CSC 181F Lecture Notes

These lecture notes are provided for the personal use of students taking CSC181F in the Fall term 1999/2000 at the University of Toronto

Reading Assignment

```
K.N. King         Chapter  11,  12

K.N. King         Section 17.7
```


Pointers in C

- During the execution of a program, all variables in the program are stored in the memory of the computer.
- The location of a variable in memory is called the *address of the variable.* Thus every variable has *two* attributes.

  Value   The contents of the variable

  Address   The location of the variable in memory
- In C a *pointer* is a special kind of variable whose value if the *address of* other variables.
- In C almost all pointers hold the addresses of variables of one specific type.

Pointer Declarations

type-name ***pointerVar***

- *type-name* is the base type for the pointer. Use **void** to specify a generic base type.

  ```
  int * P ;      /* P is a pointer to integer variables */
  void * Q ;     /* Q is a pointer with no type */
  ```
- Use the *address-of* operator (&) to create addresses of any ordinary variable.
- The *pointer dereferencing operator* (*) is used to access the variable that a pointer variable is pointing at.
- **WARNING:** C does **NO** automatic run-time checking for proper pointer usage

  WARNING: it is an **ERROR** to apply the * operator to an uninitialized pointer variable.
- The ++ and -- operators change a pointer variable by size of object that the pointer points at. This feature should **only** be used to access consecutive elements of arrays.

Pointer Example

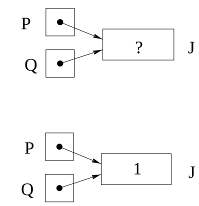
```
int J , K = 12 , * intPtr ;
intPtr = & K ;              /* Set intPtr to address of K */
J = * intPtr ;              /* Do J = K the hard way */
* intPtr = 17 ;             /* Do K = 17 using intPtr */
```

Variable	Value	Address
J	12	0xFFAD1284
K	17	0xFFAD1288
intPtr	0xFFAD1288	0xFFAD128C

More Pointer Examples

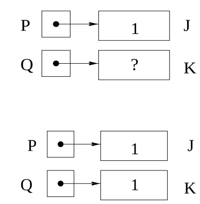
```
int J , K , *P , *Q ;
P = & J ;
Q = P ;

*P = 1 ;
```



```
P = & J ;
Q = & K ;
J = 1 ;

* Q = * P;
```



Note the difference between `Q = P` and `*Q = *P`.

Pointer Variable Values

- A pointer variable can have one of four values
  1. Uninitialized. The variable has never been given a value.
  2. NULL. This is a distinguished value that is by convention used to indicate a pointer that doesn't point at any object.
     For convenience in building loops, NULL usually has the same value as **false** .
  3. Compatible address. The pointer variable points at an object of its declared type
  4. Incompatible address. The pointer points at some object (or at some random location in memory) that is not compatible with its declared type.
- It is an ERROR in C to
  1. Apply the pointer dereferencing operator (`*`)to an uninitialized pointer.
  2. Apply the pointer dereferencing operator (`*`)to a pointer that has the value NULL

- It is **extremely poor programming practice** to do anything with a pointer that has an incompatible address. Such usage is *non-portable* and inherently error prone.
- Pointer variables of type **void *** are used to store pointers that do have a declared type.
- There are *NO* operations defined for the data type **void** , so directly dereferencing a **void** pointer is an ERROR.
  Any use of a **void** pointer *will* require a *type cast* to make it legal.

Example:

```
int J = 23, K, * IP ;
void * VP ;
IP = & J ;
VP = IP ;
K = * VP ;            /* ILLEGAL */
K = ( int ) * VP ;
IP = ( int * ) VP ;
```

Arrays and Pointers

- In C it ia assumed that if A is an array and J is less than K, the address of A[J] is less than the address A[K].

```
int A[1000] , sum = 0 , * P ;
for ( P = & A[0] ; P < & A[ 1000 ] ; P++ )
    sum += * P ;
```

- The * and ++ operator can be combined in statements that process array elements.

```
int A[ 100 ], * P , J , K , sum ;
P = & A[ K ];
```

The statement * P++ = J (or * (P++) = J) assigns J to A[K] and sets P to points at A[K + 1].

