G22.2210

Programming Languages: PL I

B. Mishra
New York University.

Lecture # 4

—Slide 1—

The C *Programming Language*
Language Survey 1

- 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>` ↦ `<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, <u>not</u> 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.

—Slide 18—

Types

- A type has two components:
  1. A set, S of elements
  2. A set of operation on S
- **Basic Data Types:**

`char`	A single byte, holds one character (`signed` or `unsigned`.)
`int`	Integers. Qualifiers: `short` & `long`. Also, `signed` & `unsigned`.
`float`	Single precision floating point.
`double`	Double precision floating point.
`long double`	Extended precision floating point.

(*The size of Integers and floating points are* **implementation-defined.**)

—Slide 19—

Types (contd)

- Arithmetic Operators:
  `+` *(Addition)*, `-` *(Subtraction)*,
  `*` *(Multiplication)*, `/` *(Division)*,
  `%` *(Modulus)—Cannot be applied to* `float` *or* `double`.
- Relational and Logical Operators:
  `>`, `>=`, `<`, `<=`,
  `==`, `!=`, `...`
- Constants:

◦ Integers	`1234`	Type = `int`
	`123456789L`	Type = `long`
	`123U`	Type = `unsigned`
◦ Doubles	`123.4`	Type = `double`
	`1e-2`	Type = `double`

—Slide 20—

Types: Constants

- Constants:
  - Characters

`'X'`	Type = `char`
`'\n'`	Type = `char`

  - Strings

`"Hello, World!"`	Type = `char*`
`"X"`	Type = `char*`

- Enumeration Constants:

```
enum boolean {NO, YES};

enum escapes {BELL = '\a', BACKSPACE = '\b',
              TAB = '\t', NEWLINE = '\n',
              VTAB = '\v', RETURN = '\r'};

enum months  {JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC};
```


—Slide 21—

TYPE CONVERSION

- *"Narrow-To-Wide"* Rule
  If an operator has operands of different types, then they are converted into a common type, automatically, by interpreting the "narrower" operand as a "wider" one.

```
float f; int i;
f + i      /* converted into float */
```

- **Information Loss:**
  Longer integers are converted to shorter ones by dropping excess higher order bits.

```
char c; int i;
i = c; c = i;  /* No information loss */
c = i; i = c;  /* Higher order bits--lost */
```

- **Explicit Conversion** Type Casting

```
(<type-name>)<expression>

int n; double a;
a = sqrt((double) n);
```


—Slide 22—

Composite Types: Arrays & Pointers

- Array

  `<type> <name>[<size>]`

  Defines an array (`<name>`) of size = `<size>` with entries of type = `<type>`.
  Entries are numbered from 0 to $< size > -1$.

```
int a[10];   \* a[0], a[1], ..., a[9] *\
```

- Pointer
  A group of cells (2 or 4) that can hold an address.
  `&` = referencing operator, and `*` = dereferencing operator

```
int x, y, a[10];
int *ip, *pa;

ip = &x; y = *ip; pa = &a[0];
y = *(pa + 3);
```

Note: `*(pa + 3)` ≡ `a[3]`

—Slide 23—

Multidimensional Arrays

- Multidimensional arrays are defined as arrays of arrays.

```
int daytab[2][13] = {
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,28,31,30,31,30,31,31,30,31,30,31}
};
int leap; int days;
leap = year%4 == 0;
days = daytab[leap][i];/* Not daytab[leap,i] */
```

A two-dimensional array is really a one dimensional array, *each of whose element is an array.*

- Pointer Array

```
int a[10][20];
int *b[10];
```

Note: `a[3][4]` and `b[3][4]` are syntactically legal.

- `a` = a true 2D array: 200 `int`-sized locations have been set aside.
- `b` = a 1D array of pointers: the pointers are not initialized.

—Slide 24—

Strings

- An array of `char`s
- A String Constant:

```
"I am a string"    "A"
```

- **Definition**

```
char amessage[] = "now is the time";
char *pmessage = "now is the time";
```

Note: `pmessage` is a pointer to a character array.

- String Copy: copy `t` to `s`

```
void strcpy(char *s, char *t){
   while((*s++ = *t++) != '\0');
}
```


—Slide 25—

Structures

- `struct` = A heterogeneous collection of one or more variables, possibly of different types.
  Similar to PASCAL `record`.

```
struct point{  /* point is a structure tag */
  int x;
  int y;
};             /* x, y = members */

struct point maxpt = {320, 320};
```

- Structure may be copied and assigned to, passed to functions and returned by functions.
- Structure Selector:
  A member of a particular structure.

```
<structure-name>.<member>
dist = sqrt((double) pt.x * pt.x
              + (double) pt.y * pt.y);
```


—Slide 26—

Union

- A `union` may hold objects of different types and sizes.
- similar to variant records in PASCAL.

```
union u-tag{
  int   ival;
  float fval;
  char  *sval;
} u;
```

- `u` can be of type `int`, `float` or a `char`-pointer.
- The usage must be *consistent*: The type received must be the type most recently stored.

—Slide 27—

Type Abstraction

- Just as subroutines provide procedural abstraction, *abstract data types* provide *type abstraction*.
- `C` provides a facility called `typedef` for creating new data type names.

```
typedef int  Length;
typedef char *String;
Length len; String lineptr[MAXLINES];
```

- *Type Equivalence*
  1. **Name Equivalence:** Two objects have same types if they have same type names.
  2. **Structural Equivalence:** Two objects have same types if they have the same structures.
- `C` uses structural equivalence—
  However, `struct`s, `union`s and `enum`s with *distinct tags* are *distinct*.

—Slide 28—

Procedure Declarations

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

int succ(int i){
  return (i+1)%size;
}
```

- Missing result-type is by default `int`.
- A result-type `void` indicates a proper procedure with no result.
- `C` uses *call-by-value* for parameter passing. *Call-by-reference* can be simulated by calling with pointers.

—Slide 29—

Parameter Passing in `C`

- **Call-by-Value**:
  The R-values of actual parameters are computed and assigned to formal parameters just before activating the function call.
- The following program has no effect:

```
void bad-swap(int x, int y){
   int z;
   z = x; x = y; y = z;
}
int a = 0; int b = 1;
bad-swap(a, b);
```

- Simulating call-by-reference:

```
void swap(int *px, int *py){
   int z;
   z = *px; *px = *py; *py = z;
}
int a = 0; int b = 1;
swap(&a, &b);
```


—Last Slide—

Summary

○ `C` Design

- **GOOD**
  1. Simple, Versatile
  2. Block-Structured (Algol-like Syntax)
  3. Rich type structure
  4. Powerful environments
     (UNIX, Debugger, Separate Compilation, ...)
- **BAD**
  1. Too simple for large applications
  2. Quirky Syntax, Poor Readability
  3. Weakly-typed, Error-Prone
     (NO Array Bound Checking, etc.)
  4. No module structure to organize the programs.

[End of Lecture #4]