Review of C Programming Language

Structure of a C Program

```
/* File: powertab.c
 * ----------------
 * This program generates a table comparing values
 * of the functions n^2 and 2^n.
 */
 #include <stdio.h>

/* Constants
 * ---------
 * LowerLimit - Starting value for the table
 * UpperLimit - Final value for the table
 */
#define LowerLimit 0
#define UpperLimit 12

/* Function prototypes */
int RaiseIntPower(int n, int k);
```

```
/* Main program */
int main()
{
   int n;

   printf("    |   2 |    N\n");
   printf("  N |  N  |   2\n");
   printf("----+-----+------\n");

   for (n = LowerLimit; n <= UpperLimit; n++){
      printf(" %2d | %3d | %4d\n", n,
             RaiseIntPower(n, 2),
             RaiseIntPower(2, n) );
   }
}
```

```
/*
 * Function: RaiseIntPower
 * This function returns n to the kth power.
 */

int RaiseIntPower(int n, int k)
{
   int i, result;

   result = 1;

   for (i = 0; i < k; i++){
      result *= n;
   }
   return (result);
}
```

Variables, values and types

- A ***variable*** can be thought of as a named box, or cell, in which one or more data values are stored and may be changed by the algorithm.
- An ***identifier*** is simply the algorithmic terminology for a name that we make-up to "identify" the variable.
  - **Every** variable must be given a ***unique identifier.***
- Rules for Variable Identifiers (in C)
  - A sequence of letters, digits, and the special character _.
  - A letter or underscore must be the 1st character of an identifier.
  - C is case-sensitive: Apple and apple are two different identifiers.

Data Types

- A data type is defined by two properties:
  - a *domain*, which is a set of values that belong to that type
  - a *set of operations*, which defines the behavior of that type
- Fundamental types (***atomic types***) in C:
  - can be grouped into 3 categories: integer, floating point and character.

Atomic Types

- **Integer Types:**
  - **short**: 2 bytes
  - **int**: 4 bytes
  - **long:** 4 bytes
  - **unsigned:** 4 bytes
- **Floating-point Types:**
  - **float**: 4 bytes
  - **double**: 8 bytes
  - **long double:** 8 bytes
  - **signed/unsigned**
- **Characters:**
  - **char:** 1 byte

The range of values for each type depends on the particular computer's hardware.

Examples

```
char grade;
char first, mid, last;

int age, year;
double tax_rate;

grade = ‘A’;
age = 20;
mid = ‘\0’;
```

Assignment Operator

variable = *expression*

- The expression can simply be a constant or a variable:

```
int x, y;
x = 5;
y = x;
```

- The expression can be an arithmetic expression:

```
x = y + 1;
y = y * 2;
```

- Embedded assignments:

```
z = (x = 6) + ( y = 7);
n1 = n2 = n3 = 0;
```

- Shorthand assignments:

```
x += y;
z -= x;
y /= 10;
```

Boolean Operators

C defines three classes of operators that manipulate Boolean data:

1. **relational operators**
   - Greater than `>` Greater than or equal `>=`
   - Less than `<` Less than or equal `<=`
   - Equal to `==` Not equal to `!=`
2. **logical operators**
   - **AND : &&** (TRUE if both operands are TRUE)
     - ((5 > 4) && (4 > 7)) is FALSE
   - **OR : ||** (TRUE if either or both operands are TRUE)
     - ((5 >4) || (4 > 7)) is TRUE
   - **NOT: !** (TRUE if the following operand is FALSE)
     - ! (4 > 7) is TRUE
3. **?: operator**
   - (*condition*) ? *expr1* : *expr2*;
     - max = (x>y) ? x : y;

Input and Output

- ***scanf:*** obtains (reads) an input value
  - e.g. `scanf("%d", &num);`
- ***printf:*** sends out an output value
  - e.g. `printf("%d", num);`
    `printf("Hello world!");`
- **General format:**

  scanf(*format control string*, *input var list*);

  printf(*format control string*, *output var list*);

  - number of conversion characters = number of arguments.

Conversion characters

character	type
%c	char
%d	int
%f	float, double (for printf)
%lf	double (for scanf)
%Lf	long double (for scanf)

Statements

- **Simple Statement**

  expression ;

  – The expression can be a function call, an assignment, or a variable followed by the ++ or -- operator.

