C Programming for Engineers

A Review of C. Introducing iostream

Nick Urbanik `nicku@nicku.org`

2006 February

Outline

Contents

1 Who am I? Who are You?

Welcome!

- Welcome to our class in Electrical Control, C Programming
- My name is Nick Urbanik
  - Call me Nick
- My email address is `nicku@nicku.org`
- The notes for this class are always available at `http://nicku.org/c-for-engineers/`
- I taught in Hong Kong for ten years
- I taught in Macau one year before that
- Before that I wrote software in C++ to collect data for medical experiments

How we will work in these classes

- We will learn by doing
  - I will explain something for a short time
  - you will then try it out for a short time
- Next week:
  - I will print the notes out for you, now that I know what you need
  - I will give you more information about this course
- This is always available from `http://nicku.org/`; click on the link on the left "C for Engineers".
- I will publish all the teaching material also on `http://gonzo.org.au/`, and I'll explain to you how to use it

2 Overview

2.1 Quick Tour

C — Quick tour

- C was originally designed as a low-level systems programming language for an early version of the UNIX operating system.
- Combine:
  - efficiency and hardware access capability of assembler with
  - high level language structures and
  - portability
- Most of the Linux and UNIX operating systems are written in C.

C is portable

- Although C matches the capabilities of many computers, it is independent of any particular machine architecture.
- it is not difficult to write portable programs that can be run without change on a variety of hardware
- the standard makes portability issues explicit, and prescribes a set of constants that characterize the machine on which the program is run.
  - See `limits.h`

C — quick tour (continued)

- The standard library is a suite of ready-written commonly used functions
- header files declare the library functions, and any symbolic constants required.
- The (obsolete) header file for the `iostream` library is *included* into your program file with:

  #include <*iostream*>

- Header files tell the compiler about the library functions and about the type of the parameters and return types
- linker automatically links in the required functions at compile-time.
- C programs make use of functions from the standard library; e.g., all input and output operations are usually performed using standard library functions.

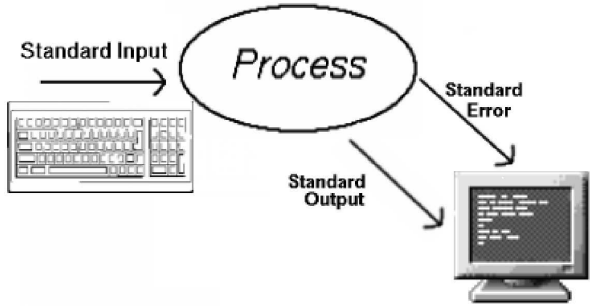
2.2 Standard Input, Standard Output

Standard Input, Output

- If you run a program in Windows or Linux, it usually has three files already open:

Name	*File Descriptor*	*normally connected to*
Standard Input	0	keyboard
Standard output	1	screen
Standard error	2	screen

- These are normally connected to the keyboard and your command prompt window



2.3 Redirecting Output and Input

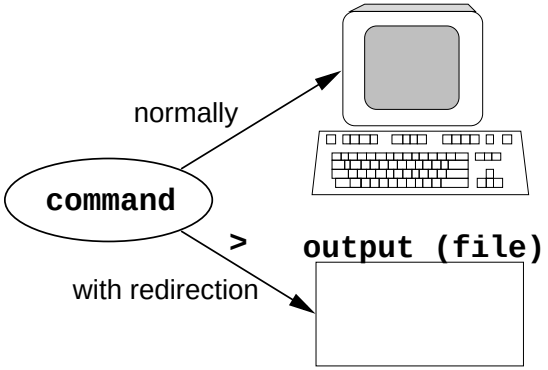
Output Redirection

- We redirect output using '>'
- For example:

  `C:\STUDENT>` **`command > output`**

  Creates the file `output` (or overwrites it if it already exists) and places the standard output from `command` into it

