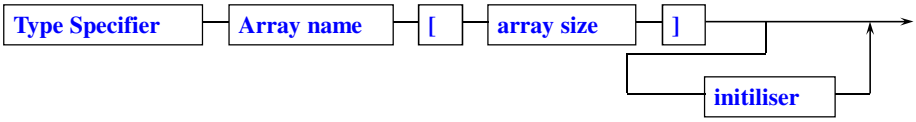
Arrays and Pointers




```
int daily_temp[365];
```

Subscripts begin at 0

More efficient code

easier to access array elements through pointers

Arrays and Pointers

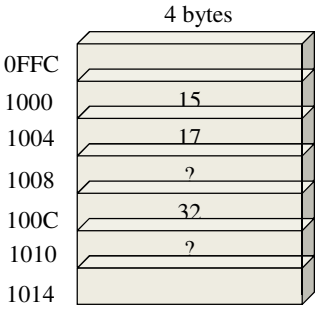
```
int ar[5];
ar[0] = 15;
ar[1] = 17;
ar[3] = ar[0] + ar[1];
```

sizeof(ar)

evaluates to 20 (i.e., 5*4-bytes)

sizeof(ar[0])

evaluates to 4



Arrays and Pointers

- Initialising Arrays
  - Prior to ANSI: declaration outside of a function or precede with the *static* keyword - these methods gave the array a *fixed duration*.
  - By default fixed duration arrays have all elements initialised to zero.
  - Assigning different initial values:
    - static int a_ar[5];
    - static int b_int[5] ={1, 2, 3.5, 4, 5};
  - Too many initialisation values is an error.
  - Too few initialisation values, and the remaining elements are initialised to zero.
    - static int c_int[5] ={1, 2, 3};
  - When initialising, you may omit the array size:
    - static char d_ar[] = {'a', 'b', 'c', 'd'};

Arrays and Pointers

ANSI Feature - Initialisation of Arrays

Most older C compilers require an array to have fixed duration to be initialised. This means that the declaration must occur outside of a function or be preceded by the *static* keyword. The ANSI Standard, however, permits automatic arrays to be initialised as well.

The rules for initialising automatic arrays are similar to the rules for initialising fixed arrays. As with fixed arrays, the uninitialised elements in an **automatic** array are initialised to zero. However, if no initialiser is present, none of the elements receive a default initial value (as is the case with the old semantics). The initialisation values must be constant expressions.

Arrays and Pointers

```
#define ILLEGAL_VAL -1

 char encode (char ch){

 static unsigned char  encoder[128]={127, 124, 121, 118, 115, 112,
                                                  .....5,2,0
                                          };

 if (ch > 127)

   return ILLEGAL_VAL;

 else

   return encoder[ch];

}
```

Arrays and Pointers

- Pointer Arithmetic
  - In C you can add/subtract integers to/from pointers
    - p+3 /* three objects after the object that p points to. */
    - The compiler multiplies 3 by the size of the object pointer to by p and adds the result to p - *Scaling*
  - Subtracting two pointers that point to the same type of object yields the number of objects between the two pointers.
    - &a[3] - &a[0] /* evaluates to 3 */
    - &a[0] - &a[3] /* evaluates to -3 */

```
long *p1, *p2;
int j;
char *p3;
p2 = p1 +4;
j  = p2 - p1;
j  = p1 - p2;
p1 = p2 - 2;
p3 = p1 -1;
j  = p1 - p3;
```

Arrays and Pointers

- Null Pointer
  - Guaranteed not to point to a valid object.
  - Assigned the integral value zero.
  - Useful in control-flow statements since the zero-valued pointer evaluates to FALSE, whereas all other pointer values evaluate to TRUE.

```
char *p;
p = 0; /* makes p a null pointer
        * no cast needed
        */
```

```
char *p;
while (p){
      /* iterate until p is a null pointer */
}
```

Arrays and Pointers

- Passing Pointers as Function Arguments
  - Normally the compiler complains if you try to mix different types of pointer. The exception occurs when you pass pointers as arguments.
  - In the absence of function prototyping, the compiler does not check to make sure that the type of the actual argument is the same as the type of the formal argument. If the types are different, strange behaviour can result.

