Stream Input/Output

The header file <iostream> includes definitions for the classes*:

```
ios                         // the base class
streambuf                   // defines the buffer
istream                     // input
ostream                     // output
```

It also includes definitions of the following objects

```
extern istream cin
extern ostream cout
extern ostream cerr;
```

* This is a functionally equivalent simplification.

operator>>

The operator >> is defined within the class `istream` for the following types:

```
char*           /* strings signed and unsigned*/
char&           /* signed and unsigned */
short&          /* signed and unsigned */
int&            /* signed and unsigned */
long&
float&
double&
long double&
istream&        /* used for i/o manipulators */
```

The action performed by these operators is as follows:

1) Skip white space characters. (space, tab, newline)
2) Read characters that conform to the format of constants defined for the type. (For `char*` read characters until the next white space character, for `char&` read the next character.)
3) Convert the read string into the target type.

operator<<

The operator << is defined within class ostream for the following types:

```
char*            /* signed and unsigned */
char             /* signed and unsigned */
short            /* signed and unsigned */
int              /* signed and unsigned */
long
float
double
long double
ostream&         /* used for i/o manipulators */
```

The action of these operators is as follows:

1) Convert the argument to a character string in accordance with the format flags.

2) Output the result.

Format Flags

The format flags are defined within the base class (ios) as follows:

Flag Name	**Meaning if set**
`left`	left-adjust output
`right`	right-adjust output
`internal`	padding after sign or base indicator
`showbase`	use base indicator on output
`showpoint`	force decimal point and trailing zeros (floating output)
`uppercase`	upper-case hex or scientific output
`scientific`	use 1.2345E2 floating notation
`fixed`	use 123.45 floating notation

I/O Manipulators

The I/O manipulators are defined in header file iomanip.h. They provide an eligant way to set the format flags. The following manipulators are defined:

Manipulator	**Argument**	**Effect**
`dec`		Output integers in base 10
`hex`		Output integers in base 16
`oct`		Output integers in base 8
`resetiosflags`	`long`	Turns off the flag bits corresponding to those set in the argument
`setiosflags`	`long`	Turns on the flag bits corresponding to those set in the argument
`setfill`	`int`	Sets the fill character to the argument. The defalult is the space character.

Manipulator	**Argument**	**Effect**
`setprecision`	`int`	Sets the precision to the argument. The default is 6.
`setw`	`int`	Sets the width to the argument. Only affects the next output value, after which the width returns to the default of 0.
`ws`		Eat white space
`endl`		Write a new-line character and flush the buffer
`left`		Place the fill character after the value to pad up to the width.
`right`		Place the fill character before the value to pad up to the width.

Manipulator	Argument	Effect
`internal`		Place the fill character between the sign and the value.
`fixed`		Output float values as xxx.xxx. Default is general.
`scientific`		Output float values as x.xxxxxenn. Default is general.
`showpoint`		For float values, always output the decimal point and trailing zeros (as defined by the precision).

Note: some of these manipulators are defined in the standard, but not available in some compilers. Use of `setiosflags` and `resetiosflags` is a possible workaround.

Example

```
// FILE: PrntDemo.cpp
// A DEMONSTRATION OF FORMATTED OUTPUT USING MANIPULATORS
#include <iostream>
#include <iomanip>

int main ()
{
  using namespace std;
  int value = 0x68BF;
  short short_value = -1;
  long long_value = 123456789L;
  float x = -78.4569;
  double pi = 3.1415926536;
  cout << "Plain decimal integer value: " << value << '\n';
  cout << "Decimal integer value with forced sign: "
       << setiosflags(ios::showpos) << value
       << resetiosflags(ios::showpos)  << '\n';
  cout << "Decimal integer -- right justification: "
       << setiosflags(ios::right) << setw(10) << value
       << resetiosflags(ios::right) << '\n';
  cout << "Decimal integer -- zero fill on left: "
       << setfill('0') << setw(10) << value
       << setfill(' ') << '\n';
  cout << "Decimal integer -- left justification: "
       << setiosflags(ios::left) << setw(10) << value
       << resetiosflags(ios::left) << '\n';
  cout << "Hexadecimal with no preface: " << hex << value << '\n';
  cout << "Hexadecimal with preface: "
       << setiosflags(ios::showbase) << value << '\n';
```

```
  cout << "Uppercase hexadecimal with preface: "
       << setiosflags(ios::uppercase) << value
       << resetiosflags(ios::uppercase) << '\n';
  cout << "Octal (base 8) with preface and field width of 6: "
       << oct << setw(6) << value
       << resetiosflags(ios::showbase) << dec << '\n';
  cout << "Unsigned short decimal integer: "
       << (unsigned)short_value << '\n';
  cout << "Signed long decimal integer: "
       << setiosflags(ios::showpos) << long_value
       << resetiosflags(ios::showpos) << '\n';
  cout << "Floating point, width 10, one place precision: "
       << setiosflags(ios::fixed | ios::showpoint)
       << setprecision(1) << setw(10) << x << '\n';
  cout << "Floating point scientific notation: "
       << resetiosflags(ios::fixed) << setiosflags(ios::scientific)
       << setprecision(0) << pi << '\n';
  cout << "Floating point, precision 10: "
       << resetiosflags(ios::scientific) << setiosflags(ios::fixed)
       << setprecision(10) << pi << '\n';
  return 0;
}
```



```
Plain decimal integer value: 26815
Decimal integer value with forced sign: +26815
Decimal integer -- right justification:      26815
Decimal integer -- zero fill on left: 0000026815
Decimal integer -- left justification: 26815
Hexadecimal with no preface: 68bf
Hexadecimal with preface: 0x68bf
Uppercase hexadecimal with preface: 0X68BF
Octal (base 8) with preface and field width of 6: 064277
Unsigned short decimal integer: 4294967295
Signed long decimal integer: +123456789
Floating point, width 10, one place precision:      -78.5
Floating point scientific notation: 3.141593e+000
Floating point, precision 10: 3.1415926536
```