Example:

```
P = & A[0] ;
while ( P < & A[ 100 ])
    sum += * P ++ ;
```


Pointer Arithmetic and Arrays

C supports the following *pointer arithmetic*

- Adding/subtracting an integer K to/from a pointer P yields a pointer to the element that is K places after/before the one that P points to.

```
int A[10], * P , * Q , N ;
P = & A[2];
Q = P + 3;     /* Q points to A[5] */
P += 6;        /*P points to A[8] */
```

- Subtracting pointers yields the distance (measured in array element) between the pointers.

```
P = & A[5];
Q = & A[1];
N = P - Q ;    /* N is 4 */
N = Q - P;     /* N is -4 */
```


- The name of an array is the same as a pointer to the first element of the array. It is a *pointer constant*.

```
int A[10] ;
* A = 7 ;          /* stores 7 in A[0] */
* (A+3) = 12 ;     /* stores 12 in A[3] */
A++ ;              /* ERROR, can't modify pointer constant */
```

- A pointer to an array can be denoted by enclosing * *pointerVar* in parentheses.

```
int (* P)[10];     /* P is a pointer to array of length 10 */
int * Q[20];       /* Q is an array of 20 pointers to int */
```

- **WARNING:** C does not check addesses computed using pointer arithmetic. **Good Style:** NEVER use pointer arithmetic to address outside of an array.
- There is an equivalence between array subscripting and pointer dereferencing.

  A[K] is the same as * (A + K)   or   * (& A[0] + K)

HOW TO Use Pointers in Array Loops

- It is often more efficient or more convenient to use a pointer rather than an index to process an array. If the type of the array element is *type-name* , use a pointer variable with type *type-name* *
- For an array A, the address of the first element in the array is & A[0]
  If ASIZE is the size of the array, the address of the last element in the array is & A [ASIZE - 1]
  The address of the first element that is **not** in the array is & A[ASIZE]

```
double X[ 400 ] , Y[ 400 ] ;
int K ;
...
for ( I = 0 ; I < 400 ; I++ )
    X[ K ] = Y[ K ] ;
```

```
double X[ 400 ] , Y[ 400 ];
double * XP , * YP ;
...
for ( XP = & X[ 0 ] , YP = & Y[ 0 ] ; XP < & X[ 400 ] ;  )
    * XP++ = * YP++ ;
```

- For *any* one dimensional array *type-name* A [ASIZE] **;**
  - *type-name* ***** P is a pointer to array elements
  - & A [0] is the address of the first element
  - & A[ASIZE - 1] is the address of the last element
  - & A[ASIZE] is an address *just beyond* the end of the array
- To loop through an entire array, set the pointer to start at the first or last element of the array P = & A[0] **;** or P = & A[ASIZE - 1] **;**
- Move through the array using the ++ or - - operators
  Note: that these operators change P in units of **one array element**
- The end of loop condition is
  P < & A [ASIZE] or P <= A[ASIZE - 1] (up counting)
  P >= & A [0] (down counting)
- Loop templates
  for (P = & A[0] **;** P < & A [ASIZE] **;** P ++) **...**
  for (P = & A[ASIZE - 1] **;** P >= & A[0] **;** P - -) **...**

Multidimensional Arrays and Pointers

- For any two-dimensional array A, the expression A[K] is a pointer to the first element in row K of the array.
- The name of two-dimensional array is a pointer to pointer.
  int A[10][10]; /* A has type **int * *** (pointer to pointer to int) */

Examples:

int A[NUM_ROWS][NUM_COLS] **,** *P **,** K **;**
/* Clears row K of the array A */
for (P = A[K] **;** P < A[K] + NUM_COLS **;** P++)
  ***** P = 0 **;**
int A[NUM_ROWS][NUM_COLS] **,** K **,** (*P)[NUM_COLS] **;**
/* Clears column K of the array A */
for (P = A **;** P <= & A[NUM_ROWS - 1] **;** P++)
  (***** P)[K] = 0 **;**

Pointers, Parameters and Arguments

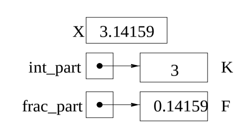
- A function parameter that is declared as a pointer allows the function to directly access variables in the calling program.
- This mechanism is used for several purposes
  - As an alternative way to specify array parameters.
    type-name A[] is the same as *type-name* ***** A
  - To avoid passing large objects (e.g. structures and unions) by value.
  - To allow a function to return more than one value.
  - To allow a function to process linked data structures like trees and lists.
- The function parameter is declared as a pointer to the appropriate type.
  The corresponding argument **must be**
  1. The address of a variable of that type created using the & operator.
  2. A pointer to an object of the same type.