Arrays and Pointers

- Accessing Array Elements Through Pointers

```
short ar[4];
short *p;
p = &ar[0]; /* *p is the same as ar[0] */
*(p+3); /* same as ar[3] */
```

  - In fact: *(p+e) is the same as ar[e] for integer expression e.
  - *Adding an integer to a pointer that points to the beginning of an array, and then dereferencing that expression, is the same as using the integer as a subscript value to the array.*
  - *An array name, not followed by a subscript, is interpreted as a pointer to the initial element of the array.*
  - ar is the same as &ar[0]
  - Thus ar[n] is the same as *(ar +n). This is unique to C.

Arrays and Pointers

- Accessing Array Elements Through Pointers
  - Array names and pointer variables can be used interchangeably to reference array elements. However, pointer variables can be changed whereas array names cannot.
  - Scaling allows the use of increment and decrement operators to point to next and previous elements of an array.

```
float ar[5], *p;
p = ar;
ar = p;
&p = ar;
ar++;
ar[1] = *(p+3);
p++;
```

Arrays and Pointers

- Passing Arrays as Function Arguments

```
int main(){
  extern float func();
  float x, farray[5];
  x = func(farray);
}
```

Form 1

```
func(float *ar){
 .....
}
```

Form 2

```
func(float ar[]){
.....
}
```

Form 2 declares *ar* to be an array of indeterminate size. The compiler converts *ar* into a pointer to a float, just as in Form 1.

Superior readability

It is also legal to declare the size of the array in an argument declaration. The compiler uses this size information for bound-checks only - provided this feature is supported.

Arrays and Pointers

Because of the pointer-array equivalence you can access *ar* as if it were an array. However, you cannot find out the size of the array in the *called function* by using the *sizeof* operator on the argument.

```
void print_size(float arg[]){
 printf("arg size = %d", sizeof(arg));
}
int main(){
  void print_size();
  float farray[10];
  printf("array size = %d",sizeof(farray));
  print_size(farray);
  exit(0);
}
```

It is often a good idea to pass the size of the array along with the base address.

You can obtain the number of elements in an array by dividing the size of the array by the size of each element

```
sizeof(farray) /sizeof(farray[0])
```

Arrays and Pointers

Bug Alert- Walking off the end of an array

C does not require compilers to check array bounds. This means that you can attempt to access elements for which no memory has been allocated. The results are unpredictable. Sometimes you will access memory that has been allocated for other variables. Sometimes you will attempt to access special protected areas of memory and your program will abort.

```
int main(){
  int ar[10],j;
  for (j=0; j<=10; j++)
    ar[j] = 0;
  exit(0);
}
```

Keep functions small. Catch these bugs early before they become a major problem.

Arrays and Pointers

- Strings
  - A string is an array of characters terminated by a *null character*- a character with a numeric value of zero. (\0).
  - A string constant (literal) is a series of characters enclosed in double quotes. It has the data type of array of char. Each char in the string takes up one byte. The compiler automatically appends a null character to designate the end of the string.
  - The array is one element longer than the number of characters in the string.
  - If you specify the size of the array you must allocate enough chars to hold the strings.

```
static char str[] = “some text”;
static char str[10] = “yes”;/* zero as default in other elements */
static char str[4] = “four”;/*illegal*/
```

Arrays and Pointers

You may also initialise a *char* pointer with a string constant

```
char * ptr = “more text”;
```

The pointer declaration creates an additional 4-byte variable for the pointer. The pointer is a variable that is initialised with the address of the array’s initial element. If you assign a different address value to the pointer the address with which it was initialised is lost. This contrasts with the static char str[] declaration.

You can assign a string constant to a pointer that points to a char. You must, however, be careful about allocating enough memory for the string.