Other members of istream

`get();`	Extracts a single character and returns it
`get(char * pch, int nCount, char delim = '\n');`	Extracts characters from stream until `delim` is found, the limit `nCount` is reached, or the end of file is reached. The characters are stored in the array `pch` followed by a null terminator. If `delim` is found it is neither extracted nor stored. **Note**: `pch` must point to an array that has space for at least `nCount` characters.
`get(char& rch);`	Extracts a single character from the stream and stores it in `rch`.
`getline(char *pch, int nCount, char delim = '\n');`	Extracts characters from stream until either `delim` is found, the limit `nCount-1` is reached, or the end of file is reached. The characters are stored in the array `pch` followed by a null terminator. If `delim` is found, it is extracted, but not stored. **Note**: `pch` must point to an array that has space for at least `nCount` characters..

`ignore(int nCount=1, int delim = EOF);`	Extracts and discards up to `nCount` characters. Extraction stops if `delim` is extracted or if the end of file is reached.
`int peek();`	Returns the next character without extracting. Returns EOF if at end of file or if an error is detected.
`putback(char ch);`	The character `ch` is put back; must be the character previously extracted. May only be called once after a character is extracted.
`read(char *pch, int nCount);`	Extracts bytes from the stream unit limit `nCount` is reached or until end of file. Used for binary streams. **Note**: `pch` must point to an array that has space for at least `nCount` characters.
`eof()`	Returns true if the last read operation failed because no more data is available.
`operator void*()` `bool operator!()`	An istream object may be used as a Boolean expression. Result is true if the stream is in the good state.

Other members of ostream

`flush();`	Flushes the output buffer.
`put(char ch);`	Inserts the single characher `ch` into the output stream.
`write(const char* pch, int nCount);`	Inserts the specified number of bytes from the array `pch` into the output stream. If in text mode `'\n'` (a single character) may be converted to ‹cr›‹lf› (two characters). This is operating system dependent.
`operator void*()` `bool operator!()`	An ostream may be used in a Boolean expression. Result is true if the stream is in the good state.

ifstream

The class `ifstream` (input file stream) is defined in the header file `<fstream>`. It provides the definition of an `istream` that is associated with an external file.

`ifstream();`	Constructs an ifstream object without opening a file.
`ifstream(const char* szName, int nMode = ios::in);`	Constructs an ifstream object, opening the file given in the character array szName. The default mode is input. The possible additional mode flags are: ios::binary — opens file in binary mode (default is text)
`bool is_open();`	Returns true if the file is open.
`open(const char* szName, int nMode = ios::in, int nProt = filebuf::openprot)`	Opens a file. Same arguments as if the file name was specified in the constructor.
`int setmode(int nMode = filebuf::text);`	Returns the previous mode and sets the mode to nMode. Must be used only on open files.
`close();`	Closes the ifstream.

ofstream

The class `ofstream` (output file stream) is defined in the header file `<fstream>`. It provides the definition of an `ostream` that is associated with an external file.

Same member functions as ifstream. More options for nMode as follows:

`ios::app`	Seek to the end of file. New bytes are appended to the end, even if the position is moved with `ostream::seekp`.
`ios::ate`	Seek to the end of file. The first byte is appended. Subsequent bytes are written to the current position, which may be changed using `ostream::seekp.`
`ios::in`	Stream is used for both input and output.
`ios::out`	Default if ofstream.
`ios::trunc`	If the file already exists, its contents are discarded. This is the default for `ios::out`, unless `ios::in`, `ios::app`, or `ios::ate` are specified.
`ios::binary`	Opens the file in binary mod (default is text).

Detecting errors and eof

There are three flags to indicate the error state:

Flag	Meaning
`badbit`	Indicates a loss of integrity in an input or output sequence (such as an irrecoverable read error from a file).
`eofbit`	Indicates that an input operation reached the end of an input sequence. Only set after an input has been attempted. Reading the last value does not set this bit.
`failbit`	Indicates that an input operation failed to read the expected characters, or that an output operation failed to generate the desired characters. Generally indicates that the input data does not match the expected format.

`operator void*()` will return a non-zero value if none of these flags is set.

`bool operator!()` will return true if any of these flags is set.

Detecting eof or error

If an input fails, the resulting value is undefined, and the stream is frozen. Thus subsequent inputs will also fail – leading to an infinite loop.

