, one pair at a time:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

average.c

```c
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %lf and %lf:%lf\n", x, y, average(x, y));
  printf("Average of %lf and %lf:%lf\n", y, z, average(y, z));
  printf("Average of %lf and %lf:%lf\n", x, z, average(x, z));

  return 0;
}
```

Program: Printing a Number

- To indicate that a function has no return value, we specify that its return type is `void`:

  ```
  void print_value(int n)
  {
    printf("The number is %d\n", n);
  }
  ```

- `void` is a type with no values.
- A call of `print_value` must appear in a statement by itself:

  ```
  print_value(i);
  ```

- The `myprog.c` program calls `print_value` 10 times inside a loop.

myprog.c

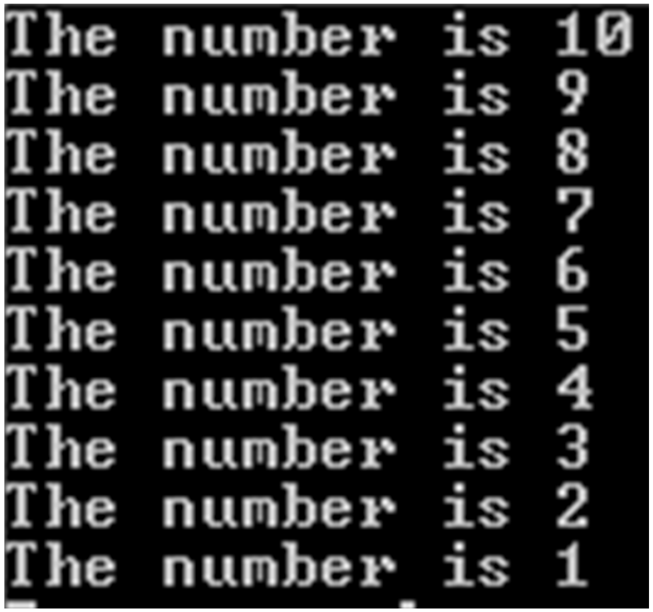
```c
#include <stdio.h>

void print_value(int n)
{
  printf("The number is %d\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_value(i);

  return 0;
}
```



```
The number is 10
The number is 9
The number is 8
The number is 7
The number is 6
The number is 5
The number is 4
The number is 3
The number is 2
The number is 1
```

Program: Printing a text

- When a function has no parameters, the word `void` is placed in parentheses after the function’s name:

```
void print_text(void)
{
  printf(“Hello World\n");
}
```

- To call a function with no arguments, we write the function’s name, followed by parentheses:

```
print_text();
```

  The parentheses *must* be present.

- The `mytext.c` program tests the `print_text` function.

mytext.c

```
#include <stdio.h>

void print_text(void)
{
  printf(“Hello World\n");
}

int main(void)
{
  print_text();
  return 0;
}
```

Function Definitions

- General form of a ***function definition:***

```
return-type  function-name  (parameters)
{
    declarations
    statements
}
```

Function Definitions

- The return type of a function is the type of value that the function returns.
- Rules governing the return type:
  - Functions may not return arrays.
  - Specifying that the return type is `void` indicates that the function doesn’t return a value.
- If the return type is omitted, the function is presumed to return a value of type `int`.

Function Definitions

- After the function name comes a list of parameters.
- Each parameter is preceded by a specification of its type; parameters are separated by commas.
- If the function has no parameters, the word `void` should appear between the parentheses.

Function Definitions

- The body of a function may include both declarations and statements.
- An alternative version of the `average` function:

```
double average(double a, double b)
{
  double sum;         /* declaration */

  sum = a + b;        /* statement */
  return sum / 2;     /* statement */
}
```

Function Definitions

- Variables declared in the body of a function can’t be examined or modified by other functions.

Function Definitions

- The body of a function whose return type is `void` (a "`void` function") can be empty:

  ```
  void print_text(void)
  {
  }
  ```

- Leaving the body empty may make sense as a temporary step during program development.

Function Calls

- A function call consists of a function name followed by a list of arguments, enclosed in parentheses:

```
average(x, y);
print_value(i);
print_text();
```

- If the parentheses are missing, the function won't be called:

```
print_text;   /*** WRONG ***/
```

Function Calls

- A call of a `void` function is always followed by a semicolon to turn it into a statement:

```
print_count(i);
print_pun();
```

- A call of a non-`void` function produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
avg = average(x, y);
if (average(x, y) > 0)
  printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

Function Calls

- The value returned by a non-`void` function can always be discarded if it’s not needed:

  ```
  average(x, y);  /* discards return value */
  ```

  This call is an example of an expression statement: a statement that evaluates an expression but then discards the result.

