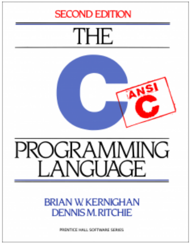
CS 241
Data Organization
Functions

Brooke Chenoweth

University of New Mexico

Fall 2014

Read Kernighan & Richie



1.9 Character Arrays

1.10 External Variables and Scope

2 Types, Operators, and Expressions

Function Declaration

- A function should be declared before it is used in a C file.
- The declaration provides the C compiler with information about the function's return value and parameters.
- The declaration is often called the function's *prototype*.
- The format is (brackets '[ ]' denote an optional word):

```
  [static|extern] type name(parameter list);
```

Function Declaration…

- For example

```c
int average(int a, int b);
```

- This declaration tells the C compiler that the function `average` returns an integer of type `int`, and has two parameters both of type `int`.
- The parameter names aren’t needed in the declaration, but are often useful to the programmer.

Function Declaration

- The optional modifier `static` indicates that the function is private to the current file.
- The optional modifier `extern` indicates that the function is actually defined in another source file.
- Function declarations are often put in a header file.
- If a function isn't declared before it is used, C assumes the function returns an `int`, and doesn't check the parameter types.

Function Definition

- A function definition consists of a function header followed by the function body. The header has the same format as the declaration above. If you define a function before it is used, you don't need the declaration although it's usually best to have one anyway.
- The function body consists of variable definitions followed by statements. Variable definitions have the following form:

```
        [static] type name-list;
```

Function Definition. . .

- The static modifier indicates that the variable is stored in permanent storage, i.e. the next time the function is called the variable has the same value as the last time.
    - `type` is the type of the variable.
    - `name-list` is a comma-separated list of variable names.
    - `Variables` can be set to initial values, e.g.:

      ```
        int foo = 10;
      ```

    - Variables that are not initialized have an undefined value.

“Global” Variables

- A variable declared outside of a function is a global variable – it is allocated permanent storage and is accessible at least to the functions in the same source file.
- The modifier static causes the variable to be private to the current file:

```
static int count = 0;
```

- Variables with initial values are initialized before the program runs.

Global Variables. . .

- The modifier extern indicates that the variable is defined in another source file:

```
extern int count;
```

- The variable must be defined in one (and only one) source file, and it cannot be `static`. Do not put the definition in a header file.

Static Overload

- The C designers loved the word `"static"`. It is used for three different purposes in C:
    - To make a global variable private to the current file.
    - To make a function private to the current file.
    - To allocate a local variable in permanent storage, so it retains its value between function invocations.

Function Prototype and Definition

```
 1  #include <stdio.h>
 2
 3  int foo(int x);
 4
 5  void main(void)
 6  {
 7    int n=5;
 8    printf("%d\n", foo(n));
 9  }
10
11  int foo(int n)
12  {
13    return 2*n;
14  }
```

Function prototype (Line 3) must agree with Function Definition (Lines 11-14)

Output:

```
10
```

Function Prototype and Definition

```c
#include <stdio.h>

int foo(int x);

void main(void)
{
   int n=5;
   printf("%d\n",foo(n));
}

int foo(int n)
{
   return 2*n;
}
```

```c
/* Prototype of foo not needed */
#include <stdio.h>

int foo(int n)
{
   return 2*n;
}

void main(void)
{
   int n=5;
   printf("%d\n",foo(n));
}
```

A Prototype is needed when:

- A function is used in a line *above* where it is defined.
- A function is defined in a *different file*.

No Overloaded Functions in C

```c
#include <stdio.h>

int foo(int n)
{
  return 2*n;
}

int foo(int k, int n)
{
  return k*n;
}

void main(void)
{
  int n=5;
  printf("%d\n", foo(n));
  printf("%d\n", foo(3,n);
}
```

```
foo.c:7:  error:
conflicting types
for ‘foo’
```

. . . and continue with about a half dozen lines of additional error messages. . .

Scope of a Variable in C

All constants and variables have *scope*:

- The values they hold are accessible in some parts of the program, where as in other parts, they don't appear to exist.

Block Scope: variables declared in a block are visible between an opening curly bracket and the corresponding closing bracket.

Function Scope: variables visible within a whole function.

File Scope: variables declared `static` and outside all function blocks.

Program Scope (global variables): variables declared outside all function blocks.