Therefore, one should always check to see if an input succeeded.

Examples:

```
while (in >> x) {
// do something with x
// loop will terminate at eof, or if there is an error
}
while (true) {
    if (!(in >> x)) break;
    // do something with x
}
```


Case Study: Preparing a Payroll File

Problem Statement:

Write a program to read a data file consisting of employee salary data. The input consists of a series of input lines containing the employee's first name, last name, hours worked, and hourly rate. An example is as follows:

```
Jim Baxter 35.5 7.25
Adrian Cybriwsky 40.0 6.50
Ayisha Mertens 20.0 8.00
```

Output shall consist of the employee's name on one line followed the employee's gross salary on a separate line. Example corresponding to the previous input is as follows:

```
Jim Baxter
$257.38
Adrian Cybriwsky
$260.00
Ayisha Mertens
$160.00
```

When processing of all employees is completed, the total payroll amount should be displayed.

```
// File: Payroll.cpp.
// Writes each employee's name and gross salary to an
//  output file and computes total payroll amount.

// INCLUDE FILES...
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
using std::cerr;
using std::string;
using std::istream;
using std::ostream;
using std::ifstream;
using std::ofstream;

int main(int argc, char* argv[])
{
  // FUNCTIONS USED...
  // PROCESS ALL EMPLOYEES AND COMPUTE TOTAL
  float process_emp
    (istream&,           // IN: employee file
     ostream&);          // OUT: payroll file

  // LOCAL DATA...
  ifstream ins;          // input: input employee data file
  ofstream outs;         // output: output employee data file
  float payroll;         // output: total payroll

  // Check for proper number of arguments
  if (argc != 3) {
```



```
    cerr << "Usage payroll employee_file payroll_file\n";
    return 1;
  }

  // Prepare in_emp and out_emp files.
  ins.open(argv[1]);
  if (!ins) {
    cerr << "Cannot open " << argv[1] << " for input.\n";
    return 1;
  }
  outs.open(argv[2]);
  if (!outs) {
    cerr << "Cannot open " << argv[2] << " for output.\n";
    ins.close();
    return 1;
  }

  // Process all employees and compute payroll title.
  payroll = process_emp (ins, outs);

  // Display result.
  std::cout << "Total payroll is " << payroll << std::endl;

  // Close files.
  ins.close();
  outs.close();

  return 0;            //Normal return from main
}

// PROCESS ALL EMPLOYEES AND COMPUTE TOTAL
```

```
float process_emp (
          istream& ins,                 // IN: employee file
          ostream& outs)                // OUT: payroll file
  // PRE : inp and out_data are prepared for input/output.
  // POST: All employee data are copies from inp to out_data
  //              and the sum of their salaries is returned.
{

  // LOCAL VARIABLES
  string first_name;       // input: employee first name
  string last_name;        // input: empolyee last name
  float hours;             // input: hours worked.
  float rate;              // input: hourly rate.
  float salary;            // output: gross salary.
  float payroll = 0.0;     // return value - total company payroll

  // Set format flags for output
  outs << std::fixed << std::showpoint << std::setprecision(2);

  // Read and process each employee's record
  while (ins >> first_name >> last_name >> hours >> rate) {
    salary = hours * rate;
    outs << first_name << " " << last_name << '\n';
    outs << salary << '\n';
    payroll += salary;
  } // end while

  return payroll;

}   // end process_emp
```



```
Total payroll is 677.375


Jim Baxter
257.38
Adrian Cybriwsky
260.00
Ayisha Mertens
160.00
```

Revised Problem Statement

The input consists of the employee's full name on one line followed by the hours and rate on a separate line. Employee's full name may include a middle name or middle initials. Example of input:

```
Jim Andrew Baxter
35.5 7.25
Adrian Cybriwsky
40.0 6.50
Ayisha W. Mertens
20.0 8.00
```


Limitation on operator>>

`istream& istream::operator>>(string&);`
Is defined as follows:

1. Skip leading white space characters.
2. Extract characters until a white space character is encountered.
3. Place the extracted characters in the rhs argument.

Thus the statement:

```
ins >> first_name >> last_name;
```
when applied to the input line

```
Jim Andrew Baxter
```
will result in

`first_name : Jim`

`last_name : Andrew`

and leave the input stream ready to read Baxter. When the statement

`ins >> hours >> rate;`

is executed the value of `hours` and `rate` will be left undefined and the input stream will be frozen in an error state.

If the statement:

```
ins >> first_name >> middle_name >> last_name;
```
when applied to the input line

```
Jim Andrew Baxter
```
will result in

```
first_name : Jim
middle_name : Andrew
last_name : Baxter
```
and leave the input stream ready to read the hours and rate. When the statement

```
ins >> hours >> rate;
```
is executed the value of `hours` and `rate` will be read correctly.

However, if the statement

Thus the statement:

```
ins >> first_name >> middle_name >> last_name;
```
when applied to the input lines

```
Adryian Cybriwsky
40.0 6.50
```
will result in

```
first_name : Adryian
middle_name : Cybriwsky
last_name : 40.0
```
and leave the input stream ready to read the hours. When the statement

```
ins >> hours >> rate;
```
is executed the value of `hours` will get the value `6.5` and `rate` will be left undefined and the input stream will be frozen in an error state.