- We can append to a file rather than overwriting it by using >>

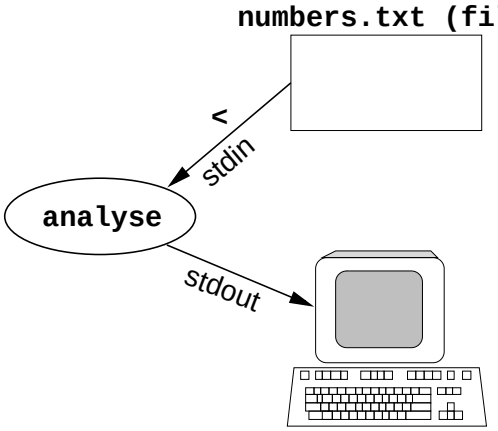


Input Redirection

- < redirects standard input from a file, e.g.,

  `C:\STUDENT>` **`analyse < numbers.txt`**

- `analyse` now take the contents of the file `numbers.txt` as its input



3 C++ and `iostream`library

C++

- C++ is forwards compatible with C
- If a C++ compiler cannot compile a C program, the C program may be poorly written
- We will use some basic C++ features in this course, where they make things easier
- One of these ways is file input and output using the `iostream` standard library.
- Another is the standard `string` library, but I don't think that Borland C++ 3.1 supports it.

3.1 An overview of the `iostream`library

Overview of `iostream`library

- To use `iostream` library in programs, we must include the header file like this:

```
#include <iostream>
```

- If you are using a very old C++ compiler, put this instead:

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

- The library defines three standard stream objects:

`std::cin` pronounced *see-in*
  – an `istream` class object representing *standard input*

`std::cout` pronounced *see-out*
  – an `ostream` class object representing *standard output*

`std::cerr` pronounced *see-err*
  – an `ostream` class object representing *standard error*

Input and Output operators

- Output is done using the left shift operator ≪
- Input is done using the right shift operator ≫

```
#include <iostream>

int main( void )
{
        char name[ 1000 ];
        std::cout << "what is your name? ";
        std::cin >> name;
        if ( name[ 0 ] == '\0' )
                std::cerr << "error: name is empty!\n";
        else
                std::cout << "hello, " << name << "!\n";
}
```

How do I remember?

- A way of remembering which operator is which:
- each operator points in the direction the data moves, e.g.,

  ≫ *x*

  puts data *into* *x*, while

  ≪ *x*

  gets data *out from* *x*

Why is this better than *printf*() and *scanf*()?

- *std*::*cin*, *std*::*cout* and *std*::*cerr* "know" the type of data they are working with, so
- there is no need for you to remember which format string is needed for which data type;
- "it all just works"

4 Continuing our tour

4.1 hello.cpp

First example

- Program `hello.cpp`:

```
// A first program in C++
#include <iostream>
int main( void )
{
    std::cout << "Hello World\n";
}
```

- Output for Program `hello.cpp`:

```
Hello World
```

Good programming practice

- Indentation can improves clarity and readability. It can be enhanced by placing braces or blank lines.
- Use variable and function names that explain themselves
- Functions should be shorter than one A4 page and should be simple to understand
- Every long or complicated function should be preceded by a comment describing the purpose of the function.
- Aim: make the program as easy for a human to understand as possible
- Saves money: less time to change/update program = less money.

4.2 Basic Syntax

Format of *main*() function without parameters

```
int main( void )
{
        ⟨declarations⟩;
        ⟨statements⟩;
}
```


Variable declarations

- Variables are declared and defined with a *data type* and a *name*.
- The name is also called an *identifier*.
- First character of a variable must be letter or underscore (_). Special characters (e.g., $ and #) are illegal.
- Some people (including me!) recommend using lowercase letters and underscores only.

4.3 Input characters without skipping whitespace

Input characters without skipping whitespace

- We can input and output characters one at a time:
- Program `one-char-io.cpp`:

```
#include <iostream>
int main( void )
{
    char letter;
    std::cin >> letter;
    std::cout << letter;
}
```

- Note that by default, *std*::*cin* ≫ *letter*; will skip over whitespace, such as spaces, tabs and newlines
- You can use the `iostream` *member function* *std*::*cin.get*() to input characters one at a time, including whitespace:

```
#include <iostream>
int main( void )
{
    char letter;
    std::cin.get( letter );
    std::cout << letter;
}
```

Data Types

- Three of the most commonly used data types:

`int`	integer
`double`	floating point number
`char`	character

- Program `vars.cpp`:

```
#include <iostream>
int main( void )
{
        int num = 5;
        float cost = 9.5;
        std::cout << "Hello: num = " << num
             << " cost = " << cost
             << '\n';
}
```