Arrays and Pointers

```
int main(){
  char array[10];
  char *ptr1 = “10 spaces”;
  char *ptr2;
  array = “not OK”; /* cannot reassign to  constant pointer */
  array[5] = ‘a’;       /* OK */
  ptr1[5] = ‘b’;        /* OK */
  ptr1 = “OK”;          /* OK */
  ptr1[5] = ‘c’;        /* questionable due to prior assignment */
  *ptr2 = “not OK”; /* type mismatch */
  ptr2 = “OK”;
  exit(0);
}
```

Arrays and Pointers

- Strings versus Chars

```
char ch   = 'a'; /* one byte allocated for 'a' */
char *ps = "a"; /* two bytes allocated for "a" and an implementation
                   * number of bytes allocated for the pointer ps */
*p = 'a'; /* assigning a char constant through a dereferenced pointer */
*p = "a";/* incorrect to assign a string to a dereferenced char pointer */
p  = "a"; /* OK */
p  = 'a'; /* illegal - p is pointer, not a char */
```

Initialisations and assignments are not symmetrical:

```
char * p = "string"; /* is OK */
*p = "string"; /* is not */
```

This is true for all data types:

```
float *pf = &f; /* OK */
*pf = &f; /* Illegal */
```

Arrays and Pointers

- Reading and writing strings
  - *scanf*: argument is an array of characters, input string is terminated by any space character.
  - *printf*: argument is a pointer to a null-terminated array of characters.
  - Format specifier is %s
- String length function

```
int strlen (char str[]){
  int i =0;
  while(str[i])
    ++i;
  return i;
}
```

```
int strlen (char *str){
  int i;
  for(i=0; *str++;  i++)
       ; /* Null Statement */
  return i;
}
```

Arrays and Pointers

ANSI Feature - string concatenation

The ANSI Standard states that two adjacent string literals will be concatenated into a single null-terminated string. For example, the statement

```
printf (“one...” “two...” “three...\n”);
```

is treated as if it had been written

```
printf (“one...two...three...\n”);
```

The terminating null characters of the strings are not included in the concatenated string. We will see that this feature is useful for macros which expand into string literals. It is also used to break up long strings which would otherwise require the continuation character.

```
printf(“this is a very long string that cannot fit \
        on one line\n”);
printf(“this is a very long string that cannot fit”
       ”on one line\n”);
```

Arrays and Pointers

- String Copy Function

```
void strcpy(char s1[], s2[]){
  int  i;
  for (i=0; s1[i]; ++i)
    s2[i] = s1[i];
  s2[++i] = ‘\0’;
}
```

```
void strcpy(char *s1, *s2){
  int  i;
  for (i=0; *(s1+i); ++i)
    *(s2+i) = *(s1+i);
  s2[++i] = ‘\0’;
}
```

```
void strcpy(char *s1, *s2){
  while(*s2++ = *s1++)
    ;
}
```

Arrays and Pointers

- Pattern Matching (strstr in the ANSI library)
  - The function accepts two arguments, both pointers to character strings. It searches the first string for an occurrence of the second string. It returns the byte position of the occurrence, if successful, and -1 otherwise.

```
int pat_match(char str1[], str2[]){
  int j, k;
  for (j=0;j<strlen(str1);++j){
    for (k=0;(k<strlen(str2) && (str2[k] == str1[k+j])); k++)
          ;
    if (k==strlen(str2))
       return j;
  }
  return -1;
}
```

Arrays and Pointers

Pattern Matching Version 2

```
Pat_match(char *str1, *str2){
  char *p, *q, *substr;
  for (substr = str1; *substr; substr++){
    p = substr;
    q = str2;
    while (*q)
      if (*q++ != *p++)
    goto no_match;
    return substr -str1;
    no_match: ;
  }
  return -1;
}
```

We have to use the goto to get out of the while loop and continue the for loop at the same time.

The semicolon is required after the label so that the label prefixes a statement (albeit a null one).

