ESSENTIALS OF THE C PROGRAMMING LANGUAGE

BY

ISIDORO COUVERTIER REYES

OBJECTIVES

AT THE END OF THIS TRAINING THE PARTICIPANTS WILL BE:

- **ABLE TO RECOGNIZE AND UNDERSTAND THE MOST FUNDAMENTAL C LANGUAGE CONSTRUCTS.**
- **ABLE TO UNDERSTAND AND WRITE C LANGUAGE PROGRAMS.**
- **ABLE TO LEARN ON ITS OWN THE REST OF THE MORE ADVANCED FEATURES OF THE C LANGUAGE NOT COVERED IN THIS TRAINING BECAUSE OF TIME CONSTRAINS.**

OUTLINE

- **INTRODUCTION**
- **STRUCTURE OF A C PROGRAM**
- **VARIABLE DECLARATIONS**
- **PRIMITIVE DATA TYPES (void, char, int, long, unsigned, float, double, *etc.*)**
- **BASIC I/O OPERATIONS (cout AND cin)**
- **DIRECTIVES (#include, #define)**
- **HEADER FILES (math.h, stdlib.h, *etc.*)**
- **OPERATORS**
- **DECISION STRUCTURES (if AND switch STATEMENTS)**
- **LOOP STRUCTURES (for, while, do-while)**
- **FUNCTIONS**

OUTLINE

- **VARIABLE ATTRIBUTES (***name, type, storage class, storage duration, and scope***)**
- **POINTERS (***passing parameters by-reference, functions as parameters)*
- **USER-DEFINED TYPES (typedef)**
- **RECORDS (struct STATEMENT)**
- **FILE I/O**
- **LINKED LISTS**

AN INTRODUCTION TO C

- **EVOLVED FROM BCPL (1967) AND B (1970) WHICH WERE BOTH TYPELESS.**
- **DEVELOPED BY DENNIS RITCHIE IN 1972 AND IMPLEMENTED ON PDP-11.**
- **THE DEVELOPMENT LANGUAGE OF THE UNIX OPERATING SYSTEM.**
- **MOST NEW OPERATING SYSTEMS ARE DEVELOPED IN C.**
- **A STRONGLY TYPED LANGUAGE.**
- **PROVIDES FOR DYNAMIC MEMORY ALLOCATION THROUGH THE USE OF POINTERS.**

STRUCTURE OF A *C* PROGRAM

- **preprocessor directives**
- **global variables;**
- **functions**

```
{
        local variables;
        program block;
}
```

- **main()**

```
{
        local variables;
        program block;
}
```

EXAMPLE

```
#include <iostream.h>  //note no semicolon ;
#define MAX 15 //note no  ;
void hello(int k)  //note no  ;
{
        cout << "Hewlett Packard " << k);
}

main() // note no ;
{
        int i = MAX;
        hello(i);
}
/* COMMENTS IN C MUST BE
 ENCLOSED LIKE THIS ONE BUT WE
ARE BORROWING THE // FROM C++ */
```

ALTERNATE STRUCTURE OF A C PROGRAM

- **preprocessor directives**
- **global variables;**
- **function prototypes;**
- **main()**
  {
  local variables;
  program block;
  }

- **functions**
  {
  local variables;
  program block;
  }

EXAMPLE

```
#include <iostream.h>
#define MAX 15
void hello(int k);//note the use of the semicolon

main()
{
        int i = MAX;
        hello(i);
}

void hello(int k)  //note no  ;
{
        cout << "Hewlett Packard " << k;
}
```

VARIABLE DECLARATIONS

STRUCTURE OF A DECLARATION
type var1_name, var2_name, …,varn_name;

Examples:
#define N 3
•int a, b = N + 5, c, array1[10], mat[N][N];
•float x, y, z;
•char t1, t2, t3;
•double arrd[N*N] = {0.0}, v2;
•int *p = &a , q, *z[7];
• FILE *fin, *fout; // pointers to FILE
•fin = fopen(“input_file_name”, “r”);
•fout = fopen(“output_file_name”, “w”);
•ifstream inf (“inputfile”);// c++
•ofstream outf (“outputfile”);// c++

