ANSI C Programming Language

- A small, general-purpose, initially ***systems programming language***
  - Used for writing the UNIX OS and tools for many computers
  - Now also very popular for general-purpose computing
- A "low-level" language

  datatypes and control structures are close to those on most machines
- Notable features

  pointer (address) arithmetic and operators

  all functions are call-by-value

  simple, 2-level scope structure

  no I/O or memory management facilities (provided by library routines)

  "flexible" type structure
- History

  BCPL → B → C → K&R C → ANSI C
  ~1960 ~1970 ~1972 ~1978 ~1988

C Program Structure

- ***Programs***

  are composed of one or more ***files***

  each file contains ***global variables*** and ***functions***

```
/* this is the function "main" */
int main(int argc, char *argv[]) {
   hello();
   return 0;
}

/* this is the function "hello" */
void hello(void) {
   printf("hello world\n");
}
```

- Execution

  begins by calling **`main`**

  ends when **`main`** returns (or some function calls the library function **`exit`**)

Function Definitions

- General form of an ANSI C function ***definition***

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

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

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

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

- Unlike in Pascal, functions are never defined within functions

Declarations & Definitions

- ***Declaration***: specifies (announces) the ***properties*** of an identifier

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

  specify that "**`sp`** is an **`int`**" and "**`stack`** is an array of **`int`**s"

  `extern` indicates they are ***defined*** elsewhere
    - outside this routine, or even outside this file
- ***Definition***: declares the identifier ***and*** causes ***storage*** to be allocated

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

  declare **`sp`** and **`stack`**, allocates storage, **`sp`** is initialized to 1
- Can a variable have multiple declarations? multiple definitions?
- Why does a language have declarations for variables?

Global Variables & 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() {
    ...
}

int sp;
void push(int x) {
...
}
```

stack is visible

stack, sp are visible

- Global variables and functions in other files are accessed with `extern`

file **b.c**:

```
extern int stack[];
void dump(void) { ... }
```

stack defined in **a.c** is visible here

Scope, cont'd

- Formal parameter 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;
    }
}
```

formal parameter x hides global x

local a hides formal parameter a

- Cannot declare the same variable name ***twice*** in one scope
-

```
f(int x) {
int x;
...
}
```

error!

Function Arguments and Local Variables

- ***Local*** variables are ***temporary*** variables (unless declared static)

  created upon entry to the function in which they are declared

  destroyed upon return
- ***Arguments*** are transmitted ***by value***

  the values of the arguments are ***copied*** into "local variables"
- **Arguments are *initialized local variables***

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

output:

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


Function Declarations

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

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

- A **void** function is a ***procedure***
- A **void** argument list means ***no*** arguments

```
void hello(void)
```

- **A function without a declaration**

  assumes the function returns an **int**

  assumes arguments have the types of the corresponding expressions

  "i = f(2.0, 1);" implies "int f(double, int);"

  if **f** is defined otherwise, ***anything goes!***

  Use prototypes to avoid problems with implicit declarations.

Static Variables

- **`static`** keyword in a declaration specifies

  lifetime: static vs dynamic

  scope: static vs global

- ***Static*** variables are

  allocated at *compile time* and exist throughout program execution

- ***Statics*** are ***permanent***; ***locals*** are ***temporary***

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

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

- Scope of static variables: within the file or block in which they are defined
- What is the difference between static and local variables?

Static Functions

- Scope restricts the visibility of variables and functions

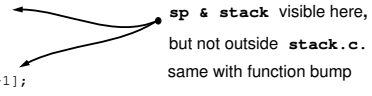
  file **`stack.c`**:

```
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;
}

int pop(void) {
    bump(-1);
    return stack[sp+1];
}
```



`sp & stack` visible here, but not outside **`stack.c`**. same with function bump

- Static ***functions*** are visible only within the file in which they are defined

Example

```
#include <stdio.h>
void f(void);
void g(void);

main() {
    f();
    f();
    g(); }

void f(void)
{   static int i;
    printf("%d \n", i);
    i = 2;
}

void g(void)
{   int foo 10, j;
    printf("%d -- %d\n", foo, j);
    j = 5;
}
```