

C vs Modula-3

feature	C	Modula-3
safe	no	yes
efficient	yes	yes
garbage collection	no	yes
static typechecking	mostly	yes
enforced interfaces	no	yes
concurrency	no	yes
Huh?		
widely available	yes	no
everyone knows it	yes	no
software tools	yes	some
good for a summer job	yes	no

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

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

C Program Structure

- *Programs*
 - are composed of one or more *files*
 - each file contains *global variables* and *functions*

```
int a, b;          /* global variables */

int main(int argc, char *argv[]) {
    hello();
    return 0;
}

void hello(void) {
    printf("hello world\n");
}
```
- *Execution*
 - begins by calling `main`
 - ends when `main` returns (or some function calls the library function `exit`)

Scope

- How do functions defined in different files communicate?
 - by calling one another (parameter passing and return values
 - through global (externally declared) variables
- External variables
 - Externally declared versus `extern`?
 - Can we have multiple declarations of an externally defined variable within a file?
 - What if an external declaration is not initialized? Is it treated as defined?
- So which functions and data may a function reference?
 - determined by the **scope** of identifiers

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 `ints`"
 - `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;
int ptr = 1;
int stack[100];
```

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

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

```
int x;
...
}
```

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() {
  ...
}
```
- Global variables and functions in other files are made available with `extern`

```
file b.c:
extern int stack[];
void dump(void) { ... }
```
- Global variables and functions in other files are made available with `extern`

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:
3 4
3 2
1 5

Copyright ©1995 D. Hanson, K. L. & J. P. Singh

Computer Science 217: Function Arguments and Local Variables

Page 23

September 7, 1997

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;
 }
}
```
- Cannot declare the same variable name **twice** in one scope
  - `f(int x) {`  
  `int x;`  
  `...`  
}
  - Different **name spaces** allow same identifier to be multiply declared in a scope
    - function and typedef names; labels; struct/union tags; struct/union members

Copyright ©1995 D. Hanson, K. L. &amp; J. P. Singh

Computer Science 217: Scope, cont'd

Page 22

## 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
  - scope versus lifetime
- What if a variable is declared `extern` inside a function?

Copyright ©1995 D. Hanson, K. L. &amp; J. P. Singh

Computer Science 217: Static Variables

Page 25

September 7, 1997

## 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)
```
- Unlike Pascal, 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 expressions
  - "`1 = f(2.0, 1);`" implies "`int f(double, int);`"
  - if `f` is defined otherwise, **anything goes!**

Copyright ©1995 D. Hanson, K. L. &amp; J. P. Singh

Computer Science 217: Function Declaration

Page 24

## Initialization Rules

---

- **Local** variables have ***undefined values***
- 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 rely on this feature

## 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];
}
```
- **Static *functions*** are visible only within the file in which they are defined