Time and Space

CS 217

Declarations

- A declaration specifies (announces) the properties of an identifier

```
extern int sp;
extern int stack[];
```

- Specifies that "**sp** is an **int**" and "**stack** is an array of **ints**"
- **extern** indicates that they are defined elsewhere
  - outside this routine, or even outside this file

Definitions

- A definition declares the identifier and causes storage to be allocated for it

```
int sp = 1;
int stack[100];
```

- Declare **sp** and **stack**, allocate storage, **sp** is initialized to **1**
- Questions
  - can a variable have multiple declarations?
  - why does a language have declarations?

Function Definitions

- General form

  [type] name (argument-declarations) { body }

  ```
  int twice(int x, double y) { ... }
  ```

- If no return value, type of function should be **`void`**
- **`return`** statements specify return values

```
int twice(int x, double y) {
   return 2*x + y;
}
```


Scope

- The scope of an identifier says where the identifier can be used
- Functions can use global variables declared outside and above them

file **`a.c`**

```
int stack[100];
main() {
   . . .  <-------- stack is visible
}
int sp;
void push(int x) {
   . . .  <-------- stack, sp are visible
}
```


Scope (cont)

- Global variables and functions in other files are made visible with **`extern`**

file **`b.c`**

```
extern int stack[];
voide dump(void) {
   . . . <-------- stack defined in a.c is visible
}
```

Scope (cont)

- Formal parameters and local declarations "hide" outer-level declarations

```
int x, y;
. . .
f(int x, int a) {
    int b;
    . . .
    y = x + a*b;
    if (. . .) {
        int a;
        . . .
        y = x + a*b;
    }
}
```


Scope (cont)

- Cannot declare the same variable twice in one scope

```
f(int x) {
    int x; <-------------- error!
    . . .
}
```


Scope (cont)

- Different name spaces allow same identifier to be multiply declared in a scope

```
struct a {
    int a;
    float b;
} *f;
float a = 1;
typedef int a;
int a(void) {
    char *a;
    {
        double a;
        . . .
    }
}
```

- function and typedef names
- labels
- struct/union tags
- struct/union members

Arguments & Local Variables

- Local variables are temporary (unless declared **`static`**)
  - created upon entry to the function
  - destroyed upon return
  - stored on processes stack
- Arguments are transmitted by value
  - values copied into "local variables"
  - behave like initialized local variables

Example

```
int a, b;
main (void) {
    a = 1; b = 2;
    f(a);
    print(a, b);
}

void f(int a) {
    a = 3;
    {
        int b = 4;
        print(a,
  b);
    }
    print(a, b);
    b = 5;
}
```

Output

```
3 4
3 2
1 5
```


Function Declarations

- Declares the type of the value returned and the types of the arguments

```
extern int f(int, float);
extern int f(int a, float b);
```

- Functions can be used before they are declared, as long as defined in same file or declared **`extern`**
- A function without a declaration…
  - assumes the function returns an **`int`**
  - assumes arguments have the types of the corresponding actual parameters (if not, anything goes)

Static Variables

- **`static`** keyword in a declaration specifies
  - extent: static versus dynamic
  - scope: private versus global
- Scope of static variables
  - within the file or block in which defined
  - used to hide "private" variables
- Extent (lifetime) of static variables
  - allocated at compile time and exist throughout program execution (not on the process stack)

Example

```
void f(int v)  {
    static int lastv = 0;

    print(lastv, v);
    lastv = v;
}
```


Static Functions

file **`stack.c`**

```
static int sp;
static int stack[100];
static void bump(int n); {
    sp = sp + n;
    assert(sp >= 0 && sp < 100);
}
void push(int x) {
    bump(1);
    stack[sp] = x;
}
void pop(void) {
    bump(-1);
    return stack[sp+1];
}
```

Initialization Rules

- Local variables have undefined values
- If you need a variable to start with a particular value, use an explicit initializer
- External and static variables are initialized to 0 by default
  - some consider it bad style to depend on this

C Preprocessor

- Invoked automatically by the C compiler
  - try `gcc –E foo.c`
- C preprocessor manipulates text
  - file inclusion
  - macros
  - conditional compilation

File Inclusion

- Header files contain declarations for one or more C program files
- Names of header files should end in `.h`
- System header files: `< ... >`

```
#include <stdio.h>
#include “mydefs.h”
```

Macros

- Provide parameterized text substitution
- Macro definition

```
#define MAXLINE 120
#define lower(c) ((c)-`A’+‘a’)
```

- Macro replacement

```
char buf[MAXLINE+1] → char[buf[120+1]
c = lower(buf[i]); → c = ((buf[i])-`A’+‘a’);
```


Macros (cont)

- What happens?

```
#define plusone(x) x+1
i = 3*plusone(2);
```

- Use parenthesis liberally

```
#define plusone(x) ((x)+1)
```

- Avoid using argument more than once

```
#define max(a, b) ((a)>(b)?(a):(b))
y = max(i++, j++)
```

becomes

```
y = ((i++)>(j++)?(i++):(j++));
```


Conditional Compilation

- Removing macro definitions

```
#undef plusone
```

- Conditional compilation

```
#ifdef name
#ifndef name
#if expr
#elif expr
#else
#endif
```

- Why use?

Summary

- Important times
  - preprocessing time
  - compile time
    - initialization time
  - runtime
- Kinds of space (memory)
  - **`data`** and **`bss`**: permanent storage
  - **`stack`**: temporary storage (implicitly managed)
  - **`heap`**: dynamic storage (explicitly managed)