Arrays

- An array is a sequence of objects (elements) all of which have the same type.

One-dimentional array

- The format for an array declaration is:

Type array_name[size];

Two-dimentional array

- The format for an array declaration is:

Type array_name[size1][size2];

Example

This code reads 4 numbers and then prints them in reverse order:

```
main( )
{
int a[4];
cout << "Enter 4 int numbers:\n';
for (int i = 0; i < 4; i++)
cin >> a[i];

cout << "Here they are in reverse order:\n";
for (i = 3; i >= 0; i--)
cout << a[i] << '\n';
}
```

Example

Using a symbolic constant to declare and process an array

```
main( )
{
const int size = 4;
int a[size];
cout << "Enter 4 int numbers:\n';
for (int i = 0; i < 4; i++)
cin >> a[i];

cout << "Here they are in reverse order:\n";
for (i = 3; i >= 0; i--)
cout << a[i] << '\n';
}
```

Example

To show that an array can be initialized with a single initializer list

```
main( ){
double a[4] = {22.2, 44.4, 66.6, 88.8};
for (int i = 0; i < 4; i++)
cout << a[i] << '\n';
}
```

- product part numbers:

```
int part_numbers[] = {123, 326, 178, 1209};
```

- student scores:

```
int scores[10] = {1, 3, 4, 5, 1, 3, 2, 3, 4, 4};
```

- characters:

```
char alphabet[5] = {'A', 'B', 'C', 'D', 'E'};
```

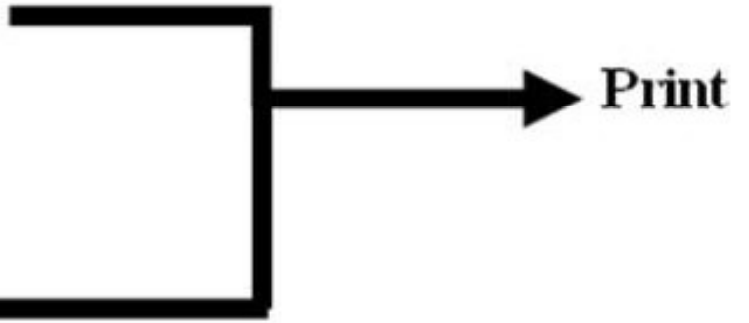
Example

To read and print a two dimensional array

```
main( )
{
int a[4][4];

for (int i = 0; i < 4; i++)
for (int j = 0; j < 4; i++)
cin>> a[i][j];

cout<< "the output array";
for (int i = 0; i < 4; i++)
{
cout<<'\n';
for (int j = 0; j < 4; i++)
cout<< a[i][j]<<'\t';
}
```



Example

To find the sum of numbers in main diagonal

```
main( )
{
int a[4][4];
for (int i = 0; i < 4; i++)
for (int j = 0; j < 4; i++)
cin>> a[i][j];
int sum=0;
for (int i = 0; i < 4; i++)
for (int j = 0; j < 4; i++)
if(i==j)
sum+=a[i][j];
cout << "sum=" << sum <<'\n';
}
```

Strings

- A string is an array of characters.
- An extra component is appended to the end of the array, and its value is set to the NUL character '\0'. This means that the total number of characters in the array is always 1 more than the string length.
- The functions declared in the <string.h> header file may be used to manipulate strings. These include the string functions.

In C++, a **string** is defined as a character array terminated by a null symbol ('\0').

'H'	'e'	'l'	'l'	'o'	'\0'

The following program reads (part of) a string entered by the user:

```
#include <stdio.h>
int main()
{
  char str[80];

  cout << "Enter a string: ";
  cin >> str; // read string from keyboard
  cout << "Here is your string: ";
  cout << str;

  return(0);}
```

Problem: Entering the string "**This is a test**", the above program only returns "**This**", not the entire sentence.

Reason: The C++ input/output system stops reading a string when the first **whitespace** character is encountered.

Solution: Use another C++ library function, **gets().**

```
#include <iostream.h>
#include <cstdio.h>
int main()
{
  char str[80]; // long enough for user input?

  cout << "Enter a string: ";
  gets(str); // read a string from the keyboard
  cout << "Here is your string: ";
  cout << str << endl;
  return(0);}
```

C++ Library Functions for Strings

C++ supports a range of string-manipulation functions.

The most common are:

- **`strcpy()`** : copy characters from one string to another
- **`strcat()`** : concatenation of strings
- **`strlen()`** : length of a string
- **`strcmp()`** : comparison of strings

strcpy(*to_string, from_string*) — String Copy:

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

int main()
{
  char a[10];

  strcpy(a, "hello");

  cout << a;
  return(0);
}
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
h	e	l	l	o	\0	?	?	?	?

strlen(*string*) — String Length

strlen(str) returns the length of the string pointed to by **str**, i.e., the number of characters **excluding** the null terminator.

```
#include <iostream.h>
#include <cstdio.h>
#include <cstring.h>

int main()
{
  char str[80];

  cout << "Enter a string: ";
  gets(str);

  cout << "Length is: " << strlen(str);
  return(0);}
```

strcat(*string_1*, *string_2*) — Concatenation of Strings

The strcat() function appends *s2* to the end of *s1*. String *s2* is unchanged.