WARNING: Many C compilers do not check pointer arguments carefully.

Pointer Parameter Example

```
int F( int * Z ) ;      /* function prototype */
. . .
int X, N, * P ;
P = & X ;
scanf("%d", P);         /* Note no & in front of P */
N = F( & X );           /* equivalent to N = F( P ) */
```

```
decompose( 3.14159, & K, & F );
. . .
void decompose( float X, int *int_part, float *frac_part )
{
  *int_part = (int) X;
  *frac_part = X - *int_part;
}
```



Pointer Returning Functions

- A function may be declared to return a pointer to some type of object. Declare the function as:

  type-name ***** functionName(*parameters*) **;**

- the following rules apply to the value of the pointer returned by such a function
  - The pointer should always be a pointer to an object of the correct type.
  - If the pointer isn't null then it should point to a variable outside of the function or to some storage that was newly allocated by the function.
    WARNING: a pointer pointing to the local (automatic) storage of a function points at GARBAGE after the function returns.
  - It is an ERROR to return a pointer to any of the functions local variables. Those variables *cease to exist* when the function returns.

Pointers to Functions

type-name (***** *funcPointer*)(*parameterList*) **;**

- This declaration declares a pointer variable (*funcPointer*) that is a pointer **to a function**.
  The function returns the type of value specified by *type-name* and accepts the arguments specified by *parameterList*
- The name of any declared or defined function is a *function pointer constant*. These are the only values that may be assigned to function pointer variables.'
- This mechanism is typically used for two purposes
  1. To allow functions to take a *function name* as an argument.
  2. To create function *variables* that take on the value of more than one function constant.

- The function constant assigned to a function pointer should **always** be compatible with the declaration for the function pointer variable.
- The function constant passed as an argument to a function pointer parameter should **always** be compatible with the corresponding parameter declaration.
- Compatible means
  - The type returned by the function is the same.
  - Corresponding parameters are of the same type.
  - The number of parameters is the same.
- If these rules are not followed **CHAOS** will ensue.

Function Pointer Examples

```
float bisection( float (*funcPtr)( float ), float x0, float x1 ) ;
float function1( float x ) ;
float function2( float x ) ;
. . .
y = bisection( function1, 0.0, 2.0 ) ;
. . .
y = bisection( function2, -10.0, 10.0 ) ;
. . .
float bisection( float (*funcPtr)( float ), float x0, float x1)
{
. . .
y = ( * funcPtr)(x0) ;
. . .
}
```

Reading Assignment

```
K.N. King         Chapter  16, 18

Supplementary reading

S. McConnell      Chapter 12
```


Structures and Unions

- A structure is a mechanism that allows several data items of arbitrary types to be treated as a single entity.
- Structures are typically used when some block of *logically related* informantion needs to be processed in a program.
  Examples: name, address, telephone number
  X coordinate, Y coordinate, Z coordinate
  student name, student number, assignment marks
- A union is a mechanism for saving space when several *mutually exclusive* data items need to be stored.
- **Good Style:** always use a **typedef** to create a single point of definition for any structure or union that has a significant use in a program.

Structure Declaration

```
struct structureTag {
        structureFields
    } identifierList ;
```

- The *structureTag* provides a name for the structure. This name can be used like a type name to declare variables of the structure type.
- The optional *identifierList* is a list of structure *variables* that are being declared at the same time as the structure.
- The *structureFields* are ordinary data declarations that describe data contained in the structure.