Arrays and Pointers

strcpy()	copies a string to an array
strncpy()	copies a portion of a string to an array
strcat()	appends one string to another
strncat()	copies a portion of one string to another
strcmp()	compares two strings
strncmp)	compares two strings up to a specified number of characters
strchr()	finds the 1st occurrence of a specified character in a string
strcspn()	computes the length of a string that does not contain a specified char.
strpbrk()	finds the 1st occurrence of any specified character in a string
strrchr()	finds the last occurrence of any specified character in a string
strspn()	computes the length of a string that contains only specified chars.
strstr()	find the 1st occurrence of one string embedded in another
strtok()	breaks a string into a sequence of tokens
strerror()	maps an error number with a textual error message
strlen()	computes the length of a string.

Arrays and Pointers

- Multidimensional Arrays
  - int x[3][4][5]; is a 3-element array of 4-element arrays of 5-element arrays.

```
static int magic[5][5] = { { 17, 24, 1,  8, 15},
                           {23,  5,  7, 14, 16},
                           {4,   6, 13, 20, 22},
                           {10, 12, 19, 21, 23},
                           {11, 18, 25, 2,  9}
                         }
```

int ar[2][3]={{0,1,2},{3,4,5}};

ar[0][0]	**0**
ar[0][1]	**1**
ar[0][2]	**2**
ar[1][0]	**3**
ar[1][1]	**4**
ar[1][2]	**5**

Multidimensional arrays are stored in row-major order, which means that the last subscript varies fastest.

Arrays and Pointers

The array reference ar[1][2] is interpreted as *(ar[1] + 2) which is further expanded to

*(*(ar + 1) + 2)

The 1 is scaled to the size of a 3-element array of *ints* (we assume each is 4 bytes long) and the 2 is scaled to the size of an *int*.

*((int *) ((char *) ar + (1*3*4) + (2*4))

The (char *) cast is used to turn off scaling because we have already made the scaling explicit. The (int *) cast ensures that we get all four bytes of the integer when we dereference the address value.

*(int *) ((char *) ar + 20)

The value 20 has been scaled so that it represents the number of bytes to skip.

Specifying fewer subscripts than there are dimensions results in a pointer to the base type of the array. Thus, ar[1] is the same as &ar[1][0] -- a pointer to an *int*.

Arrays and Pointers

- Initialising Multidimensional Arrays
  - If there are too few initialisers, the extra elements in a row are initialised to zero.

static int example[5][3] = { {1,2,3},{4},{5,6,7}};

1	2	3
4	0	0
5	6	7
0	0	0
0	0	0

If we omit the inner brackets:

static int example[5][3] = { 1,2,3,4,5,6,7};

1	2	3
4	5	6
7	0	0
0	0	0
0	0	0

You may omit the first size specification but you must specify the other sizes.

static int ar[][3][2] = {{{1,1},{0,0},{1,1}},{{0,0},{1,2},{0,1}}}; a 2-by-3-by-2 array

static int ar[][] = {1,2,3,4,5,6}; This is illegal --- should the compiler create a 2-by-3 array or a 3-by-2 array? There is no way to tell. However, if you specify the size of each array other than the first, the declaration becomes unambiguous.

Arrays and Pointers

Bug Alert-- referencing elements in a multidimensional array

ar[1,2] = 0; /* Legal, but probably wrong. */

ar[1][2] = 0; /* correct */

The comma notation is used in some other languages, such as Pascal and FORTAN. In C, this notation has a very different meaning because the comma is a C operator in its own right. The first statement above causes the compiler to evaluate the first expression and discard the result; then it evaluates the second expression. The result of the comma expression is the value of the rightmost operand, so the value 2 becomes the subscript to *ar*. As a result the array reference accesses element 2 of *ar*.

If *ar* is a two-dimensional array of *ints*, the type of *ar[2]* is a pointer to an *int*, so this mistake will produce a type incompatibility error. This can be misleading since the real mistake is using a comma instead of brackets.