PRIMITIVE DATA TYPES

- **void // generic data type**
- **char // character type, takes up one byte**
- **int // same as short for 2 byte machines or as long for 4 byte machines**
- **long // integer from -2147483648 to 2147483647 for four bytes**
- **unsigned // approx. twice the positive integer range since no sign bit**
- **float // 6 significant figures and range of 10^-38 to 10^+38**
- **double // 15 significant figures and range of 10^-308 to 10^+308**
- **long double // long double type**
- **short, signed or unsigned char, etc.**

BASIC I/O OPERATIONS C

```
#include <iostream.h>
main()
{
    int i;
    float f;
    char c;

    cout << “Enter an integer, a real,
                and a character: “;
    cin >> i >> f >>c;
    cout << “\nThe values for i, f, and c are:
          << i << f <<c;
    // The \n is the newline character.
    return 0;
}
```

BASIC I/O OPERATIONS C++

```
#include <iostream.h>
main()
{
    int i;
    float f;
    char c;

    cout << “Enter an integer, a real,
                and a character: “;
    cin >> i >> f >> c;
    cout << “\n The values for i, f, and c are:”
              << i <<‘  ‘ << f <<‘  ‘ << c <<“\n”;
    //The \n is the newline character.
    return 0;
}
```

DIRECTIVES

- **#include <header filename>**
  Used to include optimized functions that are used by many programs. Among the header filenames we find stdio.h, math.h, and stdlib.h. If <> used as shown above, then will look for header file in all system dependent directories but not in current. If “” used instead, then first look for file in current directory and then in all system dependent directories.
- **#define NAME value**
  Used to declare constant values to be used throughout the program. Try to stick to all uppercase names.

  Example:

  #define PI 3.141592653589

HEADER FILES

- **stdio.h contains the definitions of printf, scanf, getc, getchar, gets, fread, fwrite, putc, puts, etc.**
- **iostream.h contains the definition of cout, cin, etc. (C++)**
- **math.h contains the definitions of pow, sqrt, exp, log, sin, asin, cos, acos, tan, atan, etc.**
- **There are many more header files in C. See the handouts.**

OPERATORS

() [] . ->
++ -- * & ! ~ + - sizeof (typecast)
*** / %**
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= %= /= >>= <<= &= ^=
,

if STRUCTURE

```
if  (condition)
{
   statements;
}
```

Example:

```
#include <iostream.h>
main()
{    int a;
     if  (a % 2 == 1)
     {
          cout <<“The value of a is odd.”;
     }//body is one statement, {} not needed
     return 0;
}
```

if STRUCTURE

Example:

```
#include <iostream.h>
main()
{   int a;
    if  (a % 2 == 1)
    {
        cout<<“The value of a is odd.”;
    }
    else
    {
        cout <<“The value of a is even.”;
    }
    return 0;
}
```

switch STATEMENT

```
switch (a % 2)
{
    case 1:
        cout<<“The value of a is odd.”;
        break;
    case 0:
        cout<<“The value of a is even.”;
        break;
    default:
        //not needed for this example
}
```

for STRUCTURE

```
for(cv=initial;final expression on cv;cv=cv + 1)
{
        statements;
}
```

Example:

```
for (i=1; i<=N; i++)
{
        cin >> array[i];
        array[i] = array[i]*array[i-1] + 4;
}
//note that an integer is expected for the
//control variable and that arrays begin with 0
```

for STRUCTURE

Assuming all arrays are declared and initialized correctly, e.g. array c is loaded with zeroes, **int c[N][N] = {0};**.

```
for (i = 0; i < N; i++)
{
      for (j = 0; j < N; j++)
      {
            for (k = 0; k < N; k++)
            {
                  c[i][j] = c[i][j] + a[i][k]*b[k][j];
            }
      }
}
```

while STRUCTURE

```
cv = initial;
while (condition)
{
      statements; //must include an update of cv
}
```

Example:

```
i = 1;
while (i <= N)
{
      cin >> array[i];
      array[i] = array[i]*array[i-1] + 4;
      i++;
}
```

do-while STRUCTURE

```
cv = initial value;
do
      statements; //must include an update of cv
while (some condition);
```