The *std*::*endl* manipulator

- The `iostream` libraries support *manipulators*
- A manipulator changes the state of a stream
- The *std*::*endl* manipulator:
  - prints a newline `'\n'`, and
  - sends any remaining characters stored ready for output, to the output
    - We say that this "flushes the buffer"

4.4 I/O of other data to standard output and from standard input

Input and output with `iostream`

- Simple input and output operations using `iostream` library objects *std*::*cout* and *std*::*cin*
- Program `celcius.cpp`:

```
#include <iostream>
int main( void )
{
        double fahr = 212, cel;
        cel = ( 5.0 / 9.0 ) * ( fahr - 32 );
        std::cout << fahr << " deg F => "
                  << cel << " deg C\n";
}
```


Output of program `celcius.cpp`:

```
212 deg F => 100 deg C
```

Using *std*::*cin* for input

- *std*::*cin* object performs the opposite operation.
- reads text from *standard input* and assigns to variables
- automatically converts the input text data to appropriate types
- Program `cin-cout-2.cpp`:

```
#include <iostream>
int main( void )
{
    int num;
    float cost;
    std::cout << "Enter number: ";
    std::cin >> num;
    std::cout << "Enter cost: ";
    std::cin >> cost;
    std::cout << "Num = " << num << ", cost = "
              << cost << '\n';
}
```

4.5 Reading into strings

Strings — array of characters

- A *string* is an array of characters, i.e., text.
- The length of the string can be defined with a number enclosed in brackets. e.g., array of 10 characters with name *lname*:

```
char lname[ 10 ];
```

- Program `cin-cout-3.cpp`:

```
#include <iostream>
int main( void )
{
    int num;
    char lname[ 1000 ];
    std::cout << "Enter class number: ";
    std::cin >> num;
    std::cout << "Enter last name: ";
    std::cin >> lname;
    std::cout << "class number is " << num
              << ", lastname is " << lname
              << '\n';
}
```

4.6 Loops

Loops

- A loop will cause statements to be executed repeatedly until a test condition proves false
- Program `while-1.cpp`:

```
#include <iostream>
int main( void )
{
        int j = 0;
        while ( j < 5 ) {
            std::cout << "j has the value " << j << '\n';
            j = j + 1;
        }
}
```

4.7 Reading till the end of file

End of file

- When there is no more input, a program has reached the *end of file*
- When reading standard input:
  - that is redirected from a file (see section 2.3 on page 6), the program has reached end of file when, well, the last line is read.
  - from the keyboard, you can type Control-Z on Windows, or Control-d on Linux to tell your program that it has reached the end of file

Reading till the end of file

- If you execute the code *std*::*cin* >> *name*; after reaching end of file, the result is *false*.
- That means you can write code like this:

```
if ( std::cin >> name ) {
        // do something with name
}
```

- You can also write a loop that automatically terminates when there is no more to read:

```
float num;
while ( std::cin >> num ) {
        // Now we know that num is a valid float,
        // so do something with it
}
```

  - After this loop is finished, either we have reached end of file, or the next input is not a valid **float**

4.8 Infinite Loop while reading

Infinite Loop while reading

- If you do something like this:

```
float num = 1.0;
std::cout << "enter positive floats: ";
while ( num > 0 ) {
        std::cin >> num;
        std::cout << "You gave me " << num << '\n';
}
```

  if you enter something that is not a valid **float**, you get an infinite loop.

- Use the method I described in the last slide.
- See `http://www.parashift.com/c++-faq-lite/input-output.html#fac` for more about this.