The function `getline`

The following function is defined in the <string> header file as a friend of the string class:

```
void getline(istream& is, string& str, char delim = '\n');*
```

Which extracts characters from `is` and places them in `str` until `delim` is encountered. The delimiter character `delim` is extracted but not placed in `str`.

Thus,

```
getline(ins, full_name, '\n');
```

will read either

```
Jim Andrew Baxter
```

or

```
Adrian Cybriwsky
```

into `full_name` and leave `ins` ready to read the `hours` and `rate`.

* This function does not work for cin on Microsoft Visual C++ version 6.

Need to Skip Leading White Space

The statement

```
ins >> hours >> rate;
```

Leaves `ins` ready to read the newline character that follows the rate.

Thus when the statement

```
getline(ins, full_name, '\n');
```

is next encountered, `full_name` will be set to blank and the input stream will now be ready to read the next name. However, the program will attempt to read the name into `hours` and `rate` resulting in the input being frozen in an error state.

The statement

```
ins >> ws;
```

will skip white space and leave the input stream ready to read the next non-white space character.

float vs. int

Note that the program produced the output:

```
Total payroll is 677.375


Jim Baxter
257.38
Adrian Cybriwsky
260.00
Ayisha Mertens
160.00
```

Jim Baxter worked 35.5 hours at the rate of $7.25 per hour. Thus, his total is $257.375. When output, this is correctly rounded to 275.38, but the total is output as 677.375. Had there been others with fractional cents, the total would not be the sum, even if rouned.

To avoid this, the salaries should be computed in cents as long int, and then converted back to float for output.

Revised version of `process_emp`

```
// PROCESS ALL EMPLOYEES AND COMPUTE TOTAL
float process_emp (
    istream& ins,                  // IN: employee file
    ostream& outs)                 // OUT: payroll file

// PRE :   ins and outs are prepared for input/output.
// POST:   All employee data are copies from ins to outs
//         and the sum of their salaries is returned.
{
    // LOCAL VARIABLES
    string name;           // input: employee first name
    float hours;           // input: hours worked.
    float rate;            // input: hourly rate.
    long salary;           // output: gross salary.
    long payroll = 0;      // return value - total company payroll
    // Set format flags for output
    outs << std::fixed << std::showpoint << std::setprecision(2);
    // Process each employee's record
    while (true) {
    ins >> std::ws;
    std::getline(ins, name);
    if (!(ins >> hours >> rate)) break;
    salary = long(hours * rate * 100.0 + 0.5);
    outs << name << '\n';
    outs << salary/100.0 << '\n';
    payroll += salary;
    }  // end while
    return payroll/100.0;
}   // end process_emp
```

Array Declaration

An array is a collection of objects having the same data type.

Form: *element-type array-name* [*dimension*]

Example: `char my_name [5]`

Interpretation: The identifier *array-name* describes a collection of array elements each of which may be used to store an object of type *element-type*. The *dimension*, enclosed in brackets, `[]`, specifies the number of elements contained in the array. The dimension value must be a constant expression; that is, it must be possible to compute the value of the expression at compile time. This value must be an integer and must be greater than or equal to one. There is one array element for each value between 0 and the value *dimension* - 1, and all elements of an array are the same type, *element-type*, which may be one of the fundamental C++ types or a user-defined type.

Array Access

‹postfix expression› ::= ‹postfix expression›[‹expression›]

In general ‹postfix expression› is either an array name or a pointer type, and ‹expression› is an integer (or converted to an integer).

Form: *name* [*subscript*]

Example: `x[3 * i - 2]`

Interpretation: The *subscript* must be an expression with an integral value. If the expression value is not in range between 0 and the dimension of `x -1` (inclusive), a memory location outside the array will be referenced. If referenced in an expression, the value is unpredictable. If referenced on the left-hand-side of an assignment, some other variable, or even the program code, may be modified unexpectedly.

EXAMPLE

```
float x[8]
```

declares an array x to contain 8 floating point numbers.

Array x

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5
first element	second element	third element		...			eighth element

Statement	Explanation
`cout << x[0];`	Displays the value of `x[0]` or 16.0.
`x[3] = 25.0;`	Stores the value 25.0 in `x[3]`.
`sum = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]` or 28.0 in the variable `sum`.
`sum = sum + x[2];`	Adds `x[2]` to sum. The new sum is 34.0.
`x[3] = x[3] + 1.0;`	Adds 1.0 to `x[3]`. The new `x[3]` is 26.0.
`x[2] = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]` in `x[2]`. The new `x[2]` is 28.0.