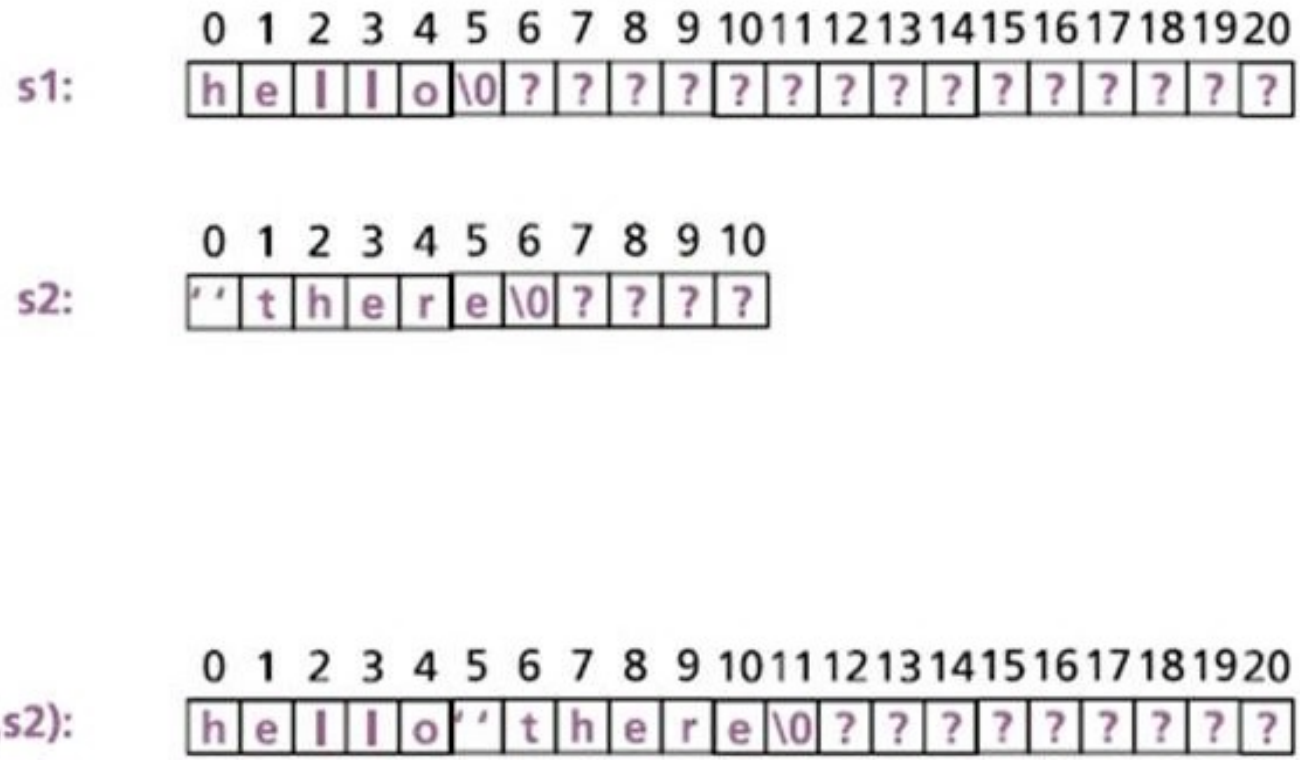
```
// includes ...
int main()
{
  char s1[21], s2[11];

  strcpy(s1, "hello");
  strcpy(s2, " there");
  strcat(s1, s2);

  cout << s1 << endl;
  cout << s2 << endl;
  return(0);
}
```

Note:

- The first string array has to be *large enough* to hold both strings:



- To be on the safe side:
  `strlen(s1concats2) >= strlen(s1) + strlen(s2)`

strcmp(*string_1*, *string_2*) — Comparison of Strings

The **strcmp(str_1, str_2)** function compares two strings and returns the following result:

- str_1 == str_2 : 0
- str_1 > str_2 : positive number
- str_1 < str_2 : negative number

```
// Comparing strings
#include <iostream.h>
#include <cstring.h>
#include <cstdio.h>

int main()
{
  char str[80];

  cout << "Enter password: ";
  gets(str);

  if(strcmp(str, "password")) {
    // strings differ
    cout << "Invalid password.\n";}
  else cout << "Logged on.\n";
  return(0);
}
```