5 Data Types

Data Types

- Data is represented in memory by a sequence of bits, arranged into bytes, eight bits to a byte
- These bits could represent
  - strings or characters
  - integers
  - floating point values
  - memory addresses
  - binary values representing music or video
  - ...
- Each item of data has a *data type*
- The *data type* determines how the program will interpret the data

Declaration & data types

- A variable declaration consists of a data type and an identifier, followed by a semicolon:

  ⟨*Data Type*⟩ ⟨*Variable Name*⟩ ⟨*semicolon*⟩

- Example:

```
int count;
```

Basic data types	Keywords
character	**char**
signed character	**signed char**
unsigned character	**unsigned char**
integer	**int**
unsigned integer	**int**
long integer	**long**
unsigned long integer	**unsigned long**
long long integer	**long long**
unsigned long long integer	**unsigned long long**
short integer	**short**
floating point	**float** or **double** or **long double**

Integer and floating point data types

- Types of integer data (each has an *unsigned* counterpart):
  - character
  - integer
  - short integer
  - long integer
  - long long integer
- Types of floating point values:
  - single precision (**float**)
  - double precision (**double**)
  - long double precision (**long double**)
- The range of values that can be held are *machine dependent*.
- The ranges of integer types are in `limits.h`
- The ranges of floating types are in `float.h`

5.1 Integer Types

Signed integers — 1

- An unsigned integer has only positive values while a signed integer can have a positive or negative value.
- Signed Integers:

int	**use:** standard integer
size:	system dependent, usually size of a word.
range:	*INT_MIN* to *INT_MAX*, defined in `limits.h`. For 4 byte word, $-2^{31}-1$ to 2^{31}, i.e., -2147483648 to 2147483647
example declaration:	**int** *num*;
example constant:	1000

Signed integers — 2

long	**use:** large numbers
size:	usually 4 bytes.
range:	*LONG_MIN* to *LONG_MAX*. For 4 bytes, -2^{31} to $2^{31}-1$, i.e., -2147483648 to 2147483647
example declaration:	*long lnum*;
example constant:	5212000L

short	**use:** smaller numbers
size:	2 bytes or same size as integer
range:	*SHRT_MIN* to *SHRT_MAX*. For 2 bytes, -2^{15} to $2^{15}-1$, i.e., -32768 to 32767
declaration:	*short snum*;
constants:	120

long long

long long	**use:** very large numbers
size:	usually 8 bytes.
range:	for 8 bytes, -2^{63} to $2^{63}-1$, i.e., $-9{,}223{,}372{,}036{,}854{,}775{,}808$ to 9,223,372,036,854,775,807
example declaration:	**long long** *lnum*;
example constant:	5212000LL

unsigned long long	**use:** very large positive numbers
size:	usually 8 bytes.
range:	for 8 bytes, 0 to $2^{64}-1$, i.e., 0 to 18,446,744,073,709,551,615
example declaration:	**unsigned long long** *lnum*;
example constant:	5212000LL

Unsigned integers — 1

unsigned int	
size:	system dependent; always same size as **int**
range:	0 to *UINT_MAX*. For 4 byte word, 0 to $2^{32}-1$, i.e., 4294967295
declaration:	**unsigned int** *unum*; or **unsigned** *unum*;
constants:	5530u

Unsigned integers — 2

unsigned long	
size:	usually 4 bytes; always same size as **long**
range:	0 to *UINT_MAX*. For 4 byte, 0 to $2^{32}-1=4294967295$
example declaration:	**unsigned long** *ulnum*;
example constant:	76212000uL

unsigned short	
size:	usually 2 bytes; always same size as **short int**
range:	0 to *USHRT_MAX*. For 2 bytes, 0 to 65536
example declaration:	**unsigned short** *usnum*;
example constant:	34000u

5.2 Characters

Characters

- When working with characters, use the type **char**.
- Note that the type **char** can be signed *or* unsigned, depending on the compiler.

char	
size:	usually 8 bits; *CHAR_BIT* in `limits.h`
range:	*CHAR_MIN* to *CHAR_MAX*. For 1 byte, could be -2^7 to 2^7-1, i.e., -127 to 128 *or* 0 to 255
example declaration:	**char** *ch*;
example constant:	`'a'`

Characters: signed, unsigned

- specify the type as signed or unsigned only if you care.

signed char	
size:	usually 8 bits; *CHAR_BIT* in `limits.h`
range:	*SCHAR_MIN* to *SCHAR_MAX*. For 1 byte, -2^7 to $2^7 - 1$, i.e., -127 to 128
example declaration:	**signed char** *ch*;
example constant:	`'a'`

unsigned char	
size:	usually 8 bits; *CHAR_BIT* in `limits.h`
range:	0 to *UCHAR_MAX*. For 1 byte, 0 to $2^8 - 1 = 255$
example declaration:	**unsigned char** *ch*;