Example:

```
i = 1;
do
       cin >> array[i];
       array[i] = array[i]*array[i-1] + 4;
       i++;
while (i  <= N);
```

THE break AND continue STATEMENTS

- **break - The break statement causes the innermost enclosing loop or switch statement to be exited.**

- **continue - The continue statement causes the current iteration of a loop to stop and the next iteration of the same loop to begin immediately.**

break AND continue EXAMPLES

```
for (i = 1; i <= 100; i++)
{
    if (i % 5 == 0)
    {
            break;
    }
    else
    {
        cout << i << "\n";
    }
}
```

vs.

```
for (i = 1; i <= 100; i++)
{
    if (i % 5 == 0)
    {
        continue;
    }
    else
    {
        cout << i << "\n";
     }
}
```

FUNCTIONS

```
return_type  name(parameter list with types)
{
        local variable declarations;

        function block;
        return expression;
}
```

Example:

```
float maximize(float x, float y) //no type then int
{
        //no local variables except for x and y
        if ( x > y)
            return  x;
        else
            return  y;
}
```

VARIABLE ATTRIBUTES

- **name**
- **type**
- **storage class**
- **storage duration**
- **scope of visibility**

STORAGE CLASS

1. **auto - All local variables without any class specifier or preceded with the auto class specifier are of this storage class. They are created when the program enters the block in which they are declared and destroyed when the block is exited.**

2. **extern - All global variables whether they are preceded or not by the extern class specifier. Thus no need to specify it. Created when program begins execution and destroyed when execution stops. Used to allow other programs to access variables.**

STORAGE CLASS

3. **static - Must be specified. Mostly used for local variables though it is also used for global variables and functions which must be hidden from external programs. Created when program begins execution and destroyed when execution stops.**

4. **register - Used to tell the compiler that it is desired that a variable be assigned a high speed CPU register. Compiler will most probably ignore it and perform its own optimizations.**

STORAGE DURATION

1. **Automatic - Variables of automatic storage duration exist only when the block in which they are declared is executing. They are created when the block is entered and destroyed when the block is exited. All variables of storage class auto have automatic storage duration.**

2. **Static - All global variables by definition and all local variables declared with the static class specifier have static storage duration. This means that they are created when program begins execution and initialized to zero unless done otherwise. Static local variables can only be accessed within the block that declares them.**

SCOPE OF VISSIBILITY

- **File scope - all global variables and all function definitions have file scope and are thus known throughout the file unless a global is hidden by a local.**
- **Function scope - only labels have function scope and are thus known throughout the function but not outside of it.**
- **Block scope - local variables declared within a block are only known within that block even if declared as static, which by definition are created from the moment the program begins execution.**

SCOPE OF VISSIBILITY

- **Function prototype scope - parameter names in the prototype are only known within the prototype and are ignored by the compiler.**

```
#include <iostream.h>
int x = 1;
void f1(void)
{    int x = 25;
     cout << “Value for x on entering f1 is ” << x << “.\n”;
     ++x;
     cout << “Value for x on exiting f1 is “ << x << “.\n\n”;
}
void f2(void)
{    static int x = 50;
     cout << “Value for static x on entering f2 is “ << x << “.\n”;
     ++x;
     cout << “Value for static x on exiting f2 is “ << x << “.\n\n”;
}
void f3(void)
{    cout << “Value for global x on entering f3 is  “ << x << “.\n”;
     x *= 10;
     cout << “Value for global x on exiting f3 is “ << x <<“.\n\n”;
}
main()
{    int x = 5;
     cout << “Value for x in outer main block  is “ << x << “.\n”;
     {    int x = 7;
          cout << “Value for x in inner main block  is “ << x << “.\n”;
     }
     cout << “Value for x in outer main block  is “ << x << “.\n\n”;
     f1();
     f2();
     f3();
     f1();
     f2();
     f3();
     cout << “Value for x in  main block  is “ << x << “.\n”;
     return 0;
}
```

ADAPTED FROM *C HOW TO PROGRAM* BY DEITEL AND DEITEL

PROGRAM OUTPUT

Value for x in outer main block is 5.
Value for x in inner main block is 7.
Value for x in outer main block is 5.