Using the Null Terminator

Operations on strings can be simplified using the fact that all strings are null-terminated.

```
// Convert a string to uppercase
// ... includes ...
int main()
{
  char str[80];
  int i;

  strcpy(str, "this is a test");

  for(i=0; str[i]; i++)
    str[i] = toupper(str[i]);

  cout << str; return(0); }
```

Functions

Important Concepts about Functions

- Function prototype: Tells compiler argument type and return type of function

  int rootn(int);

  – Function header
  – Function declaration

- Function call: Prototype must match function definition
  – Can be Constants **rootn(4);**
  – Can be Variables **rootn(x);**
  – Can be Expressions **rootn(factorial(x)) ;** or **rootn(2 – 5/x);**

- Function definition: Function implementation

```
return-value-type function-name( parameter-list )
{
declarations and statements
}
```

Function Definitions

```
int square1( int y )
{
return y * y;
}
```

```
void square2( int y )
{
cout << y * y ;
}
```

- **return** keyword

– Returns data, and control goes to function's caller

– Function ends when reaches right brace or return

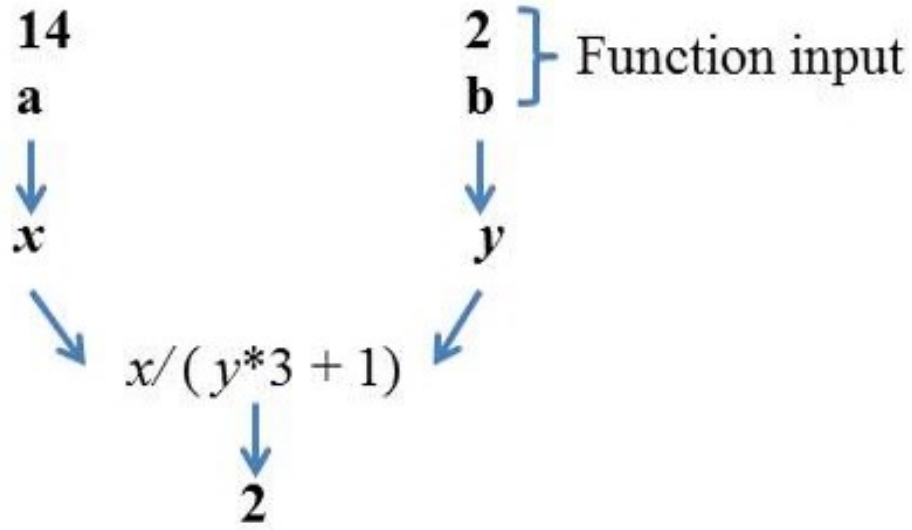
- Control goes to caller

Note:

- Functions cannot be defined inside other functions

Functions

```
int a = 14, b = 2;
cout<< fun(a,b);
```



Function definition

```
int fun(int x, int y)
{
return x /(y*3 + 1);
}
```

Finding the maximum of three floating-point numbers

```
#include <iostream>

double maximum( double, double, double );

main() {

double num1, num2, num3;

cout << "Enter three floating-point numbers: ";

cin >> num1 >> num2 >> num3;

cout << "Maximum is:" << maximum( num1, num2, num3 ) << endl;}

double maximum( double x, double y, double z ){

double max = x;

if ( y > max )

max = y;

if ( z > max )

max = z;

return max; }
```

Math Library Functions

- Perform common mathematical calculations

– Include the header file

#include <cmath>

– Trignometric functions (e.g. **cos**, **sin**, **tan**)

– Exponential and logarithmic functions (e.g. **exp**, **log**)

– Power functions (e.g. **pow**, **sqrt**)

– Rounding, absolute value and remainder functions (e.g. **ceil**, **floor**)

– All functions in math library return a **double**

- Example **cout << sqrt(900);**

http://www.cplusplus.com/reference/cmath/

Function	Header File	Purpose	Parameter(s) Type	Result
`abs(x)`	`<cmath>`	Returns the absolute value of its argument: `abs(-7) = 7`	`int` (`double`)	`int` (`double`)
`ceil(x)`	`<cmath>`	Returns the smallest whole number that is not less than `x`: `ceil(56.34) = 57.0`	`double`	`double`
`cos(x)`	`<cmath>`	Returns the cosine of angle: `x`: `cos(0.0) = 1.0`	`double` (`radians`)	`double`
`exp(x)`	`<cmath>`	Returns e^x, where `e = 2.718`: `exp(1.0) = 2.71828`	`double`	`double`
`fabs(x)`	`<cmath>`	Returns the absolute value of its argument: `fabs(-5.67) = 5.67`	`double`	`double`
`floor(x)`	`<cmath>`	Returns the largest whole number that is not greater than `x`:`floor(45.67) = 45.00`	`double`	`double`
`islower(x)`	`<cctype>`	Returns 1 (`true`) if `x` is a lowercase letter; otherwise, it returns 0 (`false`); `islower('h')` is 1 (`true`)	`int`	`int`
`isupper(x)`	`<cctype>`	Returns 1 (`true`) if `x` is an uppercase letter; otherwise, it returns 0 (`false`); `isupper('K')` is 1 (`true`)	`int`	`int`
`pow(x, y)`	`<cmath>`	Returns x^y; if `x` is negative, `y` must be a whole number: `pow(0.16, 0.5) = 0.4`	`double`	`double`
`sqrt(x)`	`<cmath>`	Returns the nonnegative square root of `x`; `x` must be nonnegative: `sqrt(4.0) = 2.0`	`double`	`double`
`tolower(x)`	`<cctype>`	Returns the lowercase value of `x` if `x` is uppercase; otherwise, it returns `x`	`int`	`int`
`toupper(x)`	`<cctype>`	Returns the uppercase value of `x` if `x` is lowercase; otherwise, it returns `x`	`int`	`int`

