Arrays, Pointers, and Strings
Address Arithmetic

- Address of operator &
  The value of an expression `&x` is an address
- Other expressions yield addresses
  - the name of an array, written without brackets
  - the address is that of the first element of the array

    ```
    char s[50];
    ```

    s is equivalent to `&(s[0])`
  - we can combine the name of an array with integers

    `s` is equivalent to `&(s[0])`

    `s+i` is equivalent to `&(s[i])`

Arrays, Pointers, and Strings
Address Arithmetic

- Such expressions are valid even when the array elements are not 1 byte in size
- In general address arithmetic takes into account the size of the element

```
int a[10];
```

`a+i` is equivalent to `&(a[i])`

  - Such a capability leads some people to write:

```
for (i=0; i<10; i++) scanf("%d", a+i);
```

rather than

```
for (i=0; i<10; i++) scanf("%d", &a[i]);
```

Arrays, Pointers, and Strings
Address Arithmetic

- Indirection operator *
  The value of an expression such as `*a` is the object to which the address `a` refers

`*a`	is equivalent to	`a[0]`
`*(a+i)`	is equivalent to	`a[i]`

Arrays, Pointers, and Strings
Function Arguments and Arrays

- In C and C++ there is no need for special parameter-passing for arrays
- We pass the address of the first element of the array
- Which is the array name!
- We automatically have access to all other elements in the array

Functions
Function Arguments and Arrays

```
// MINIMUM: finding the smallest element of an
// integer array

#include <iostream.h>

int main()
{  int table[10], minimum(int *a, int n);
   cout << “Enter 10 integers: \n”;
   for (int i=0; i<10; i++) cin >> table[i];
   cout << “\nThe minimum of these values is “
        << minimum(table, 10) << endl;
   return 0;
}
```

Functions
Function Arguments and Arrays

```
// definition of minimum, version A

int minimum(int *a, int n)
{  int small = *a;
   for (int i=1; i<n; i++)
      if (*(a+i) < small)
         small = *(a+i);
   return small;
}
```

Functions
Function Arguments and Arrays

```
// definition of minimum, version B (for Better!)

int minimum(int a[], int n)
{  int small = a[0];
   for (int i=1; i<n; i++)
      if (a[i] < small)
         small = a[i];
   return small;
}
```

Arrays, Pointers, and Strings
Pointers

- In the following `p` is a pointer variable

```
int *p, n=5, k;
```

- Pointers store addresses

```
p= &n;
k = *p  // k is now equal to???
```

  - `*` is sometimes known as a dereferencing operator and accessing the object to which the pointer points is known as dereferencing

```
p= &n
```

Arrays, Pointers, and Strings
Pointers

- It is essential to assign value to pointers
  - after declaring `p` we must not use `*p` before assigning a value to `p`.

```
int main()
{  char *p, ch;
   *p = ‘A’; // Serious error!
   return 0;
}
```

Arrays, Pointers, and Strings
Pointers

- It is essential to assign value to pointers
  - after declaring `p` we must not use `*p` before assigning a value to `p`.

```
int main()
{  char *p, ch;
   p = &ch;
   *p = ‘A’;
   return 0;
}
```

Arrays, Pointers, and Strings
Pointers

- Pointer conversion and void-pointers

```
int i;
char *p_char;

p_char = &i; // error: incompatible types
             // pointer_to_char and
             // pointer_to_int

p_char = (char *)&i;
             // OK: casting pointer_to_int
             // as pointer_to_char
```

Arrays, Pointers, and Strings
Pointers

- In C++ we have generic pointer types: void_pointers

```
int i;
char *p_char;
void *p_void;

p_void = &i; // pointer_to_int to pointer_to_void
p_char = (char *)p_void;
              // cast needed in C++ (but not ANSI C)
              // for pointer_to_void to
              // pointer_to_int
```

Arrays, Pointers, and Strings
Pointers

- void_pointers can be used in comparisons

```
int  *p_int;
char *p_char;
void *p_void;

if (p_char == p_int) ... // Error
if (p_void == p_int) ... // OK
```

- Address arithmetic must not be applied to void_pointers. Why?