Array x

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	28.0	26.0	2.5	12.0	14.0	-54.5
first element	second element	third element		...			eighth element

Statement	Effect
`cout << 3 << x[3];`	Displays 3 and 26.0 (value of `x[3]`).
`cout << i << x[i];`	Displays 5 and 12.0 (value of `x[5]`).
`cout << x[i] + 1;`	Displays 13.0 (value of 12.0 + 1).
`cout << x[i] + i;`	Displays 17.0 (value of 12.0 + 5).
`cout << x[i+1];`	Displays 14.0 (value of `x[6]`).
`cout << x[i+i];`	Illegal attempt to display `x[10]`.
`cout << x[2*i];`	Illegal attempt to display `x[10]`.
`cout << x[2*i-3];`	Displays -54.5 (value of `x[7]`).
`cout << x[floor(x[4])];`	Displays 6.0 (value of `x[2]`).
`x[i] = x[i+1];`	Assigns 14.0 (value of `x[6]`) to `x[5]`.
`x[i-1] = x[i];`	Assigns 14.0 (new value of `x[5]`) to `x[4]`.
`x[i] - 1 = x[i-1];`	Illegal assignment statement.


```
// FILE: ShowDiff.cpp
// Computes the average value of an array of data and prints
// the difference between each value and the average.

#include <iostream>
#include <iomanip>
using namespace std;
const int max_items = 8;
float x[max_items];      //array of data
int i;               //loop-control variable
float average,               //average value of data
   sum;          //sum of the data

int main()
{
  // Set output format for float.
  cout << setiosflags(ios::fixed | ios::showpoint);
  // Enter the data
  cout << "Enter " << max_items << " numbers: ";
  for (i = 0; i < max_items; i++)
    cin >> x[i];
```

```
  // Compute the average value
  sum = 0.0;         //initialize sum
  for (i = 0; i < max_items; i++)
    sum += x[i];     //add each element to sum
  average = sum / max_items;//get average value
  cout << "The average value is "
    << setprecision(1) << setw(3) << average << "\n\n";

  // Display the difference between each item and the average
  cout << "Table of differences between x[i] and the average.\n";
  cout << setw(4) << "i" << setw(8) << "x[i]" << setw(14)
    << "difference" << '\n';
  for (i = 0; i < max_items; i++)
    cout << setw(4) << i << setw(8) << x[i] << setw(14)
      << (x[i] - average) << '\n';
  return 0;
}
```



```
Enter 8 numbers: 16.0 12.0 6.0 8.0 2.5 12.0 14.0 -54.5
The average value is 2.0

Table of differences between x[i] and the average.
   i    x[i]    difference
   0    16.0          14.0
   1    12.0          10.0
   2     6.0           4.0
   3     8.0           6.0
   4     2.5           0.5
   5    12.0          10.0
   6    14.0          12.0
   7   -54.5         -56.5
```

Pointers

A pointer is an object that can be used to access another object. A pointer provides indirect access rather than direct access.

Real life examples:

- If someone asks for directions, but you do not know the answer, you may reply "Go to the gas station and ask them."
- If someone asks for a phone number, but you do not know the answer, you may reply "Let me look it up in the phone book."
- A professor says "Do problem 1.1 in the textbook." This is an indirect address of the problem.

Pointer Type Declaration

Form: ‹type› *‹variable›;

Example: `float *px;`

Interpretation: Pointer variable `px` is of a data type whose values may be thought of as memory cell addresses. A data variable whose address is stored in this variable must be type ‹type›.

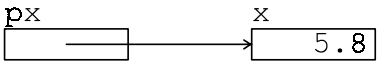
Pointer-to Operator

Form: The prefix operator & will generate a pointer to its operand.

Example:

```
float x = 5.8;
float *px = &x;
```

results in the following:



De-Referencing Operator

Form: The prefix operator * when applied to a pointer will evaluate to the pointed-to object.

Example:

```
cout << *px;
```

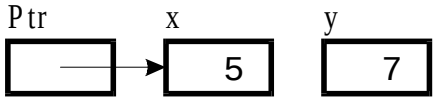
will result in the output `5.8`.

The value of a pointer

Pointers should be thought of as abstract objects. However, for ease of understanding, one can think of them a containing the memory address of the first addressable memory unit containing the object pointed to.

Example:

Variable (object)	Address	Contents
X	1000	5
Y	1004	7
⋮	⋮	⋮
Ptr	1200	1000



References vs. Pointers

A reference is an alias (i.e., alternate name) for an object. Once it is declared, it cannot be changed. Under some circumstances a reference is a constant pointer what is automatically de-referenced. Under other circumstances, it is merely a notation convenience and has no representation in your program.

A pointer is an object in its own right, and its value can be changed.

Example:

```
float x = 5.8;
float y = 7.9;
float z = 3.5;
float *px = &x;  // px now points to x
float &ry = y;   // ry refers to y
float *pz = &z;  // pz now points to z
pz = px;         // pz now points to x
ry = px;         // ILLEGAL
ry = x;          // the value of y is now 5.8
*px = z;         // the value of x is now 3.5
float* py = &y;   // py points to y
float* pry = &ry; // pry == py it does not point to ry
```


The null pointer

There is a special pointer that points to nothing. This is called the *null* pointer. The exact bit pattern is implementation defined.

The integer constant `0` when used as a pointer is converted to the null pointer.

When used in a Boolean expression, the null pointer is converted to `false` and all other pointers are converted to `true`.

The header file `<cstdlib>` contains the macro definition `NULL` that is defined to be `0`. However, its use is deprecated.