Call by Value versus Call by Reference

- Call by value (used by default)
  - Copy of data passed to function
  - Changes to copy do not change original
  - Prevent unwanted side effects

- Call by reference
  - Function can directly access data
  - Changes affect original

•Why should I need call by value?

– It is safer

•Arguments protected from side effects

•It works with a copy of the arguments passed in the function call

– But, it implies overhead

•Memory space and time needed to make the copies

•What if an argument is many bytes long?

•Why should I need call by reference?

– Efficiency

– But, there's more

Call by Value

```
//swap the value of two variables
void swap1(int x, int y)
{
int temp = x;
x = y;
y = temp;
}
```

Call by Reference

```
//swap the value of two variables
void swap2(int &x, int &y)
{
int temp = x;
x = y;
y = temp;
}
```

Calling **swap1** and **swap2**

- Call by value swap1(a, b);
  - x and y in swap gets copies of a and b
  - Local copies of the arguments are changed
- Call by reference swap2(a, b);
  - Reference (address) to a and b is copied to x and y
  - Arguments are changed

Recursive Functions

- Recursive functions N! = N * (N-1)!
  - Functions that call themselves
  - There must be a base case 0! = 1
- If not base case N > 0
  - Break problem into smaller problem(s)
  - Launch new copy of function to work on the smaller problem (recursive call/recursive step)
- Slowly converges towards base case
- Function makes call to itself inside the return statement

$$factorial(n) := n!$$
$$:= n * (n-1) * \ldots * 2 * 1$$

$$factorial(n) := 1 \quad \text{if } n = 1$$
$$:= n * factorial(n-1) \quad \text{if } n > 1$$

Iterative Implementation

```
int factorial_1 (int n)
{
int i, f=1;
for(i=1; i <= n; i++) f *= i;
return f;
}
```

Recursive Implementation

```
int factorial_2 (int n)
{
if(n == 1) return 1;
return n * factorial_2(n-1);
}
```

Comparison of Iterative and Recursive Functions

```
int factorial_2 (int n)
{
if(n == 1) return 1;
return n * factorial_2(n-1);
}
```

```
int factorial_1 (int n)
{
int i, f=1;
for(i=1; i <= n; i++) f *= i;
return f;
}
```

```
factorial_2(4)
(4 * factorial_2(3))
(4 * (3 * factorial_2(2)))
(4 * (3 * (2 * factorial_2(1))))
(4 * (3 * (2 * 1)))
(4 * (3 * 2))
(4 * 6)
24
```

Iteration	i	f	n
1	1	1	4
2	2	2	4
3	3	6	4
4	4	24	4
5	5		

Function that computes the sum of numbers from 0 to n

```
int sum (int n)
{
 int s = 0;
 for (int i=0; i<n; i++)
 s+= i;
 return s;
}
```

```
int sum (int n) {
 int s;
 if (n == 0) return 0;
else
 s = n + sum(n-1);
 return s;
}
```

```
sum(4)
(4 + sum(3))
(4 + (3 + sum(2)))
(4 + (3 + (2 + sum(1))))
(4 + (3 + (2 + (1 + sum(0)))))
(4 + (3 + 2+ (1 + 0)))
10
```

0	1	2	3	5	8	13	21	34	.....
1	2	3	4	.....					

```
int numbers(int n)
{
if (n <= 0) return 0;
else if (n == 1) return 1;
else return numbers(n − 1) + numbers(n − 2);
}
```

```
numbers(4) = numbers(3) + numbers(2)
numbers(3) = numbers(2) + numbers(1)
numbers(2) = numbers(1) + numbers(0)
numbers(1) =  1
numbers(0) =  0
```

Power function

```
int power( int x, int y )
{
if ( y == 0 )
return  1 ;
else if ( y == 1 )
return  x ;
else
return ( x * power( x, y-1 ));
}
```