Arrays, Pointers, and Strings
Pointers

- Typedef declarations
  - used to introduce a new identifier denote an (arbitrarily complex) type

```
typedef double real;
typedef int *ptr;
...
real x,y;   // double
ptr p;      // pointer_to_int
```

Arrays, Pointers, and Strings
Pointers

- Initialization of pointers

```
int i, a[10];
int *p = &i;  // initial value of p is &i
int *q = a;   // initial value of q is the
              // address of the first element
              // of array a
```

Arrays, Pointers, and Strings
Strings

- Recap: addresses can appear in the following three forms
  - expression beginning with the & operator
  - the name of an array
  - pointer
- Another, fourth, important form which yields an address
  - A string (string constant or string literal)
  - “ABC”

Arrays, Pointers, and Strings
Strings

- “ABC”
  - effectively an array with four char elements:
  - ‘A’, ‘B’, ‘C’, and ‘\0’
  - The value of this string is the address of its first character and its type is pointer_to_char

```
*”ABC”          is equal to   ‘A’
*(“ABC” + 1)    is equal to   ‘B’
*(“ABC” + 2)    is equal to   ‘C’
*(“ABC” + 3)    is equal to   ‘\0’
```

Arrays, Pointers, and Strings
Strings

- “ABC”
  - effectively an array with four char elements:
  - ‘A’, ‘B’, ‘C’, and ‘\0’
  - The value of this string is the address of its first character and its type is pointer_to_char

```
”ABC”[0]          is equal to   ‘A’
”ABC”[1]          is equal to   ‘B’
”ABC”[2]          is equal to   ‘C’
”ABC”[3]          is equal to   ‘\0’
```

Arrays, Pointers, and Strings
Strings

- Assigning the address of a string literal to a pointer variable can be useful:

```
// POINTER
#include <stdio.h>
int main()
{  char *name = “main”;
   printf(name);
   return 0;
}
```

```
// POINTER
#include <iostream.h>
int main()
{  char *name = “main”;
   cout << name;
   return 0;
}
```

Arrays, Pointers, and Strings
Strings Operations

- Many string handling operations are declared in string.h

```
#include <string.h>

char s[4];

s = “ABC”; // Error: can’t do this in C; Why?
strcpy(s, “ABC”);  // string copy
```

Arrays, Pointers, and Strings
Strings Operations

- Many string handling operations are declared in string.h

```
#include <string.h>
#include <iostream.h>

int main()
{  char s[100]=“Program something.”, t[100];
   strcpy(t, s);
   strcpy(t+8, “in C++.”;
   cout << s << endl << t << endl;
   return 0;
}  // what is the output?
```

Arrays, Pointers, and Strings
Strings Operations

- Many string handling operations are declared in string.h

```
strnen(string);
   // returns the length of the string
E.g.
int length;
char s[100]=“ABC”;
length = strlen(s); // returns 3
```

Arrays, Pointers, and Strings
Strings Operations

- Many string handling operations are declared in string.h

```
strcat(destination, source);
   // concatenate source to destination

strncat(destination, source, n);
   // concatenate n characters of source
   // to destination
   // programmer is responsible for making
   // sure there is enough room
```

Arrays, Pointers, and Strings
Strings Operations

- Many string handling operations are declared in string.h

```
strcmp(string1, string2);
   // returns 0 in the case of equality
   // returns <0 if string1 < string2
   // returns >0 if string1 > string2

strncmp(string1, string2, n);
   // same as strcmp except only n characters
   // considered in the test
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

- Array declarations
  - require a constant length specification
  - cannot declare variable length arrays
- However, in C++ we can create an array whose length is defined at run-time

```
int n;
char *s;
...
cin >> n;
s = new char[n];
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

- If memory allocation fails
  - would have expected `new` to return a value `NULL`
  - however, in C++ the proposed standard is that instead a *new-handler* is called
  - we can (usually) force `new` to return `NULL` by calling `set_new_handler(0);` before the first use of `new`
  - This has been adopted in the Borland C++ compiler

Arrays, Pointers, and Strings
Dynamic Memory Allocation

```
// TESTMEM: test how much memory is available
#include <iostream.h>
#include <new.h> required for set_new_handler

int main()
{  char *p;
   set_new_handler(0); // required with Borland C++
   for (int i=1;;i++)  // horrible style
   {  p = new char[10000];
      if (p == 0) break;
      cout << “Allocated: “ << 10 * i << “kB\n”;
   }
   return 0;
} // rewrite this in a better style!
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

- Memory is deallocated with `delete()`
  - `p = new int // deallocate with:`
  - `delete p;`

  - `p = new int[m] // deallocate with:`
  - `delete[] p;`

  - `delete` is only available in C++

Arrays, Pointers, and Strings
Dynamic Memory Allocation

- `malloc()`
  - standard C memory allocation function
  - declared in stdlib.h
  - its argument defines the number of bytes to be allocated