Character types

- Characters are represented in C with integer values.
- The correspondence between a given character and an integer value is determined by an agreed-upon character set, such as the ASCII character set.
- Examples of declarations:

  char *letter*;
  signed char *sletter*;
  unsigned char *uletter*;

Program `princhar.cpp`


```
#include <iostream>

int main( void )
{
    char letter = 'A';      // 'A' is character constant
    int num     = letter;
    char ch1    = 'b';      // ASCII code = 98
    char ch2    = 'B';      // ASCII code = 66

    std::cout << "letter = " << letter
         << ", num = " << num << '\n';
    std::cout << "letter + 1 = " << letter + 1
         << ", num = " << num << '\n';
    std::cout << ch1 << " - " << ch2
         << " = " << ch1 - ch2 << std::endl;
}
```

Output for Program `princhar.cpp`:

```
letter = A, num = 65
letter + 1 = 66, num = 65
b - B = 32
```

5.3 Octal, Hexadecimal Output with *ostream*

Integer constants: octal and hexadecimal

- An integer can be represented in an octal or a hexadecimal form. Octal integer constants are represented with a leading zero. Hexadecimal integer constant is represented with the leading characters 0x, or 0X.

Integer	Octal	Hexadecimal
4	04	0x4
12	014	0xc
123	0173	0x7b

- The long integer qualifier *L* can also used with the octal and hexadecimal
- Example of octal constant: 0553000L
- Example of hexadecimal constant: 0x2f6c7a333L

Program `printvar.cpp`

```
#include <iostream>
#include <iomanip>

int main( void )
{
    int num = 77;
    short int small = 0173;
    short little = 0x7b;
    long int big = 88000;
    long large = -43000L;
    unsigned int unum = 45000;
    unsigned long ubig = 330000000UL;

    std::cout << std::showbase;
    std::cout << "num (dec) = " << num
        << ", (oct) = " << std::oct << num
        << ", (hex) = " << std::hex << num << '\n';
    std::cout << "small (oct) = " << std::oct << small
        << ", little (hex) = " << std::hex << little << '\n';
    std::cout << "big (dec) = " << std::dec << big
        << ", large (dec) = " << large << '\n';
    std::cout << "unum = " << unum
        << ", ubig = " << ubig << '\n';
    std::cout << "small (dec) = " << small
        << ", little (dec) = " << little << std::endl;
}
```

Output for Program `printvar.cpp`

```
num (dec) = 77, (oct) = 0115, (hex) = 0x4d
small (oct) = 0173, little (hex) = 0x7b
big (dec) = 88000, large (dec) = -43000
unum = 45000, ubig = 330000000
small (dec) = 123, little (dec) = 123
```

- We need to include `iomanip` here for the *manipulators*:

  `std::oct` Changes the *state* of the `ostream` to displaying all integer type numbers in *octal*. *All* numbers printed while in this state are in octal.

  `std::hex` Changes the *state* of the `ostream` to displaying all integer type numbers in *hexadecimal*

`std::dec` Changes the *state* of the `ostream` to displaying all integer type numbers in *decimal*

`std::showbase` a state that displays *octal* with leading "0", *hexadecimal* with leading "0x"

5.4 Floating Point Types

Floating point data types — 1

- The floating point data type is used to represent real numbers. Real numbers include the fractional number between integers. two components: an exponent and a fraction

float	
size:	system dependent, usually four bytes.
range:	*FLT_MIN* to *FLT_MAX* defined in `float.h`
example declaration:	**float** *fnum*;
example constant:	3.456f

Floating point data types — float

double	
size:	twice the size of float, usually eight bytes
range:	*DBL_MIN* to *DBL_MAX* defined in `float.h`
example declaration:	**double** *dnum*;
example constants:	floating point notation: 3.4567 exponential notation: 4.788e+5, 3e1

Floating point data types — long double

long double	
size:	bigger than the size of double.
range:	*LDBL_MIN* to *LDBL_MAX* defined in `float.h`
example declaration:	**long double** *ldnum*;
example constants:	3.4567L exponential notation: 4.788e+5L