Examples:

```
px = 0;   // px is the null pointer
if (px == 0) // test is true if px is null
if (px == NULL)  // test is true if px is null
if (!px)  // test is true if px is null
if (px != 0) // test is true if px is not null
if (px != NULL)  // test is true if px is not null
if (px)   // test is true if px is not null
```

`new` operator

Form: `new` ‹type›

Example: `new float`

Interpretation: Storage for a new data variable is allocated from a pool of storage known as the *heap*, and the address of this data variable is result of the operator. The internal representation and size of the new data variable is determined from the declaration of the type (or is known by the compiler for built-in types). If storage is not available, an exception is raised.

`new[]` operator

Form: `new` ‹type›`[`‹size›`]`

Example: `new float[10]`

Interpretation: Storage for a new data array is allocated from a pool of storage known as the *heap*, and the address of the first data variable is result of the operator. The internal representation and size of the new data variable is determined from the declaration of the type (or is known by the compiler for built-in types). If storage is not available, an exception is raised.

The “Equivalence” of Arrays and Pointers

- The `operator[]` is defined for pointers in terms of `operator*` as follows:

  `a[i]` ≡ `*(a+i)`

- Thus, the results of the `new[]` operator can be used as if it was an array.

Limitations of arrays

- An array is of fixed size. You must specify the size of the array as a constant when the program is compiled, and the size cannot be changed.
- There is no checking done during program execution that the index in an array access is valid.
- An array cannot be copied using a simple assignment operation.
- Arrays are passed to functions as constant pointers to their first element.

The vector class

C++ is an extensible language. Programmers can extend the language by defining new types, called classes that behave almost the same as the built-in types. There is an extensive library of classes defined in the standard. To allow users to only get those portions of the library they want, individual components are grouped and defined in various headers. The objects `cin` and `cout` are defined in the `iostream` header as objects of the classes `istream` and `ostream` respectively.

The standard library contains the header `vector` that defines the class vector with all of the features of an array, but without the limitations.

Vector vs. Array

Feature	array	vector
Capacity	Fixed at declaration	Grows as necessary
Size	Must be known by the user	Provides current size function `size()`
Random access without bounds checking	via `operator[]`	via `operator[]`
Random access with bounds checking	not available	via function `at()`
Assignment	via function `memcpy`	via `operator=`.
Change the size	Not Available.	via function `resize()`.

Changing size and capacity

The size of a container (such as an array or vector) represents the number of objects stored in the container. The capacity of a container represents the maximum value of size without a re-allocation. For the array, size equals capacity and neither can be changed. For the vector, size can be changed and the capacity grows automatically as required.

Operations that change size:

```
name.resize(new size);      // The size of the vector name is changed to
                            //  be the new size. If the old size was smaller,
                            // new elements of a default value are inserted
                            // at the end. If the old size was larger,
                            // elements are removed from the end.
```

The function `size()` returns the current size.

Declaration of a vector

The following shows how a vector is declared:

```
#include <vector>
#using std::vector;
⋮
vector<type> v;                 // declare an empty vector
vector<type> v(initial size);   // declare a vector of a given
                                // size, all values initialized
                                // to default.
vector<type> v(initial size,  initial value);
```

Inserting Data at the End of a Vector

The vector class has the member `push_back(T v)` that inserts the value v at the end. It is equivalent to the following:

```
void vector<T>::push_back(T v) {
   resize(size()+1);
   operator[](size()) = v;
}
```


Revised ShowDiff

```
// FILE: ShowDiff2.cpp
// Computes the average value of an array of data and prints
// the difference between each value and the average.

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;
const int max_items = 8;
vector<float> x;   //array of data
int i;          //loop-control variable
float average;     //average value of data
float sum;     //sum of the data

int main(int argc, char* argv[])
{
  if (argc < 2) {
    cerr << "Usage ShowDiff2 <input file>";
    return 1;   // Error return from main
  }
  ifstream in(argv[1]);  //declare and open input stream
  if (!in) {
    cerr << "Unable to open " << argv[1] << " for input.";
    return 1;
  }
  // Set output format for float.
  cout << setiosflags(ios::fixed | ios::showpoint);
```

```
  // Enter the data
  float v;
  while (in >> v) {
    x.push_back(v);
  }
  // Compute the average value
  sum = 0.0;           //initialize sum
  for (i = 0; i < x.size(); i++)
    sum += x[i];       //add each element to sum
  average = sum / x.size();   //get average value
  cout << "The average value is "
       << setprecision(1) << setw(3) << average << "\n\n";

  // Display the difference between each item and the average
  cout << "Table of differences between x[i] and the average.\n";
  cout << setw(4) << "i" << setw(8) << "x[i]" << setw(14)
       << "difference" << '\n';
  for (i = 0; i < x.size(); i++)
    cout << setw(4) << i << setw(8) << x[i] << setw(14)
     << (x[i] - average) << '\n';
  return 0;
}
```


Strings

- An array of char is known as a string.
- A string's capacity is the size of the array.
- A string's length is the number meaningful characters.
- Immediately following the last meaningful character is the value zero.
- Like the arrays that they are, strings cannot be directly operated upon.
- As part of the C programming language, the library contains several functions to operate upon strings:

Function name	Purpose
`strcpy`	Copy one string to another. Note the destination must have the capacity required. There is no run-time check.
`strcat`	Concatenate one string onto the end of another. Note that the destination must have the capacity required. There is no run-time check.
`strcmp`	Compare two strings. Result returned is > 0 if the left hand operand is > the right hand operand, < 0 if the left hand operand is < the right hand operand, and ==0 if they are equal.
`strlen`	Determine the length of a string. Note this is done by counting the characters up to the zero.