Program Scope and Function Scope

```c
#include <stdio.h>
int a=4;
int b=7;
void foo()
{
  int b = 12;
  a++;
  printf("foo: a=%d, b=%d\n", a, b);
}

void main(void)
{
  foo();
  printf("main: a=%d, b=%d\n", a, b);
}
```

foo does not return a value but it has two *side effects*:

- Sends data to the standard output stream.
- Changes a *global field*: a
- Output:

  ```
  foo: a=5, b=12
  main: a=5, b=7
  ```

Increment Elements of Global Array

```
#include <stdio.h>

#define DATA_COUNT 4
#define MAX_VALUE 32

int x[DATA_COUNT];

int increment(void)
{ /* Adds 1 to each element of global array x[]. */
  /* Returns 1 if any element of x[] is > MAX_VALUE */
  /* Returns 0 if all elements were fine. */
}

void main(void)
{ /* Sets initial values of x[]. */
  /* Calls increment() some number of times. */
}
```

x is a *global array*

Note: this violates our standard: x is too short a name for a *global variable*.

Increment Elements of Global Array

```
int increment()
{ int i;
  for (i=0; i<DATA_COUNT; i++)
  { if (x[i] >= MAX_VALUE) return 1;
    x[i]++;
  }
  return 0;
}
void main(void)
{ x[0] = 20; x[1] = 15; x[2] = 30; x[3] = 2;
  int i;
  for (i=0; i<5; i++)
  { if (increment()) printf("ERROR\n");
    else
    { printf("%d %d %d %d\n", x[0],x[1],x[2],x[3]);
    }
  }
}
```

Increment Elements of Global Array

```
int increment()
{ int i;
  for (i=0; i<DATA_COUNT; i++)
  { if (x[i] >= MAX_VALUE) return 1;
    x[i]++;
  }
  return 0;
}
void main(void)
{ x[0] = 20; x[1] = 15; x[2] = 30; x[3] = 2;
  int i;
  for (i=0; i<5; i++)
  { if (increment()) printf("ERROR\n");
    else
    { printf("%d %d %d %d\n", x[0],x[1],x[2],x[3]);
    }
  }
}
```

Output:

```
21 16 31 3
22 17 32 4
ERROR
ERROR
ERROR
```

Increment Elements of Global Array

```
int increment()
{ int i;
  for (i=0; i<DATA_COUNT; i++)
  { if (x[i] >= MAX_VALUE) return 1;
  }

  for (i=0; i<DATA_COUNT; i++)
  { x[i]++;
  }
  return 0;
}
```

Output:

```
21 16 31 3
22 17 32 4
ERROR
ERROR
ERROR
```

Why might it be better to write the `increment` function like this?

What Does the fibonacci Function Do?

```c
#include <stdio.h>

void fibonacci(int n0, int n1)
{ int n2 = n0 + n1;
  n0 = n1;
  n1 = n2;
}

void main(void)
{ int n0 = 1;
  int n1 = 1;
  int i;
  for (i=1; i<10; i++)
  { printf("%d ", n0);
    fibonacci(n0, n1);
  }
  printf("\n");
}
```

What Does the fibonacci Function Do?

```c
#include <stdio.h>

void fibonacci(int n0, int n1)
{ int n2 = n0 + n1;
  n0 = n1;
  n1 = n2;
}

void main(void)
{ int n0 = 1;
  int n1 = 1;
  int i;
  for (i=1; i<10; i++)
  { printf("%d ", n0);
    fibonacci(n0, n1);
  }
  printf("\n");
}
```

Output:

```
1 1 1 1 1 1 1 1 1
```

Nothing!!!
fibonacci does not return a value.
fibonacci has no side effects.

Fibonacci on Global Variables

```c
#include <stdio.h>
int n0, n1;

void fibonacci()
{ int n2 = n0 + n1;
  n0 = n1;
  n1 = n2;
}

void main(void)
{ n0 = 1; n1 = 1;
  int i;
  for (i=1; i<10; i++)
  { printf("%d ", n0);
    fibonacci();
  }
  printf("\n");
}
```

Output:

```
1 1 2 3 5 8 13 21 34
```

- The body of fibonacci is unchanged from the last program.
- In the last version, n0 and n1 were local to fibonacci.
- In this version, n0 and n1 are global.
- Therefore, this version of fibonacci has side effects.

Fibonacci on Array Parameter

```
#include <stdio.h>

void fibonacci(int n[], int a)
{ /* int n2 = n0 + n1; */
  n[a] = n[a-2] + n[a-1];
}

void main(void)
{ int i, n[11];
  n[0] = 1; n[1] = 1;
  for (i=2; i<11; i++)
  { fibonacci(n, i);
    printf("%d ", n[i-2]);
  }
  printf("\n");
}
```

- `int i` allocates new memory for an integer and the value passed to `fibonacci` is copied into that new memory.
- `int n[]` does not allocate memory for a new array. Arrays are *passed by reference*. `n` in `fibonacci` points to the same memory as `n` in `main`.

Output: 1 1 2 3 5 8 13 21 34