Value for x on entering f1 is 25.
Value for x on exiting f1 is 26.

Value for static x on entering f2 is 50.
Value for static x on exiting f2 is 51.

Value for global x on entering f3 is 1.
Value for global x on exiting f3 is 10.

Value for x on entering f1 is 25.
Value for x on exiting f1 is 26.

Value for static x on entering f2 is 51.
Value for static x on exiting f2 is 52.

Value for global x on entering f3 is 10.
Value for global x on exiting f3 is 100.

Value for x in main is 5.

POINTERS

- **A pointer is an address of an object in memory.**
- **An array name is itself a (constant) pointer.**
- **Passing parameters to a function by-reference is simulated through the use of pointers.**
- **Pointers allow for dynamic memory allocation, that is, requesting memory on demand and (sort of) releasing it when not needed. This is as opposed to arrays where memory is reserved from the beginning and can not be released even if not needed.**

SAMPLE PROGRAM USING POINTERS

```
//Adapted from A Book on C  by A. Kelley and I. Pohl
#include <iostream.h>
#include <string.h>
main()
{
    char *p, c = 'a', s[100];
    p= &c;
    cout<<*p<< (char)(*p + 1) << ( char)(*p + 2)<<'  ';
    strcpy(s, "ABC");
    cout<<s <<'  '<<(char)(*s+6)<<(char)(*s+7)<< s+1;
    strcpy(s, "Antes, ahora y siempre: Colegio de
                  Mayaguez");
    p = s + 15;
    for ( ; *p != '\0'; ++p)  {
        if (*p == 'e')
              *p = 'E';
        if (*p == ' ')
               *p = '\n';        }
    cout << s;
    return 0;
}
```

POINTERS: SAMPLE PROGRAM OUTPUT

abc ABC GHBC
Antes, ahora y siEmprE:
ColEgio
dE
MayaguEz

TABLE OF EXPRESSIONS USING POINTERS

Declarations and Initializations		
int i= 3, j = 5, *p = &i, *q = &j, *r; double x;		
Expression	*Equivalent expression*	*Value*
p == &i	p == (&i)	1
**&p	*(*(&p))	3
r=&x	r=(&x)	illegal
7**p/*q+7	trouble	
7**p/ *q+7	(((7*(*p)))/(*q))+7	11
(r=&j)=*p	(*(*r=(&j))) *= (*p)	15

From *A Book on C* by A. Kelley and I. Pohl

PASSING PARAMETERS BY-REFERENCE

```
#include <iostream.h>
int cube_by_value(int a) {
        return  a*a*a;}
void cube_by_reference(int *b) {
        *b = *b * *b * *b;}
main(){
        int i = 3,  j = i, k = 0;
        cout<<“i=“<< i<<“ j=“<< j<<“ k=“
              <<k<<“\n”;
        k = cube_by_value(j);
        cout<<“i=“<< i<<“ j=“<< j<<“ k=“
              <<k<<“\n”;
        cube_by_reference(&i);
        cout<<“i=“<< i<<“ j=“<< j<<“ k=“
              <<k<<“\n”;
        return 0;
}
```

OUTPUT

i=3, j=3, k=0
i=3, j=3, k=27
i=27, j=3, k=27

PASSING A FUNCTION AS A PARAMETER

```
double f1(double  f2(double), int i, int j)
{
      int k;
      double sum = 0;

      for (k = i; k <= j; k++)
      {
          sum = sum + pow(f2(k),  2);
      }
      return sum;
}
```

or

double f1(double (*f)(double), int i, int j)

but not

double f1(double *f(double), int i, int j)

which says a pointer to double is returned.