- **Blocks**

  A block is a collection of statements enclosed in curly braces:

```
{
    statement_1
    statement_ 2
    ...
    statement_n
}
```

The `if` statement

It comes in two forms:

if (*condition*)
    statement

if (*condition*)
    statement
else
    statement

Example:

```
if (n%2) == 0)
   printf("That number is even.\n");
else
   printf("That number is odd.\n");
```

Nested `if` Statements

if (*condition1*)

  `if` (*condition 2*)

    statement 1

  `else`

    statement 2

`else`

  statement 3

The else-if Statement

if *(condition1)*

statement

`else if` *(condition2)*

statement

`else if` *(condition3)*

statement

`else`

statement

If Statement Exercise

- Write a C program that reads in
  - the dimensions of a room (length & width) and
  - the size of a desk (length & width)
- and determines if the desk can fit in the room with each of it's sides parallel to a wall in the room.
- Assume the user enters positive numbers for each of the four dimensions.

```
#include <stdio.h>
int main() {

    /* declare necessary variables */
    int roomlen, roomwid;
    int desklen, deskwid;

    /* read input */
    printf("Enter the length and width of the room.\n");
    scanf("%d%d",&roomlen, &roomwid);
    printf("Enter the length and width of the desk.\n");
    scanf("%d%d",&desklen, &deskwid);

    /* decide if the table fits in the room */
    if ((deskwid <= roomwid) && (desklen <= roomlen))
       print("The desk will fit in the room.\n");
    else
       print("The desk will not fit in the room.\n");

    return 0;
}
```

- **Will this work in every situation? Why or why not?**

```
#include <stdio.h>
int main()
{
  int roomlen, roomwid;
  int desklen, deskwid;
  int temp;

  printf("Enter the length and width of the room.\n");
  scanf("%d%d",&roomlen, &roomwid);
  printf("Enter the length and width of the desk.\n");
  scanf("%d%d",&desklen, &deskwid);

  // Check both ways of putting the desk in the room
  if ((deskwid <= roomwid) && (desklen <= roomlen))
     printf("The desk will fit in the room.\n");
  else if ((deskwid<=roomlen) && (desklen<=roomwid))
     printf("The desk will fit in the room.\n");
  else
     printf("The desk will not fit in the room.\n");

  return 0;
}
```

The `switch` statement

General Form:

```
switch (e) {
   case c1 :
      statements
      break;
   case c2 :
      statements
      break;

         more case clauses

   default:
      statements
      break;
}
```

Example:

```
int MonthDays (int month, int year)
{
   switch (month){
      case 9:
      case 4:
      case 6:
      case 11: return 30;
      case 2:
         return (IsLeapYear(year))? 29: 28);
      default :
         return 31;
   }
}
```

Iterative Statements

The while Statement

General form:

while (*conditional expression***)**

statement

Example 1

```
/* What does this program do? */
int DS (int n)
{
   int sum = 0;

   while (n > 0) {
      sum += n % 10;
      n /= 10;
   }

   return (sum);
}
```

Example 2

```
/* This program add a list of numbers */
#define sentinel 0
main()
{  int value, total =0;
   printf("This program add a list of numbers.\n");
   printf("Use %d to signal the end of list.\n",
              sentinel);
   while (TRUE) {
      printf(" ? ");
      value = GetInteger();
      if (value == sentinel) break;
      total += value;
   }
   printf("The total is %d.\n", total);
}
```

Example 3: Menu driven program set-up

```
int main() {
  int choice;
  while (TRUE) {
     Print out the menu.
     scanf("%d",&choice);
     if (choice == 1)
          Execute this option
     else if (choice == 2)
          Execute this option
           ...
     else if (choice == quitting choice)
          break;
      else
          That's not a valid menu choice!
  }
  return 0;
}
```

The `for` Statement

General form:

for (*initialization*; *loopContinuationTest*; *increment*)
    statement

which is equivalent to the `while` statement:

initialization;
while (*loopContinuationTest*) **{**
    statement
    increment;
}

Example

- Finding the sum 1+3+…+99:

```
int main() {

   int val ;
   int sum = 0;

   for (val = 1; val < 100; val = val+2) {
        sum = sum + val;
   }

   printf("1+3+5+...+99=%d\n",sum);
   return 0;
}
```

The do/while Statement

General Form:

do {
statements
} while (*condition* **);**

- **Example**: a loop to enforce the user to enter an acceptable answer of Yes or No.

```
do {
  printf("Do you want to continue? (Y/N)");
  scanf("%c",&ans);
} while (ans!='Y' && ans!='y' && ans!='N' &&
         ans!='n');
```

Nested Control Structures

Example :

```
value = 0;
for (i=1; i<=10; i=i+1)
   for (j=1; j<=i; j=j+1)
        value = value + 1;
```

- How many times the inner loop is executed?