Arrays and Pointers

- Passing Multidimensional Arrays as Arguments
  - You may omit the size of the array being passed, but you must specify the size of each element in the array. The first size is only used for bound checking -- if at all!
  - The compiler interprets the declaration of received_arg as

    int (*received_arg)[6][7];
  - We could also just pass a pointer to the first element and pass the dimensions of the array as additional arguments. In this case we pass a pointer to a pointer to a pointer to an int.

```
void f1(){
  int ar[5][6][7];
  f2(ar);
}
```

```
void f2(int received_arg[][6][7]){
.....
}
```

```
void f2(int ***received_arg;
        int dim1, dim2, dim3)
```

Arrays and Pointers

– With this approach, you do not need to know ahead of time the shape of the multidimensional array. The disadvantage is that you need to manually perform the indexing arithmetic to access an element. For example, to access ar[x][y][z] in f2(), you would need to write:

((int) received_arg + x *dim3*dim2 + y*dim3+ z);

– Note that we need to cast recieved_arg to a pointer to an int because we are performing our own scaling. With this method f2() can accept three-dimensional arrays of any size and shape.

Arrays and Pointers

- Multidimensional Arrays Example

```
typedef enum { SPECIAL = -2, ILLEGAL, INT, FLOAT,
                DOUBLE,POINTER, LAST) TYPES;
int type_needed (int type1, int type2){
 static TYPES result_type[LAST][LAST] = {
 /*                int          float      double      pointer
 */
 /*int     */   INT,        DOUBLE,   DOUBLE,   POINTER,
 /*float   */   DOUBLE , DOUBLE,   DOUBLE,   ILLEGAL,
 /*double*/     DOUBLE,  DOUBLE,   DOUBLE,   ILLEGAL,
 /*pointer*/    POINTER,  ILLEGAL,   ILLEGAL,  SPECIAL
 }
 int  result = result_type[type1][type2];
 if (result == ILLEGAL)
   printf("Illegal pointer operation.\n");
 return result;
}
```

Notes:

The use of enum to define constants for all of the return data types. We can add new types without worrying about what integer value is used to represent them.

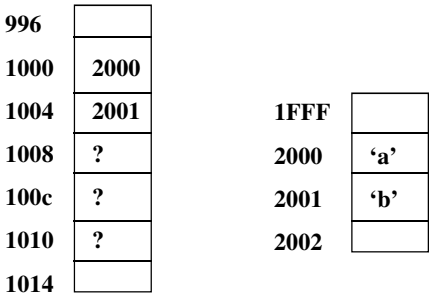
LAST represents the total number of types.

The use of comments to make the array readable.

In the special case where both operands are pointers, the expression is only legal if the pointers point to the same type of object and if the operator is a minus sign, in which case the result is int. To make this function perfect, therefore, we would need to determine what type of pointers the operands are and what the operand is.

Arrays and Pointers

- Arrays of Pointers
  - In certain situations it is useful to employ an array of pointers:
    - char *ar_of_p[5]; /* not a pointer to an array of chars */
    - char c0 = 'a';
    - char c1 = 'b';
    - ar_of_p[0] = &c0;
    - ar_of_p[1] = &c1;
  - Arrays of pointers are frequently used to access arrays of strings.



Arrays and Pointers

```
char  *month_text(int m){
  static char *month[13] = {"", "January", "February", "March","April", "May", "June",
                    "July",  "August", "September", "October", "November", "December"};
  if ((m<1) ||(m>12)){
    printf("illegal month value.\n");
    exit(1);
  }
  return month[m];
}
```

month[13] instead of month[12]

It is fairly common practice to discard the first element of an array when the subscript values naturally start at 1.

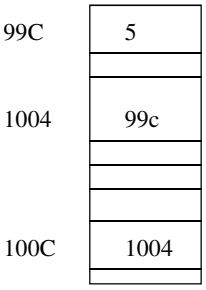
Note that the month names are not necessarily stored contiguously in memory. The chars making up each name must be contiguous but the names themselves can be placed anywhere the compiler sees fit.