USER DEFINED TYPES USING typedef

```
typedef  char * charstring;
typedef  int  computers, workstations;
typedef  double  scalar;
typedef  scalar  array[N];
typedef  scalar  matrix[N][N];
typedef  array   matrix_too[N];
typedef  float  (*pfd) (float);

charstring s1 = "IJBU",  s2 = "RUM";
computers  network;
array  a;  //equivalent to:  double  a[N];
pfd  f1;  //equivalent to: float (*f1)(float);
```

CREATING RECORDS USING struct

```
struct  struct_specifier
{
    declarator_list;
};
```

Example:

```
struct date_of_year
{
     char  month[9];
     char  *day_of_week;
     int  day_of_month;
};

struct date_of_year birhtdate ;
birthdate.month = “December”;//WRONG
birthdate.day_of_week = “Wednesday”;
birthdate.day_of_month = 15;
```

CREATING RECORDS USING struct

Or better yet

```
struct date_of_year
{
      char  month[9];
      char  *day_of_week;
      int  day_of_month;
};

typedef struct date_of_year date;

date birhtdate ;
strcpy(birthdate.month ,“December”);
birthdate.day_of_week = “Wednesday”;
birthdate.day_of_month = 15;
```

ACCESSING MEMBERS OF A STRUCTURE USING POINTERS

```
date  *p = &birthdate;

cout << p -> month <<
     p -> day_of_week << p -> day_of_month;
```

Which is equivalent to:

```
cout << (*p).month <<
     (*p).day_of_week << (*p).day_of_month;
```

but not to:

```
cout << (*p.month) <<
     (*p.day_of_week) << (*p.day_of_month);
```

which is an error since only a structure can be used with the "." operator and p is a pointer..

ANOTHER struct EXAMPLE

Declarations and Assignments		

struct employee temp, *p = &temp;
temp.sex = 'F'; //*assume* char *sex;
temp.last = "Perez"; // *assume* char *last;
temp.age = 23; // *assume* int age;		

Expression	*Equivalent expression*	*Value*
temp.sex	**p -> sex**	**F**
temp.last	**p -> last**	**Perez**
(*p).age	**p -> age**	**23**
***p->last + 1**	**(*(p -> last)) + 1**	**Q**
***(p->last+2)**	**(p -> last)) [2]**	**r**

Adapted from *A Book On C by* Kelley and Pohl

struct FOR PREVIOUS EXAMPLE

```
struct employee
{
    char  *last;
    char  sex;
    int  age;
};
```

FILE INPUT/OUTPUT

```
#include <iostream.h>
main()
{
    int a, sum = 0;
    // how to open an input file
    ifstream filein (“input_file_name”);
    // how to open an output file
    ofstream fileout (“output_file_name”);
    filein >> a;
    while (!filein.eof()){
        sum += a;
        filein >> a;}
    fileout << “sum = “ << sum << ‘\n’;
    filein.close();
    fileout.close();
    return 0;
}
```

FILE INPUT/OUTPUT

FILE MODES:

“r”	**open text file for reading**
“w”	**open text file for writing**
“a”	**open text file for appending**
“rb”	**open binary file for reading**
“wb”	**open binary file for writing**
“ab”	**open binary file for appending**
“r+”	**open text file for reading and writing**
“w+”	**open text file for writing and reading**

….

From *A Book On C* by A. Kelley and I. Pohl.

FILE I/O THROUGH ARGUMENTS IN main

```
#include <stdio.h>
main(int argc, char **argv)
{
        FILE   *fin, *fout;
        int a;
        if (argc != 3)
        {
                // error message
        }
        fin = fopen(argv[1], “r”);
        fout = fopen(argv[2], “w”);
        while ((a = getc(fin)) != EOF)
        …. // processing and later closing files
}
```

can then run executable file as follows:
exe_file file1 file2 *which is equivalent to argv[0] argv[1] argv[2]*

LINKED LISTS INTRODUCTION

```
#define NULL 0
typedef int data;
typedef struct linked_list
{
    data  d;
    struct linked_list *next;
} node;
typedef  node  *link;
link head ; // head is of type link or node *

head = malloc(sizeof(node));//returns void *
head -> d = 1; // (*head).d = 1
head -> next = malloc(sizeof(node));
head -> next -> d = 2; // (*(*head).next).d = 2
head -> next -> next = malloc(sizeof(node));
head -> next -> next -> d = 3;
head -> next -> next -> next = NULL;
// (*(*(*head).next).next).next = NULL
```