Functions

Functions can be categorized by their return types:

- Function returning a value – A function that performs some subtask that requires returning some value to the calling function.

- Void function – A function that performs some subtask that does not require a single value to be returned to the calling function.

Returning results from functions

- Functions can return values of any type.
- Example:

```
int IsLeapYear(int year)
{
   return ( ((year % 4 == 0) && (year % 100 != 0))
            || (year % 400 == 0) );
}
```

- This function may be called as:

```
if (IsLeapYear(2003))
   printf("29 days in February.\n");
else
   printf("28 days in February.\n");
```

`void` Functions

- No need to include a `return` statement in a void function.
- A common example of a void function is one that prints out a menu:

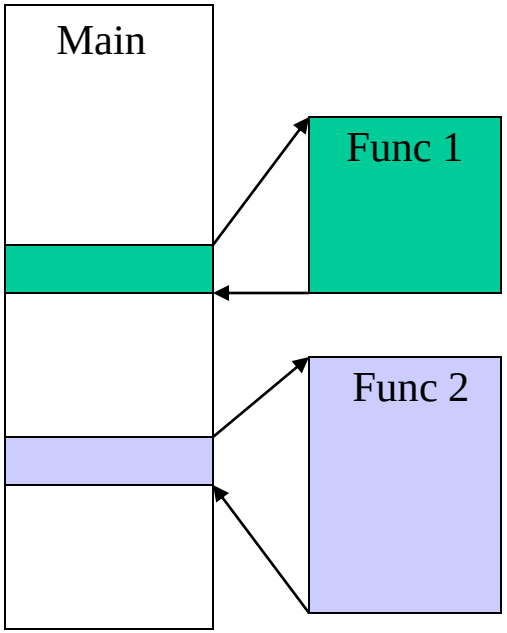
```
void menu() {
  printf("Please choose one of the following.\n");
  printf(“ Option 1\n”);
  printf(“ Option 2\n”);
  printf(“ Option 3\n”);
  printf(“ Quit\n”);
}
```

You can call this function as follows:

```
menu();
```

Function Invocation

- A program is made up of one or more functions, one of them being `main( )`.
- When a program encounters a function, the function is called or invoked.
- After the function does its work, program control is passed back to the calling environment, where program execution continues.



Value and Reference Parameters

- When an argument is passed *call-by-value*, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller.
- When an argument passed *call-by reference*, the caller actually allows the called function to modify the original variable's value.
- In C, all calls are call-by-value. However it is possible to simulate call by reference by using pointers. (Address operators (&) and indirection operators (*))

Call by value

- Let's define a function to compute the cube of a number:

```
int cube ( int num ) {
    int result;

    result = num * num * num;
    return result;
}
```

formal parameter

- This function can be called as:

```
n = cube(5);
```

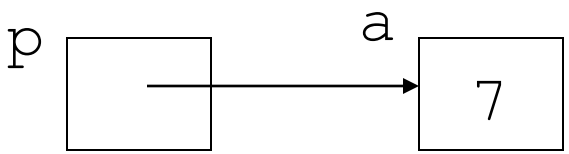
Actual parameter

Pointers

- A pointer is simply the internal machine address of a value inside the computer's memory.

```
int a;
int *p;

a = 7;
p = &a;
```

p is a pointer to int.



- We can reach the contents of the memory cell addressed by p:

```
printf("%d\n", *p);
```

Fundamental pointer operations

1. Assigning the address of a declared variable:

```
int a, *this, *that;
this = &a;
```

2. Assigning a value to a variable to which a pointer points.

```
*this = 4;
```

3. Making one pointer variable refer to another:

```
that = this;
```

4. Creating a new variable.

```
this = malloc(sizeof(int));
```

Addressing and Dereferencing

```
int a, b;
int *p, *q;

a=42; b=163;
p = &a;
q = &b;
printf("*p = %d, *q = %d\n", *p, *q);

*p = 17;
printf("a = %d\n", a);

p = q;
*p = 2 * *p - a;
printf("b = %d\n", b);
```

```
p = &a;
printf("Enter an integer: ");
scanf("%d", p);
*q = *p;

----------------------------------------

double x, y, *p;

p= &x;
y = *p;

equivalent to

y = *&x;
or
y = x;
```