Program: Testing Whether a Number Is Prime

- The `prime.c` program tests whether a number is prime:

  ```
  Enter a number: 34
  Not prime
  ```

- The program uses a function named `is_prime` that returns `1` if its parameter is a prime number and `0` if it isn't.
- `is_prime` divides its parameter `n` by each of the numbers between `2` and the square root of `n`; if the remainder is ever 0, `n` isn't prime.

prime.c

```c
#include <stdio.h>

int is_prime(int n)
{
  int divisor;

  if (n <= 1)
    return 0;
  for (divisor = 2; divisor * divisor <= n; divisor++)
    if (n % divisor == 0)
      return 0;
  return 1;
}
```

```
int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);
  if (is_prime(n)==1)
    printf("Prime\n");
  else
    printf("Not prime\n");
  return 0;
}
```

Function Declarations

- C doesn’t require that the definition of a function precede its calls.
- Suppose that we rearrange the `average.c` program by putting the definition of `average` *after* the definition of `main`.

Function Declarations

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %lf and %lf: %lf\n", x, y, average(x,
  y));
  printf("Average of %lf and %lf: %lf\n", y, z, average(y,
  z));
  printf("Average of %lf and %lf: %lf\n", x, z, average(x,
  z));
  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

Function Declarations

- When the compiler encounters the first call of `average` in `main`, it has no information about the function.
- Instead of producing an error message, the compiler assumes that `average` returns an `int` value.
- We say that the compiler has created an ***implicit declaration*** of the function.

Function Declarations

- The compiler is unable to check that we're passing `average` the right number of arguments and that the arguments have the proper type.
- Instead, it performs the default argument promotions and hopes for the best.
- When it encounters the definition of `average` later in the program, the compiler notices that the function's return type is actually `double`, not `int`, and so we get an error message.

Function Declarations

- One way to avoid the problem of call-before-definition is to arrange the program so that the definition of each function precedes all its calls.
- Unfortunately, such an arrangement doesn't always exist.
- Even when it does, it may make the program harder to understand by putting its function definitions in an unnatural order.

Function Declarations

- Fortunately, C offers a better solution: declare each function before calling it.
- A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later.
- General form of a function declaration:

  return-type function-name `(` *parameters* `) ;`
- The declaration of a function must be consistent with the function's definition.
- Here's the `average.c` program with a declaration of `average` added.

Function Declarations

```
#include <stdio.h>

double average(double a, double b);   /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %lf and %lf: %lf\n", x, y, average(x,
  y));
  printf("Average of %lf and %lf: %lf\n", y, z, average(y,
  z));
  printf("Average of %lf and %lf: %lf\n", x, z, average(x,
  z));

  return 0;
}

double average(double a, double b)    /* DEFINITION */
{
  return (a + b) / 2;
}
```

Function Declarations

- Function declarations of the kind we're discussing are known as ***function prototypes*.**
- A function prototype doesn't have to specify the names of the function's parameters, as long as their types are present:

  ```
  double average(double, double);
  ```

- It's usually best not to omit parameter names.

Arguments

- In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter.
- Since the parameter contains a copy of the argument's value, any changes made to the parameter during the execution of the function don't affect the argument.

Arguments

- The fact that arguments are passed by value has both advantages and disadvantages.
- Since a parameter can be modified without affecting the corresponding argument, we can use parameters as variables within the function, reducing the number of genuine variables needed.

Arguments

- Consider the following function, which raises a number x to a power n:

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

Arguments

- Since `n` is a *copy* of the original exponent, the function can safely modify it, removing the need for `i`:

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result = result * x;

  return result;
}
```

The `return` Statement

- A non-`void` function must use the `return` statement to specify what value it will return.
- The `return` statement has the form

  `return` *expression* `;`

- The expression is often just a constant or variable:

```
return 0;
return status;
```

The **`return`** Statement

- `return` statements may appear in functions whose return type is `void`, provided that no expression is given:

```
return;  /* return in a void function */
```

- Example:

```
void print_int(int i)
{
  if (i < 0)
    return;
  printf("%d", i);
}
```

The `return` Statement

- A `return` statement may appear at the end of a `void` function:

  ```
  void print_pun(void)
  {
    printf("To C, or not to C: that is the question.\n");
    return;   /* OK, but not needed */
  }
  ```

  Using `return` here is unnecessary.
- If a non-`void` function fails to execute a `return` statement, the behavior of the program is undefined if it attempts to use the function’s return value.

Program Termination

- Normally, the return type of `main` is `int`:

  ```
  int main(void)
  {
    …
  }
  ```

- Older C programs often omit `main`'s return type, taking advantage of the fact that it traditionally defaults to `int`:

  ```
  main()
  {
    …
  }
  ```

Program Termination

- The value returned by `main` is a status code that can be tested when the program terminates.
- `main` should return 0 if the program terminates normally.
- To indicate abnormal termination, `main` should return a value other than 0.
- It’s good practice to make sure that every C program returns a status code.