```
#include <stdlib.h>
int n;
char *s;
...
cin > n;
s = (char *) malloc (n);
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

- `malloc()`
  - but to allocate an array of floats:

```
#include <stdlib.h>
int n;
float *f;
...
cin > n;
s = (float *) malloc (n * sizeof(float));
```

  - `malloc()` returns `NULL` if allocation fails

Arrays, Pointers, and Strings
Dynamic Memory Allocation

- malloc()

```
s = (float *) malloc (n * sizeof(float));
if (s == NULL)
{  cout << “Not enough memory.\n”;
   exit(1); // terminates execution of program
}           // argument 1: abnormal termination
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

- `calloc()`
  - Takes two arguments
    - » number of elements
    - » size of each element in bytes
  - all values are initialized to zero
  - `calloc()` returns `NULL` if allocation fails

Arrays, Pointers, and Strings
Dynamic Memory Allocation

- Memory is deallocated with `free()`
  - `free(s);`

Arrays, Pointers, and Strings
Input and Output of Strings

- Input

```
char[40] s;
...
scanf(“%s”, s);  // skips whitespace and terminates on
                 // whitespace
cin >> s;        // same as scanf
gets(s);         // reads an entire line

// problems if more than 40 chars are typed:
// ABCDEFGHIJKLMNOPQRSTUVWZYZabcedfghijklmnopqrstuvwzyz
// requires a string of 53 elements
```

Arrays, Pointers, and Strings
Input and Output of Strings

- Input

```
char[40] s;
...
scanf(“%39s”, s);      //reads at most 39 characters
cin >> setw(40) >> s; // same as scanf
fgets(s, 40, stdin);  // reads a line of at most 39
                      // characters, including \n
cin.getline(s, 40);   // reads a line of at most 39
                      // characters, including \n
                      // but doesn’t put \n in s
```

Arrays, Pointers, and Strings
Input and Output of Strings

- Output

```
char[40] s;
...
printf(s);              // Display just the contents of s
printf("%s", s);        // same
cout << s;              // same
printf("%s\n", s);      // Display s, followed by newline
puts(s);                // same
```

Arrays, Pointers, and Strings
Input and Output of Strings

- Output

```
// ALIGN1: strings in a table, based on standard I/O
#include <stdio.h>

int main()
{  char *p[3] = {“Charles”, “Tim”, “Peter”};
   int age[3] = {21, 5, 12}, i;
   for (i=0; i<3; i++)
      printf(“%-12s%3d\n”, p[i], age[i]); // left align
   return 0;
}
```

Arrays, Pointers, and Strings
Input and Output of Strings

- Output

```
// ALIGN2: strings in a table, based on stream I/O
#include <iostream.h>
#include <iomanip.h>
int main()
{  char *p[3] = {“Charles”, “Tim”, “Peter”};
   int age[3] = {21, 5, 12}, i;
   for (i=0; i<3; i++)
     cout << setw(12) << setiosflags(ios::left) << p[i]
          << setw(3) < resetiosflags(ios::left)
          << age[i];
return 0;
}
```

Arrays, Pointers, and Strings
Multi-Dimensional Arrays

- A table or matrix
  - can be regarded as an array whose elements are also arrays

```
float table[20][5]
int a[2][3] = {{60,30,50}, {20,80,40}};
int b[2][3] = {60,30,50,20,80,40};
char namelist[3][30]
= {“Johnson”, “Peterson”, “Jacobson”);
...
for (i=0; i<3; i++)
   cout << namelist[i] << endl;
```

Arrays, Pointers, and Strings
Multi-Dimensional Arrays

- Pointers to 2-D arrays:

```
int i, j;
int a[2][3] = {{60,30,50}, {20,80,40}};
int (*p)[3]; // p is a pointer to a 1-D array
             // of three int elements
...
p = a; // p points to first row of a
a[i][j] = 0;          // all four statements
(*(a+i))[j] = 0;      // are equivalent
p[i][j] = 0;          // remember [] has higher
(*(p+i))[j] = 0;      // priority than *
```

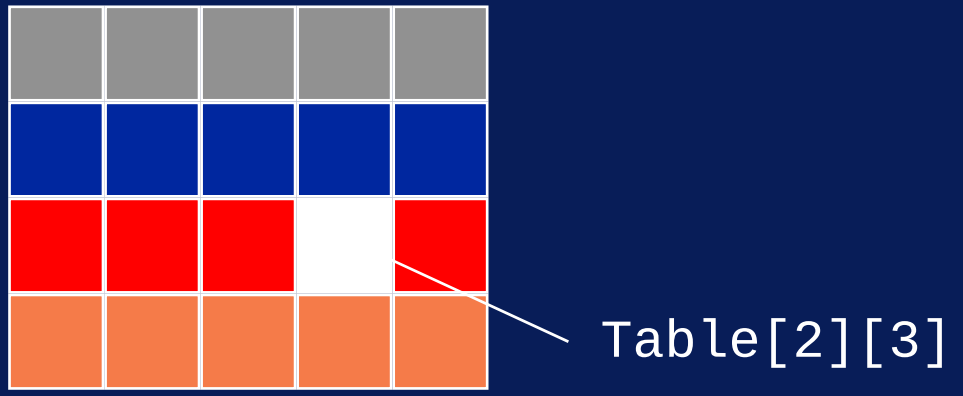
Arrays, Pointers, and Strings
Multi-Dimensional Arrays

- Function Parameters

```
int main()
{  float table[4][5];
   int f(float t[][5]);

   f(table);
   return 0;
}
int f(float t[][5]) // may omit the first dimension
{                   // but all other dimensions must
}                   // be declared since it must be
                    // possible to compute the
                    // address of each element. How?
```

Arrays, Pointers, and Strings
Multi-Dimensional Arrays



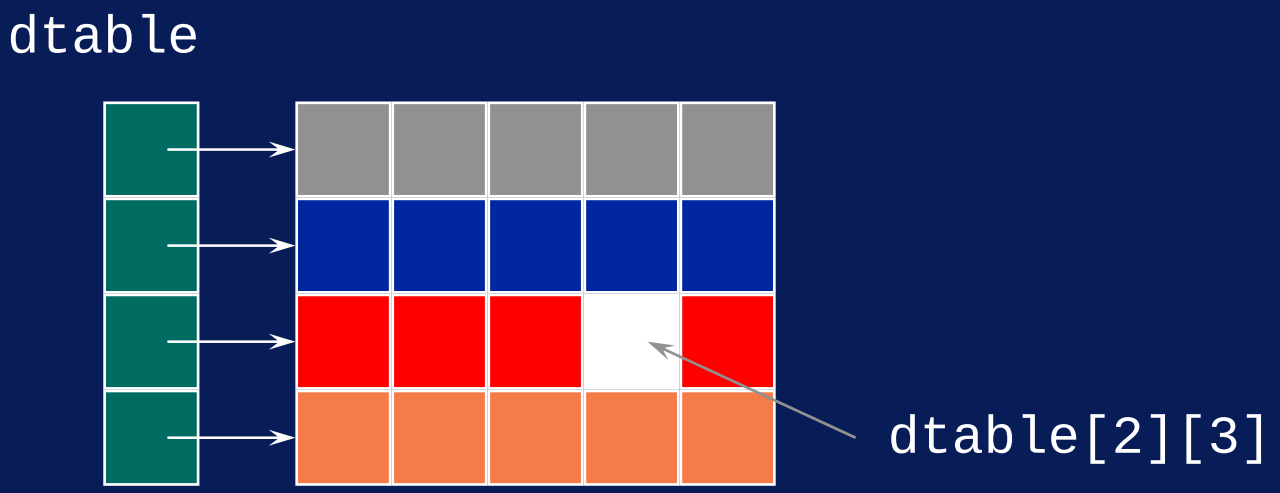
The address of Table[i][j] is computed by the mapping function 5*i + j (e.g. 5*2+3 = 13)

Arrays, Pointers, and Strings
Multi-Dimensional Arrays

- Arrays of Pointers
  - we can create 2-D ‘arrays’ in a slightly different (and more efficient) way using
    - » an array of pointers to 1-D arrays
    - » a sequence of 1-D arrays