Call by reference

```
void SetToZero (int var)
{
    var = 0;
}
```

- You would make the following call:

```
SetToZero(x);
```

- This function has no effect whatever. Instead, pass a pointer:

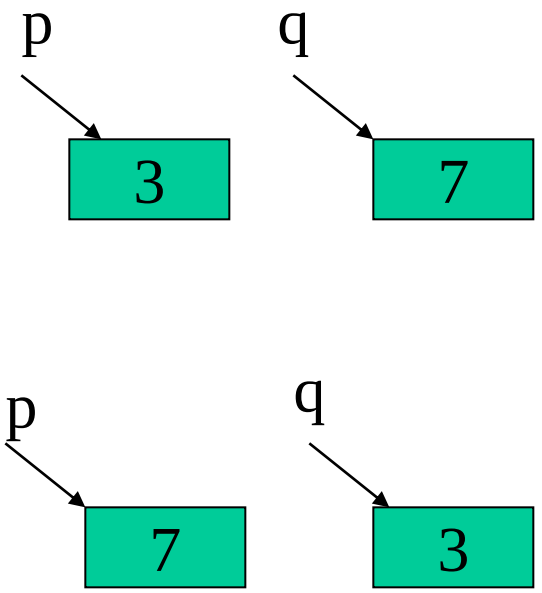
```
void SetToZero (int *ip)
{
   *ip = 0;
}
```

- You would make the following call:

```
SetToZero(&x);
```

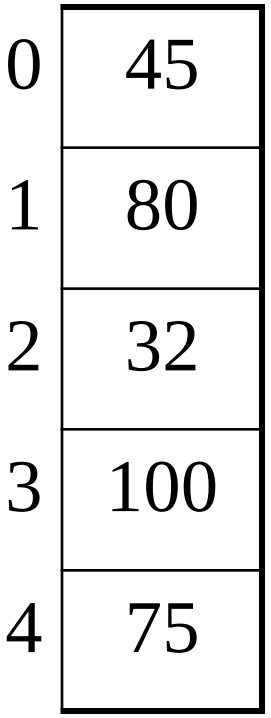
```
/* Swapping arguments (incorrect version) */
#include <stdio.h>

void swap (int p, int q);
int main (void)
{
      int a = 3;
      int b = 7;
      printf("%d  %d\n", a,b);
      swap(a,b);
      printf("%d  %d\n", a, b);
      return 0;
}

void swap (int p, int q)
{
      int tmp;

      tmp = p;
      p = q;
      q = tmp;
}
```

```
/* Swapping arguments (correct
version) */
#include <stdio.h>

void swap (int *p, int *q);
int main (void)
{
      int a = 3;
      int b = 7;
      printf("%d  %d\n", a,b);
      swap(&a, &b);
      printf("%d  %d\n", a, b);
      return 0;
}

void swap (int *p, int *q)
{
      int tmp;

      tmp = *p;
      *p = *q;
      *q = tmp;
}
```



Arrays

```
int grade[5];

grade[0] = 45;
grade[1] = 80;
grade[2] = 32;
grade[3] = 100;
grade[4] = 75;
```



- Indexing of array elements starts at 0.

Examples

- Reading values into an array

```
int i, x[100];
```

```
for (i=0; i < 100; i=i+1) {
   printf("Enter an integer: ");
   scanf("%d",&x[i]);
}
```

- Summing up all elements in an array

```
int sum = 0;
for (i=0; i<=99; i=i+1)
   sum = sum + x[i];
```

Examples (contd.)

- Shifting the elements of an array to the left.

```
/* store the value of the first element in a
 * temporary variable
 */
temp = x[0];

for (i=0; i < 99; i=i+1)
   x[i] = x[i+1];

//The value stored in temp is going to be
the value of the last element:
x[99] = temp;
```

Examples

- Finding the location of a given value (`item`) in an array.

```
i = 0;
while ((i<100) && (x[i] != item))
   i = i + 1;

if (i == 100)
   loc = -1; // not found
else
   loc = i;  // found in location i
```

Passing arrays as parameters

- In a function definition, a formal parameter that is declared as an array is actually a pointer.
- When an array is being passed, its base address is passed call-by-value. The array elements themselves are not copied.

