V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 7

—Slide 1—

The C *Programming Language*
Language Survey 2

- General Purpose "High-Level" Programming Language.

  Not 'very' high-level: Has many features allowing access to low-level operations. Similar to Bliss, in this regard.

- Originally designed by *Dennis Ritchie.* First implementation on the `UNIX` operating system on the DEC PDP-11.

- **Short History**
  - `BCPL`, *Martin Richards*. Late 60's.
  - `B`, *Ken Thompson*. 1970, First `UNIX` implementation on PDP 7.
  - `BCPL` & `B` = "typeless"

—Slide 2—

History of C

- C, designed by Dennis Ritchie.
- Typed (A hierarchy of derived data-types.)
- ANSI C, (1983-1988)
  (Syntax of Function Declaration, Elaborate Preprocessor, Arithmetic, Standard Library.)
- "*Algol Like*"
  Similar to Algol, PL/1, Bliss, Pascal, Ada, Modula, ...
  Features: Variable Declarations, Imperative, Block-Structured, ...

—Slide 3—

SYNTAX

- **Declarations:** *Variables*

```
<type-name> <name> { ',' <name> } ';'
```

Sequence of **<name>**s separated by commas and terminated by a semicolon.

```
int i,j;
int A[3], B[5][7];
int *p;      /* pointer to an integer*/
```

—Slide 4—

SYNTAX

- **Declarations:** *Functions*

```
<result-type> <name>(<formal-pars>){
   <declaration-list>
   <statement-list>
}
```

Function Procedure:
`<formal-pars>` $\mapsto$ `<result-type>`
Default Result Type = `int`

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

- **True Procedures**
  A result type '`void`' indicates that a "function" is a proper procedure with no result.

—Slide 5—

Assignment Operator

- Assignment statement is a `C` expression.

  ```
  <expression-1> = <expression-2>
  ```

  R-Value of `<expression-2>` is put in the location given by the *L-Value* of `<expression-1>`.

- **Example**

  ```
  c = getchar();

  while((c = getchar()) != EOF)
    putchar(c);

  for(A[0] = X, i = n; X != A[i]; --i);
  return i;
  ```

  Linear Search with a sentinel!

—Slide 6—

Syntax of Statements

```
<stmt-list> ::= <empty> | <stmt-list> <statement>

<statement> ::=
   ;
 | <expression> ;
 | {<stmt-list>}
 | if(<expression>)<statement>
 | if(<expression>)<statement> else <statement>
 | while(<expression>) <statement>
 | do <statement> while (<expression>)
 | for(<opt-exp>;<opt-exp>;<opt-exp>)<statement>
 | switch (<expression>) <statement>
 | case <const-exp> : <statement>
 | default : <statement>
 | break;
 | continue;
 | return;
 | return <expression>;
 | goto <label-name>;
 | <label-name> : <statement>;
```

—Slide 7—

Control Structure

- **Compound Statement**

```
{
  x = y = z = 0;
  i++;
  printf(...);
  i = x;
}
```

1. Semicolon is a statement terminator, not separator.
2. Braces `{` and `}` group declarations and statements into a block.

- **Conditional Statement**

```
if(n > 0)
  if(a > b)
    z = a;
  else
    z = b;
```

Dangling `else` is resolved by associating the `else` with the closest previous `else`-less `if`.

—Slide 8—

Control Structure

- **Conditional Statement:** `else if`

```
if(x == 0)
  y = 'a';
else if(x == 1)
  y = 'b';
else if(x == 2)
  y = 'c';
else if(x == 3)
  y = 'd';
else
  y = 'z';
```

- **Conditional Statement:** `switch`

```
c = getchar();
switch(c){
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
   ndigit[c - '0']++;
   break;
case ' ': case '\n': case '\t':
   nwhite++;
   break;
default:
   nother++;
   break;
}
```

—Slide 9—

Iterative Statement

- `while` & `for`

```
A[0] = X;
i = n;
while(X != A[i])
  --i;
return i;
```

```
for(A[0] = X, i =n;
     X != A[i]; --i)
   ;
return i;
```

```
A[0] = x;
i = n;
for(;;){
  if(X == A[i]){
     return i; break;
  }
  --i;
}
```

—Slide 10—

`break`, `continue` & `goto`

- A `break` causes the innermost enclosing loop or `switch` to be exited immediately.
- A `continue` statement causes the next iteration of the innermost enclosing loop to begin
  1. `while` & `do`: The test part is executed immediately.
  2. `for`: The increment step is executed immediately.
- A `goto` interrupts normal control flow. `goto` L causes the control to go to the statement labeled L.

—Slide 11—

Examples of `break` *&* `continue`

```
for(i = 0; i < n; i++){
  if(a[i] < 0)
    break;
  ...
}
```

```
for(i = 0; i < n; i++){
  if(a[i] < 0)
    continue;
  ...
}
```

```
for(;;c = getchar()){
  if(c == ' '||c == '\t')
    continue;
  if(c != '\n')
    break;
  ++lineno;
}
```

Skips over blanks, tabs & newlines, while keeping track of line numbers.

—Slide 12—

Program Structure

- `C` is **Block-Structured**
- Local declarations can appear within any **block** (Grouping of statements).
  Compound Statement

```
{
  <declaration-list>
  <statement-list>
}
```

- A `C` program consists of global declarations of: *procedures*, *types* and *variables*
- *Types* and *variables* can be declared local to a procedure.
- A procedure cannot be declared local to another.

—Slide 13—

Scope in C

- C is statically scoped
  Scope of a declaration of X in a block is *i) that block + ii) all its nested blocks − iii) all the nested blocks in which* X *is redeclared.*

```
    int main(void)
   |{
   |    int i;          /* Scope of i = */
   |    for( ... )      /*  A + B - C - D */
   |  | {
   |  |  int c;
   |  |  if( ... )
   |  |    |{
   | B|  C |  int i;  /* Scope of i =  */
 A |  |    |  ...     /*    C          */
   |  |    |}
   |  |  ...
   |  | }
   |    while( ... )
   |  | {
   | D|   int i;      /* Scope of i =  */
   |  |   ...         /*    D          */
   |  | }
   |   ...
   |}
```

—Slide 14—

Automatic and External Variables

- Variables declared in a function are local to that function.
- Other functions can have access to them indirectly, if they are passed as parameters.

  Or directly by name, if they are explicitly redefined as `extern`'s.
- `extern` variables are globally accessible and remain in existence permanently.

```
int getline(char line[], int maxline);

main(){...
  char line[MAXLINE];
  ...
  getline(line, MAXLINE);
}
int getline(char s[], int lim){
    ...
}
```

—Slide 15—

Usage of `extern`*: Example*

```
char line[MAXLINE];
...
int getline(void);

main(){...
   extern char line[];
    ...
   getline();
   ...
}
int getline(void){...
   extern char line[];
   ...
}
```

- **Note:** Usually all `extern` declarations are collected in a "header" file, and included by "`#include`" (compiler declarative) in each source file.

—Slide 16—

Static Variables

- **External Static**
  A static declaration, applied to an external variable, limits its scope only to the rest of its source file.
  Provides a way to hide information

```
static char buf[BUFSIZE];
static int bufp = 0;

int getch(void{...}
void ungetch(int c){...}
```

- `buf` & `bufp` can be shared by `getch` & `ungetch`. But **not visible to the user of** `getch` & `ungetch`

—Slide 17—

Static Variables

- **Internal Static**
  Like automatic variables, they are local to a particular function.

  But they remain in existence from one activation to the next.

- Provide **permanent private storage** within a single function.

[End of Lecture #7]