Arrays and Pointers

- Pointers to Pointers
  - A pointer to a pointer is a construct used frequently in sophisticated programs.

    ```
    int **p;
    ```
  - Declares p to be a pointer to a pointer to an int.
  - Consider

    ```
    j = **p; /* assigns the int to j */
    int r =5;
    int *q = &r;
    int **p = &q;
    ```

  - We can assign values to r in 3 ways

    ```
    r = 10;  *q = 10; **p = 10;
    ```

Pointers to Pointers

As an example of using a pointer to a pointer, we will write a spelling checker. The function takes a string as input and compares it to an internal dictionary to see if it matches, If it does match, a null pointer is returned; if it doesn't, a pointer to the spelling of the closest match is returned. To make the program more useful (and illustrate pointers to pointers), we'll write it so that it tests not only English words, but French words as well.

We will create a 2 dimensional array of pointers. The first subscript selects the English or French dictionaries; the second subscript selects a particular word in one of the dictionaries. The function will take two arguments, a string and an indication of the language.

```
typedef enum{FRENCH, ENGLISH, LANG_NUM} LANGUAGE;
extern char *check_spell();
#define NULL (char *)0;
```

Spell Check Version 1

- Normally the dictionary would be stored in a file
- strcmp() return 0 if the two strings are equal and the difference between the first two differing chars if they are not equal.
- If the input does not match any string in the dictionary, we return a pointer to the closest spelling.
- We use a NULL pointer at the end of each list of words.
- With the language selector, we cut our work in half since we need to check the words in only one of the dimensions.
- One inefficiency in this version is the element reference dict[language][j]. A fair amount of arithmetic needs to be done in determining offset values and scaling them to the proper size. Eliminating one or both subscript operators can make the function more efficient.

Spell Check Version 2

- z points to an element of one of two arrays, either dict[ENGLISH] of dict[FRENCH]
- This type of improvement -- removing subscripts so that the compiler can avoid excessive pointer arithmetic -- is called *strength reduction.*

Spell Checker V1

```
#include<spell.h>
#define MAX_WORDS 50
/*Dictionary in alphabetic order with NULL as last entry */
static char *dict[LANG_NUM][MAX_WORDS] = {
        { “aardvark”, “abacus”, “abash”, “abbot”, “abhor”, “able,” “about”, NULL},
        {“abeille”, “absence,” “absurde”, “accepter”, “accident”, “accord”, “achat”, “acheter”, NULL} };
/* Return NULL pointer if str is found in dictionary, otherwise return a pointer to the closest match. */
char *check_spell(char * str, LANGUAGE language){
 int j, diff;
 /*Iterate over the words in the dictionary */
 for (j=0;dict[language][j] != NULL; ++j){
  diff = strcmp(str, dict[language][j]);
   /* Keep going if str is greater than the dict entry */
  if (diff >0)
    continue;
   if (diff ==0)
     return NULL; /* Match */
     /*No match, return closest spelling */
  return dict[language][j];
 }
 return dict[language][j-1];
}
```

Spell Checker V2

```
#include<spell.h>
#define MAX_WORDS 50
/*Dictionary in alphabetic order with NULL as last entry */
static char *dict[LANG_NUM][MAX_WORDS] = {
        { “aardvark”, “abacus”, “abash”, “abbot”, “abhor”, “able,” “about”, NULL},
        {“abeille”, “absence,” “absurde”, “accepter”, “accident”, “accord”, “achat”, “acheter”, NULL} };
/* Return NULL pointer if str is found in dictionary, otherwise return a pointer to the closest match. This time use pointers
      instead of array references.*/
char *check_spell(char * str,  LANGUAGE language){
  int j;
  char **z;
  /*Iterate over dictionary  entries*/
  for (z=dict[language]; *z; z++){
    diff = strcmp(str, *z);
     /* Keep going if str is greater than the dict entry */
    if (diff >0)
      continue;
    if (diff ==0)
      return NULL; /* Match */
   /*No match, return closest spelling */
    return *z;
  }
  return z-1;
 }
```