String Literals

The language grammar defines the following:

‹s-char› ::= ‹any member of the source character set except " \ or newline› | ‹escape sequence›

‹escape sequence› ::= `\' | \" | \? | \\ | \a | \b | \f | \n | \r | \t | \v | \`‹up to 3 octal digits›
`\x`‹sequence of hexadecimal digits›

‹string literal› ::= `"`‹zero or more s-chars›`"`

In response to a string literal the compiler creates a zero-terminated array of characters.

String Initialization

A string variable can be initialized as follows:

`char` *name*`[`*capacity*`] =` *a string literal*`;`

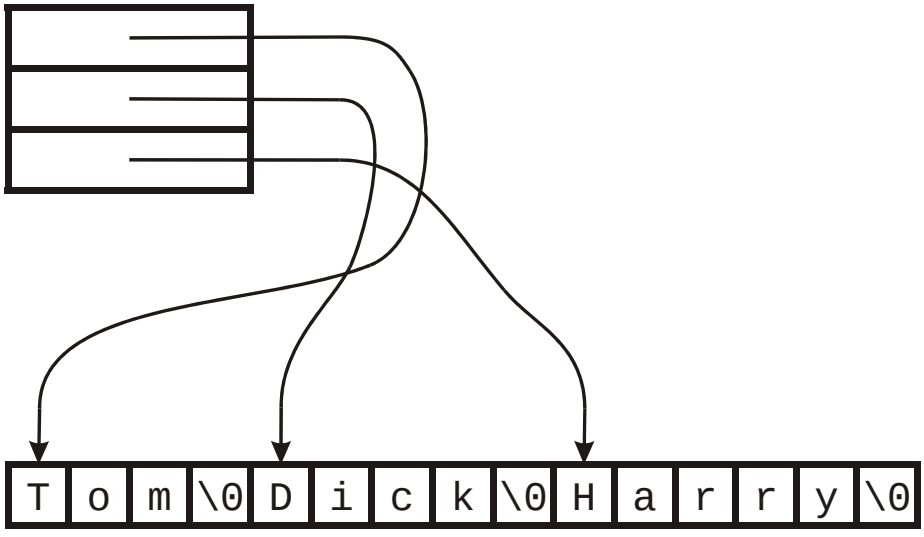
`char` *name*`[] =` *a string literal*`;`

If specified the capacity must be at least one larger than the length of the string literal.

Arrays of Strings

An array of strings is represented as an array of pointers as follows:

```
char* names[] = {"Tom", "Dick", "Harry"};
```



String Class

The string class, defined in the header `<string>` is similar to a `vector<char>` except that additional operations have been defined for it.

Objects of the string class are declared as follows:

```
#include <string>
using std::string;
⋮
string name;  // declare an empty string
string name(string literal);
string name = string literal;
```


C String vs C++ String Class

Feature	C String	C++ String Class
Capacity	Fixed at declaration	Grows as necessary
Size	Indicated by `'\0'` char at end. Found by `strlen`, which counts the characters.	Provides current size function `size()`
Random access without bounds checking	via `operator[]`	via `operator[]`
Random access with bounds checking	not available	via function `at()`
Assignment	`strcpy`	like any other object via =
Comparison	`strcmp`	like any other object via <, >, ...
Concatenation	`strcat`	`operator+` and +=

The 12 Days of Christmas

```
// FILE: 12days.cpp
#include <iostream>
#include <string>
using std::string;
void sing_song(std::ostream& voice) {
string day[] = {"first", "second", "third", "fourth",
                "fifth", "sixth", "seventh", "eighth",
                "ninth", "tenth", "eleventh", "twelfth" };
  for (int i = 0; i < 12; i++) {
    voice << "On the " << day[i]
        << " day of Christmas" << endl;
    voice << "My true love gave to me" << endl;
    switch (i+1) {
    case 12: voice << "Twelve lords a-leaping;" << endl;
    case 11: voice << "Eleven ladies dancing;" << endl;
    case 10: voice << "Ten pipers piping;" << endl;
    case  9: voice << "Nine drummers drumming;\n";
    case  8: voice << "Eight maids a-milking;" << endl;
    case  7: voice << "Seven swans a-swimming;" << endl;
    case  6: voice << "Six geese a-laying; << endl;
    case  5: voice << "Five golden rings; << endl;
    case  4: voice << "Four calling birds; << endl";
    case  3: voice << "Three French hens; << endl;
    case  2: voice << "Two turtle doves, and" << endl;
    case  1: voice << "A partridge in a pear tree" << endl << endl;
    }
  }
}
```



```
// FILE: P12DAYS.CPP
#include <iostream>
using std::ostream;
void sing_song(ostream&);

int main ()
{
  sing_song(std::cout);
  return 0;
}

// FILE: S12DAYS.CPP
#include <iostream>
using std::ostream;
#include "voice.h" // Header file defining vostream
void sing_song(ostream&);

int main()
{
  vostream speaker;// vostream is a ostream that sends
              // its output to the speaker.
  sing_song(speaker);
  return 0;
}
```