Example

```
int Mean(double a[], int n)
{
   int j;
   double sum = 0;

   for (j=0; j < n ; j++)
      sum = sum + a[j];

   return (sum/n);
}
```

Note: a[] is a notational convenience. In fact

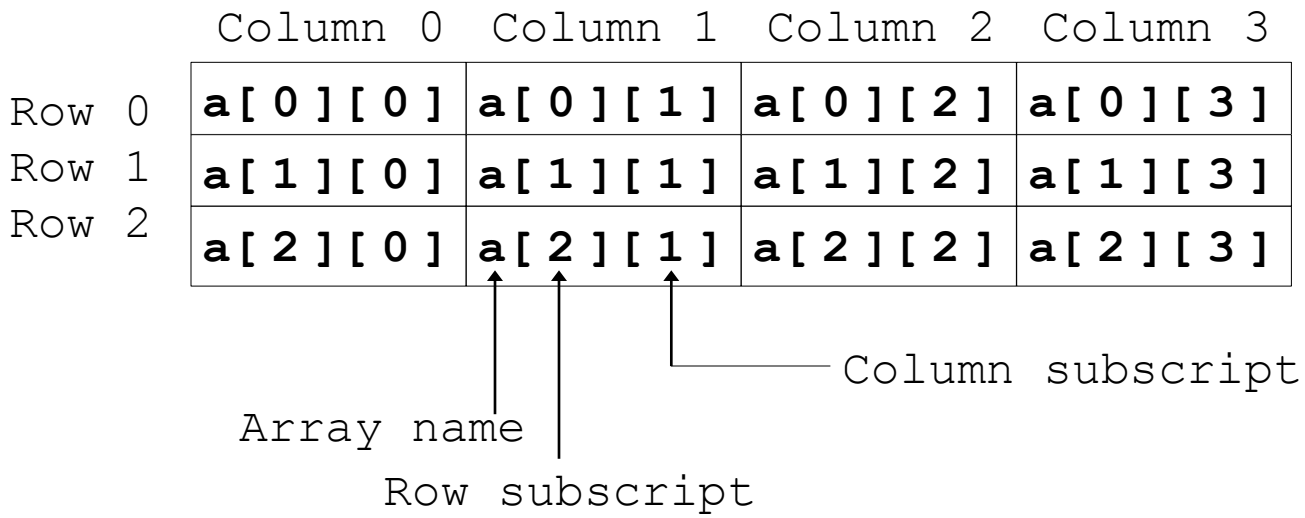
int a[] ≡ int *a

- Calling the function:

```
int total, x[100];

total = Mean(x, 100);
total = Mean(x, 88);
total = Mean(&x[5], 50);
```

Two Dimensional Arrays

- Multiple subscripted arrays
  - Tables with rows and columns (**m** by **n** array)
  - Like matrices: specify row, then column



Two Dimensional Arrays

- Initialization
  - `int b[2][2]={{1,2},{3,4}};`

1	2
3	4

  - Initializers grouped by row in braces
  - If not enough, unspecified elements set to zero

```
int b[2][2]={{1},{3,4}};
```

1	0
3	4

- Referencing elements
  - Specify row, then column

```
printf("%d",b[0][1]);
```

Example

- Reading values into a two-dimensional array:

```
int a[10][20];
for (row=0; row < 10; row = row+1){
   for(col=0; col < 20; col = col+1) {
      printf(“Enter a number: ");
      scanf("%d",&a[row][col]);
   }
}
```

Pointers and Arrays

- Arrays are implemented as pointers.
- The operations on pointers make sense if you consider them in relation to an array.
- Consider:

```
double list[3];
double *p;
```

  - `&list[1]` : is the address of the second element
  - `&list[i]` : the address of list[i] which is calculated by the formula

base address of the array + `i` * *8*

Pointer Arithmetic

- If we have :

```
p = &list[0];      // Or: p = list;
```

then

`p + k` *is defined to be* `&list[k]`

- If

```
p = &list[1];
```

then

`p - 1` *corresponds to* `&list[0]`

`p + 1` *corresponds to* `&list[2]`

Pointer Arithmetic (cont.)