LINKED LISTS HEADER FILE

```
#define NULL 0 // represents the null pointer
typedef int data;
typedef struct linked_list
{
    data  d;
    struct linked_list *next;
} node;
typedef  node  *link;
link head; // head is of type link or node *
```

Assume all of the above is saved in header file list.h

LINKED LISTS: FROM AN ARRAY TO A LIST

What does the following function do?

```
#include “list.h”
#include <stdlib.h> // malloc is found here
link r_array2list(int a[], int size)
{
    link head;

    if (size != 0)
    {
        head = malloc(sizeof(node));
        head -> d = a[0];
        head -> next = r_array2list(&a[1],
                                    size - 1);
        return head;
     }
     else
        return NULL;
}
```

LINKED LISTS FUNCTIONS

Write functions that will perform the action:

1. *iteratively convert an array of numbers to a list?*
   link it_arraytolist(int a[], int size)
2. *print a list recursively*
   void r_print(link head)
3. *print a list iteratively*
   void it_print(link head)
4. *count a list recursively*
   int r_count(link head)
5. *count a list iteratively*
   int it_count(link head)
6. *concatenate list* a *and* b
   void concatenate(link a, link b)
7. *insert an element into a list*
   void insert(link p1, link p2, link q)
9. *delete an element from a list*
   void delete(link head)

LINKED LISTS: PRINTING A LIST ITERATIVELY

```
void it_print(link head)
{
      while (head != NULL)
      {
           cout << " --> " << head -> d;
           head = head -> next;
       }
       cout << "NULL";
}
```

DIAGNOSTIC TEST

1. What is a function in computer programming?
- **A. A sub-program that can receive many parameters.**
- **B. A sub-program that can receive one parameter.**
- **C. A sub-program that can return one value.**
- **D. A sub-program that can return many values.**
- **E. A and C.**
- **F. B and D.**
- **G. Don't know.**

2. What are the difference between passing parameters to a function by-value and by-reference .
- **A. There are no differences.**
- **B. By-value passes a copy of the original, by-reference passes the way to get to the original.**
- **C. By-reference passes a copy of the original, by-value the way to get to the original.**
- **D. All of the above.**
- **E. Don't know.**

3. What are variable attributes?
- **A. The variable name.**
- **B. The duration of the variable's storage.**
- **C. The scope of visibility.**
- **D. All of the above.**

DIAGNOSTIC TEST

4. What is a variable?

A. The name of a location in memory.

B. The address of a location in memory.

C. The value of a location in memory.

D. Don't know.

5. What is a record?

A. Related structures of the same type grouped together under the same name.

B. Related structures of different types grouped together under the same name.

C. Unrelated structures of the same type grouped together under the same name.

D. Unrelated structures of different types grouped together under the same name.

E. All of the above.

F. A and B. **G. C and D.**

6. Can we build linked lists, queues, trees, and stacks using arrays?

A. Yes. **B. No.**

C. Don't know.

DIAGNOSTIC TEST

7. Can we pass a function as a parameter to another function?

A. Yes. **B. No.**

C. Depends on the language. **D. Don't know.**

8. What is the difference between recursion and iteration?

A. In recursion a program calls itself.

B. In recursion a sub-program calls itself.

C. Iteration implies interacting with the user.

D. Iteration implies use of iterative structures.

E. B and D. **F. A and E.**

9. Is it a good idea to always pass structures by-value?

A. Yes. **B. No.**

C. Don't know.

10. What is virtual memory?

A. An old computer concept with no application.

B. Allows a computer to store programs that are larger than physical memory.

C. Allows a computer to run programs that are larger than physical memory.

D. Don't know.

DIAGNOSTIC TEST

11. Where are the programs executed?

A. In memory. **B. In the disc.**

C. In the CPU. **D. In the ALU.**

12. What is a static variable?

A. A variable that exists from the moment a program begins execution.

B. Very similar to a constant since its value does not change.

C. A local variable.

D. Can be used by other programs because it is not hidden.

E. Don't know.

13. What is a external variable?

A. A variable that exists from the moment a program begins execution.

B. Very similar to a constant since its value does not change.

C. A local variable.

D. Can be used by other programs because it is not hidden.

E. Don't know.