Note on namespace

- A namespace is a block that hides all of the definitions defined within it.
- Names defined in a namespace can be accessed by prefixing the name with the namespace name followed by the symbol `::`.
- A name defined within a namespace can be made visible by the using declaration as follows:

  `using` *namespace*`::`*name*`;`
- All of the names defined within a namespace can be made visible by using the declaration:

  `using namespace` *namespace*`;`
- All of the names defined within the standard library are defined within the namespace `std`.
- Namespaces are relatively new, and not all compilers (still) fully support them.

using namespace std;

- Namespaces, especially namespace `std`, sometimes can seem to be a nuisance. Thus there is the temptation and a fairly common practice to include the following statement early in the program:

  ```
  using namespace std;
  ```
- This is not recommended, especially at the global level.
  1. The standard library may contain names that inadvertently conflict with names defined in the program. (This is the reason for namespace in the first place.)
  2. For certain compilers (Microsoft Visual C++, version 6 in particular) this can cause erroneous error messages. Note, your program will not run if the compiler, however mistaken, does not consider it to be valid.

Implementation of the Vector Class

```
// FILE: vector.h
// Declaration and definiton of template vector class
// This is a very simplified subset of the standard library
#ifndef vector_h_
#define vector_h_

template <class T>
class vector
{
public:
    vector() : buffer(0) { resize(0); }
    vector(unsigned int size) : buffer(0) {resize(size);}
    vector(unsigned int size, T initial);
    vector(const vector& v);
    ~vector() {delete [] buffer;}
    T back() {return buffer[mySize -1]};
    T front() {return buffer[0];}
    bool empty() {return mySize == 0;}
    void pop_back() {mySize--;}
    void push_back(T value)
    { resize(mySize + 1); buffer[mySize-1] = value;}
    void reserve(unsigned int newCapacity);
    void resize(unsigned int newSize)
    {reserve(newSize); mySize = newSize;}
    int size() {return mySize;}
```



```
    T& operator[] (unsigned int index) {return buffer[index];}
    T& at(unsigned int index)
    {
        if (index < 0 || index >= mySize)
            throw "Out of range";
        return buffer[index];
    }
protected:
    unsigned int mySize;
    unsigned int myCapacity;
    T* buffer;
};


template<class T>
vector<T>::vector(unsigned int size, T initial)
{
    resize(size);
    for (unsigned int i = 0; i < size; i++)
        buffer[i] = initial;
}

template<class T>
vector<T>::vector(const vector<T>& v)
{
    resize(v.size());
    for (unsigned int i = 0; i < mySize; i++)
        buffer[i] = v.buffer[i];
}
```

```
template<class T>
void vector<T>::reserve(unsigned int newCapacity)
{
    if (buffer == 0)
    {
        mySize = 0;
        myCapacity = 0;
    }
    if (newCapacity <= myCapacity) return;
    unsigned int nc = newCapacity <= 2*myCapacity ?
        2*myCapacity : newCapacity;
    T* newBuffer = new T[nc];
    for (unsigned int i = 0; i < mySize; i++)
        newBuffer[i] = buffer[i];
    myCapacity = nc;
    delete[] buffer;
    buffer = newBuffer;
}

#endif
```


Assignment 2
(due 4 March 02)

Write a program that prints payroll checks based using the file produced by the payroll program described in class. There are two lines for each employee. The first contains the employee name and the second the amount. The date of the check should be the current date. The first check number should be a random number, and subsequent checks should be sequentially numbered. Your program should take two command line arguments: the first is the input file (output from the payroll program) and the second is the file containing the checks. The format of the checks should be the same as the one shown below:

```
-------------------------------------------------------
|Temple University                Check No.    12372    |
|Philadelphia, PA                 Date:   10-31-2001    |
|                                                       |
|Pay to the                                             |
|Order of:  William Cosby              ***$20,000.00    |
|                                                       |
|                                      _____________    |
|                                                       |
-------------------------------------------------------
```

The following program will obtain the current date:

```
#include <iostream>
using std::cout;
using std::endl;
#include <iomanip>
using std::setfill;
using std::setw;
using std::setiosflags;
#include <ctime>
#ifndef _MSC_VER
using std::time_t;
using std::time;
using std::tm;
using std::localtime;
#endif

int main()
{
  time_t msec_since_19700101 = time(0);
  tm* the_time = localtime(&msec_since_19700101);
  cout << setfill('0') << setiosflags(std::ios::right);
  cout << "The date is " << std::setw(2) << the_time->tm_mon+1
       << "-" << setw(2) << the_time->tm_mday
       << "-" << setw(4) << the_time->tm_year+1900
       << endl;
  return 0;
}
```