- The arithmetic operations *, /, and % make no sense for pointers and cannot be used with pointer operands.
- The uses of + and – with pointers are limited. In C, you can add or subtract an integer from a pointer, but you cannot, for example add two pointers.
- The only other arithmetic operation defined for pointers is subtracting one pointer from another. The expression `p1 – p2` where both `p1` and `p2` are pointers, is defined to return the number of array elements between the current values of `p2` and `p1`.
- Incrementing and decrementing operators:

  `*p++` is equivalent to `*(p++)`

Example

- Illustrates the relationship between pointers and arrays.

```
int SumIntegerArray(int *ip, int n)
{
   int i, sum;

   sum = 0;
   for (i=0; i < n; i++) {
      sum += *ip++;
   }
   return sum;
}
```

- Assume

```
int sum, list[5];
```

- Function call:

```
sum = SumIntegerArray(list, 5);
```

`struct` Construct

- A **structure** is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
- **Example**:

```
struct party{
   int house_number;
   int time_starts;
   int time_ends;
};
struct party party1, party2;
```

- Rather than a collection of 6 variables, we have:
  - 2 variables with 3 fields each.
  - both are identical in structure.

Structure Definitions

- **Example:**

```
struct point {
    int x;
    int y;
};

struct point pt; /* defines a variable pt which
                    is a structure of type
                    struct point */

pt.x = 15;
pt.y = 30;
printf("%d, %d", pt.x, pt.y);
```

Structure Definitions

- Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners.

```
struct rect {
     struct point pt1;
     struct point pt2;
};

struct rect screen;

/* Print the pt1 field of screen */
printf("%d, %d",screen.pt1.x,screen.pt1.y);

/* Print the pt2 field of screen */
printf("%d, %d",screen.pt2.x,screen.pt2.y);
```

`typedef`

- **`typedef`**
  - Creates synonyms (aliases) for previously defined data types
  - Use **`typedef`** to create shorter type names
  - Example:

```
typedef struct point pixel;
```

  - Defines a new type name **`pixel`** as a synonym for type **`struct point`**
  - **`typedef`** does not create a new data type
    - Only creates an alias

Structures and Pointers

```
struct point *p;  /* p is a pointer to a structure
                     of type struct point */
struct point origin;

p = &origin;
printf("Origin is (%d, %d)\n", (*p).x, (*p).y);
```

- Parenthesis are necessary in `(*p).x` because the precedence of the structure member operator (dot) is higher than *.
- The expression `*p.x` ≡ `*(p.x)` which is illegal because `x` is not a pointer.

Structures and Pointers

- Pointers to structures are so frequently used that an alternative is provided as a shorthand.
- If `p` is a pointer to a structure, then

  `p -> field_of_structure`

  refers to a particular field.
- We could write

```
printf("Origin is (%d %d)\n", p->x, p->y);
```

Structures and Pointers

- Both `.` and `->` associate from left to right
- Consider

```
struct rect r, *rp = &r;
```

- The following 4 expressions are equivalent.

```
r.pt1.x
rp -> pt1.x
(r.pt1).x
(rp->pt1).x
```

Declarations and Assignments

```
struct student {
     char *last_name;
     int student_id;
     char grade;
};
struct student temp, *p = &temp;

temp.grade = 'A';
temp.last_name = "Casanova";
temp.student_id = 590017;
```

Expression	Equiv. Expression	Value
temp.grade	p -> grade	A
temp.last_name	p -> last_name	Casanova
temp.student_id	p -> student_id	590017
(*p).student_id	p -> student_id	590017