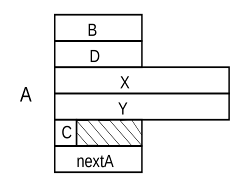
Example:

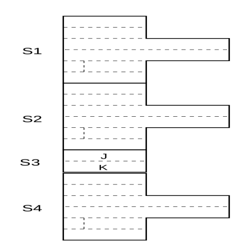
```
struct exStruct {                      typedef struct {
    char name[ 25 ] ;                      char label[ 33 ] ;
    int price ;                            float value ;
} computer1, computer2 ;               } myStruct ;
struct exStruct computer3 ;            myStruct s1, s2 ;
```


Structure Declaration Examples

```
/* Type declarations */
struct A {
    int B , C ;
    double X , Y ;
    char C ;
    struct A * nextA ;
} ;
typedef struct A myA ;

/* Variable declarations */
struct A S1 , S2 ;
struct {
    int J , K ;
    } S3 ;
myA S4 ;
```





HOW TO Use Structures

- Each structure body represents a new scope. Declare the variables that you want to treat as a unit in this scope.
- Structure variables can be initialized with declaration by giving a list of values for the structure fields.

```
struct compStruct {
    char name[ NAME_LEN + 1 ] ;
    int price;
} computer1 = {"IBM", 3499}, computer2 = {"Dell", 2265} ;
```

- The assignment operator applies to entire structures.
  The contents of the structure on the right side of the = is copied to the structure on the left side.

```
computer2 = computer1 ;
```

- Entire structures can **not** be compared.

- To access a field in a structure variable, use the *field access operator* **.**

```
computer1.price = 2199 ;
```

- If P is a pointer variable that has been declared as a pointer to some structure type S, then (assuming P points at a structure) the fields of the structure can be accessed using the *pointer operator* **->**

```
struct compStruct compPtr = & computer2 ;
compPtr -> price = 3799 ;
```

- the fields in a structure can be any C *type-name* including arrays and structures. If a structure has a structure tag, a pointer to the structure can be declared inside the structure.
- The size of a structure is **approximately** the sum of the sizes of the fields in the structure. Use **sizeof** to get the exact size of any structure.

HOW TO Use Structures

- The primary use of structures is to package several related variables together so that they can be treated as a single object.
- Structures can be used as function arguments anda function can return a structure as its value.
  WARNING: Structures are *copied* when they are passed by value or returned.
  Excessive copying of large structures can make a program inefficient.
  Consider using a pointer to the structure instead.
- Any type of object can be used as a field of a structure including another structure.

Examples:

```
struct A {
    float X , Y ;
    int K ;
} ;
typedef struct A myA ;
```

```
struct B {
    struct A BownA ;
    int K ;
    myA Array[ 10 ] ;
} ;
```


Union Types

```
union unionTag {
        fieldAlternativesList
    } identifierList ;
```

- *unionTag* is the name of the union type
- The *fieldAlternativesList* is a list of *mutually exclusive* fieldAlternatives.
  Each *fieldAlternatives* is a **single** data declaration.
  Use a structure to pack several data items into one alternative.
- The optional *identifierList* is a list of union *variables* that are being declared at the same time as the union.

Union Declaration Examples

```
/* Type declarations */
union A {
    int B , C ;
    double X , Y ;
    char C ;
    struct A * nextA ;
} ;
typedef union A myUA ;
```

```
/* Type declarations */
union typeOverlay {
    int integer ;
    float floater ;
    unsigned char bytes[ 4 ] ;
    void * pointer ;
} ;
```


HOW TO Use Unions

- Unions are a mechanism for saving space when several **mutually exclusive** alternative sets of data need to be stored and treated like a single object.
- The fieldAlternatives **overlap** in memory so only one is active at any instant in time.
- The size of a union is **approximately** the size of the largest field alternative in the union. Use **sizeof** to get the exact size of a union.
- **C does NO run-time checking for proper use of unions**
- The programmer must provide some way of indicating which field alternative is active at any instant in time

Example of a self-identifying structure/union.

```
typedef enum { point , square , circle , triangle }
    uType ;
struct uStruct {
  uType unionKind ;
  union {
    double side ;        /* square */
    int radius ;        /* circle */
    float sideA , sideB , sideC ;  /* triangle */
  } value ;
  double Xcoordinate , Ycoodinate ;
} ;
```