```
float *dtable[4]; // array of 4 pointers to floats
set_new_handler(0);
for (i=0; i<20; i++)
{  dtable[i] = new float[5];
   if (dtable[i] == NULL)
   {  cout << “ Not enough memory”; exit(i);
   }
}
```

Arrays, Pointers, and Strings
Multi-Dimensional Arrays

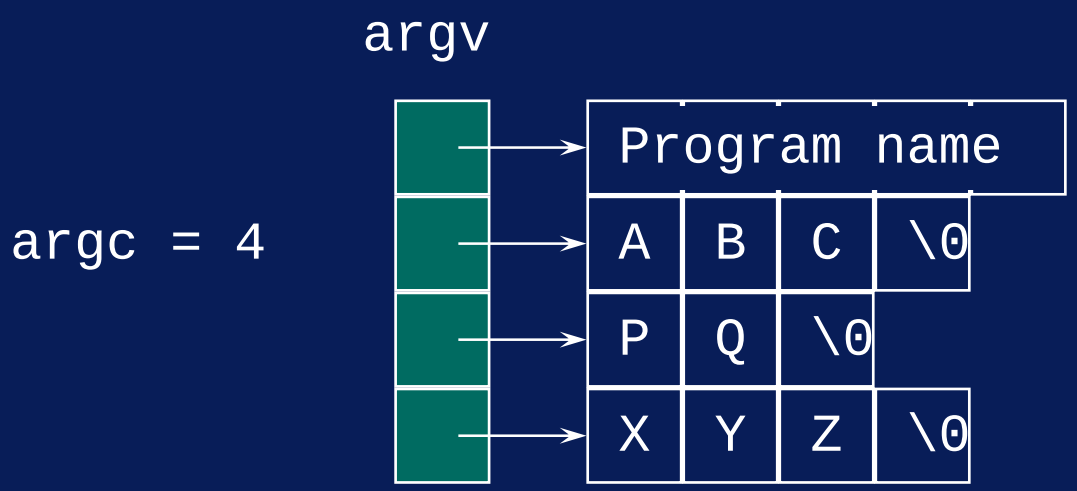



```
dtable[i][j] is equivalent to (*(dtable+i))[j] ...
there is no multiplication in the computation
of the address, just indirection.
```

Arrays, Pointers, and Strings
Program Parameters

- The main() function of a program can have parameters
  - called program parameters
  - an arbitrary number of arguments can be supplied
  - represented as a sequence of character strings
  - two parameters
    - » argc ... the number of parameters (argument count)
    - » argv ... an array of pointers to strings (argument vector)

Arrays, Pointers, and Strings
Program Parameters



Program parameters for an invocation of the form
program ABC PQ XYZ

Arrays, Pointers, and Strings
Program Parameters

```
// PROGPARAM:  Demonstration of using program parmeters
#include <iostream.h>

int main(int argc, char *argv[])
{  cout << “argc  = “ << argc << endl;
   for (int i=1; i<argc; i++)
      cout << “argv[“ << i << “]= “ << argv[i] << endl;
   return 0;
}
```

Arrays, Pointers, and Strings
In-Memory Format Conversion

- `sscanf()`
  - scans a string and converts to the designated type

```
#include <stdio.h>
...
 char s[50]="123   456 \n98.756";
 int i, j;
 double x;
 sscanf(s, "%d %d %lf", &i, &j, &x);
```

  - sscanf returns the number of value successfully scanned

Arrays, Pointers, and Strings
In-Memory Format Conversion

- `sscanf()`
  - fills a string with the characters representing the passed arguments

```
#include <stdio.h>
...
char s[50]="123   456 \n98.756";
sprintf(s,"Sum: %6.3f Difference:%6.3f",
          45 + 2.89, 45 - 2.89);
```

Arrays, Pointers, and Strings
Pointers to Functions

- In C and C++ we can assign the start address of functions to pointers

```
// function definition
float example (int i, int j)
{  return 3.14159 * i + j;
}

float (*p)(int i, int j); // declaration
...
p = example;
```

Arrays, Pointers, and Strings
Pointers to Functions

- And we can now invoke the function as follows

```
(*p)(12, 34); // same as example(12,34);
```

- We can omit the * and the () to get:

```
p(12, 34);    // !!
```

- Pointers to function also allow us to pass functions as arguments to other functions

Arrays, Pointers, and Strings
Exercise

11. Write and test a function to
   - read a string representing a WWW URL (e.g. http://www.cs.may.ie)
   - replace the // with \\
   - write the string back out again

Arrays, Pointers, and Strings
Exercises

12. Write a interactive user interface which allows a user to exercise all of the set operators for three pre-defined sets A, B, and C

    Commands should be simple single keywords with zero, one, or two operands, as appropriate

    - add A 10
    - union C A B
    - list A
    - intersection C A B
    - remove 1 B