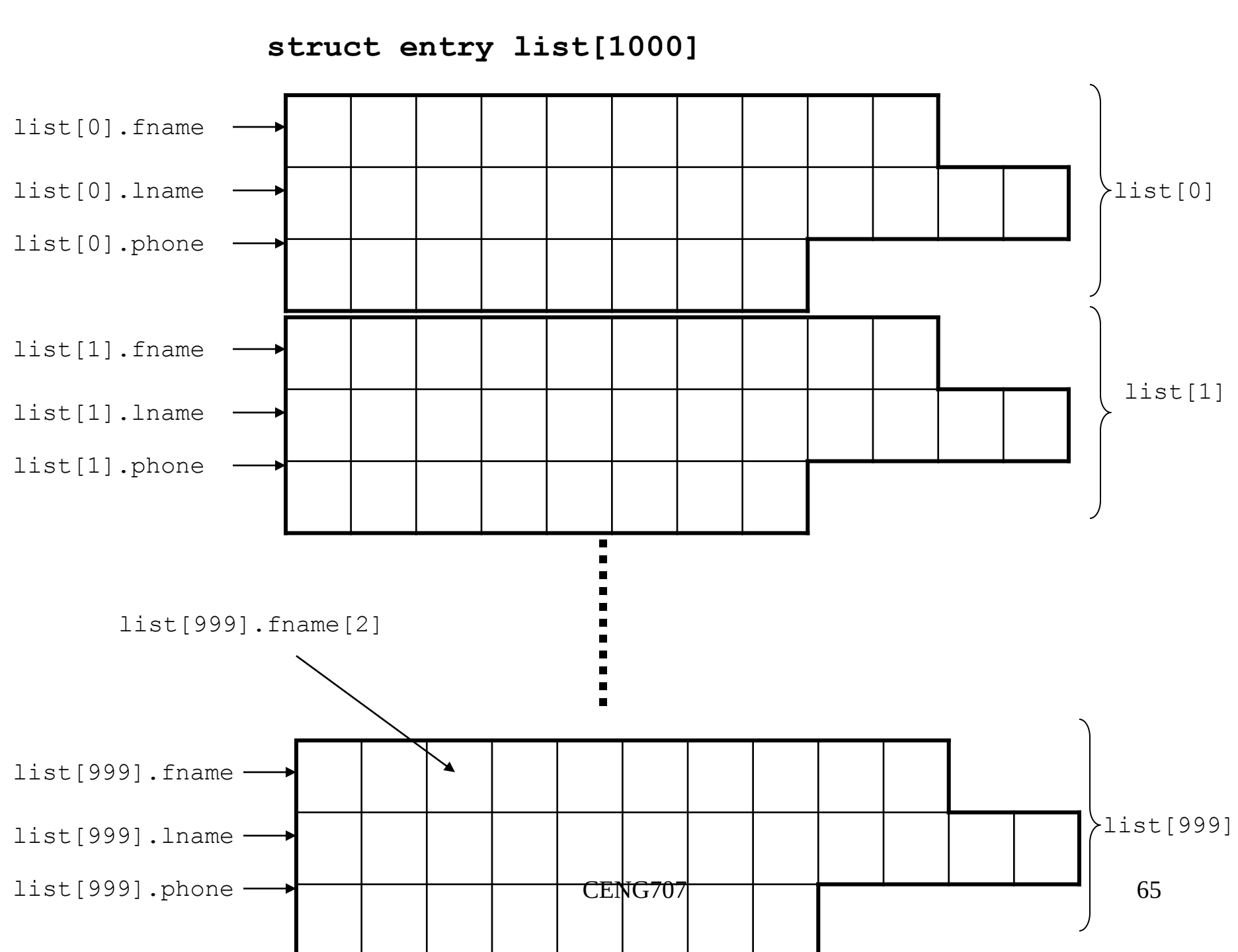
Arrays of Structures

- Usually a program needs to work with more than one instance of data.
- For example, to maintain a list of phone #s in a program, you can define a structure to hold each person's name and number.

```
struct entry {
        char fname[10];
        char lname[12];
        char phone[8];
};
```

- And an array of entries:

```
struct entry list[1000];
```


struct entry list[1000]
list[0].fname
list[0].lname
list[0].phone
list[0]
list[1].fname
list[1].lname
list[1].phone
list[1]
list[999].fname[2]
list[999].fname
list[999].lname
list[999].phone
list[999]

Using Structures With Functions

```
/* Demonstrates passing a structure to a function */
#include<stdio.h>

struct data{
    float amount;
    char fname[30];
    char lname[30];
}rec;

void printRecord(struct data x);

int main(void)
{     printf("Enter the donor's first and last names\n");
      printf("separated by a space:  ");
      scanf("%s %s",rec.fname, rec.lname);
      printf("Enter the donation amount:   ");
      scanf("%lf",&rec.amount);
      printRecord(rec);
      return 0;
}
```

```
void printRecord(struct data x)
{
     printf("\nDonor %s %s gave $%.2f.",
          x.fname, x.lname, x.amount);
}
```

```
/* Make a point from x and y components. */
struct point makepoint (int x, int y)
{
        struct point temp;

        temp.x = x;
        temp.y = y;
        return (temp);

}
/* makepoint can now be used to initialize a
   structure */
struct rect screen;
struct point middle;

screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(50,100);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

```
/* add two points */

struct point addpoint (struct point p1, struct point p2)
{
     p1.x += p2.x;
     p1.y += p2.y;
     return p1;
}
```

Both arguments and the return value are structures in the function `addpoint`.