Program `float-io.cpp`

```
#include <iostream>

int main( void )
{
        float cost = 15.92;
        float total = 3.6e5;
        float value = 357e−1;      // value is 35.7
        double debt = 1.2e15;
        long double decrease = 5e−6;

        std::cout << "cost = "    << cost
                  << ", total = " << total << '\n';
        std::cout << "value = "   << value << '\n';

        std::cout << "debt = "    << debt << '\n'
                  << "decrease = " << decrease << '\n';
}
```

Output for Program `float-io.cpp`

```
cost = 15.92, total = 360000
value = 35.7
debt = 1.2e+15
decrease = 5e-06
```

Program `float.cpp` showing limits of floating values

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

int main()
{
        std::cout << "float min: " << FLT_MIN << '\n';
        std::cout << "float max: " << FLT_MAX << '\n';
        std::cout << "double min: " << DBL_MIN << '\n';
        std::cout << "double max: " << DBL_MAX << '\n';
        std::cout << "long double min: " << LDBL_MIN << '\n';
        std::cout << "long double max: " << LDBL_MAX << '\n';
        std::cout << "float epsilon: " << FLT_EPSILON << '\n';
        std::cout << "double epsilon: " << DBL_EPSILON << '\n';
        std::cout << "long double epsilon: " << LDBL_EPSILON << '\n';
        std::cout << "A long double constant: " << 4.788e+5L << '\n';
        return 0;
}
```


Output for Program `float.cpp`

```
float min: 1.17549e-38
float max: 3.40282e+38
double min: 2.22507e-308
double max: 1.79769e+308
long double min: 3.3621e-4932
long double max: 1.18973e+4932
float epsilon: 1.19209e-07
double epsilon: 2.22045e-16
long double epsilon: 1.0842e-19
A long double constant: 478800
```

5.5 Named Constants

Symbolic constants: #define, const, and enum

- **#define** can be thought of as a global substitution command.
- The **#define** directive consists of the **#define** keyword, a define symbol, and a replacement text.
- Program `define.cpp`:

```
#include <iostream>
#define RATE 1.5 // Note: terminated by comment or newline

int main( void )
{
    float cost = RATE * 8.0;
    std::cout << "Cost = " << cost
              << ", rate = " << RATE << '\n';
}
```

5.6 Enumerated Types: enum

Kconst and enum

- Using **const** is better than **#define**. Program `const.cpp`:

```
#include <iostream>

int main( void )
{
    const float rate = 1.5;    // with ";"
    float cost = rate * 8;

    std::cout << "cost = " << cost
         << ", rate = " << rate << '\n';
}
```

- Program `enum-2.cpp`:

```
#include <iostream>

enum weather { clouds, rain, sunny, storm };

int main( void )
{
        weather today = sunny;

        std::cout << clouds << ' ' << rain << ' '
             << sunny << ' ' << storm << '\n';
        std::cout << "today: " << today << '\n';
}
```

Enumerated types — 2

- Program `enum-3.cpp`:

```
#include <iostream>

enum weather { clouds = 30, rain = 5,
               sunny = 255, storm = 1 };
enum boolean { TRUE = 1, FALSE = 0 };

int main( void )
{
    weather today = rain;
    std::cout << clouds << ' ' << rain << ' '
              << sunny << ' ' << storm << '\n';
    std::cout << "today: " << today << '\n';
}
```

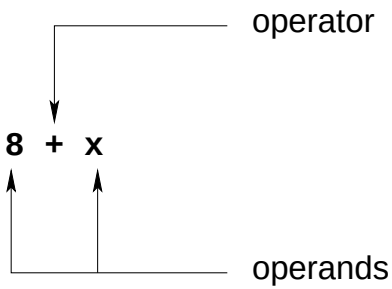

- Output of program `enum-3.cpp`:

```
30 5 255 1
```

6 Expressions

Expressions

- Expressions define the tasks to be performed in a program do calculations perform function calls and assignment operations
- Arithmetic expressions perform the standard arithmetic operations.
  - Arithmetic operator — is a symbol or a binary operator, i.e. acts upon 1 or more operands or values.
  - Operand —- may be constant, variable, or function.



6.1 Arithmetic Expressions

Arithmetic operators — unary

- Unary arithmetic operators:

  + plus, or positive

  – negative

-6	negative 6
$-(-6)$	positive 6
$-(3-7)$	positive 4
$-y$	change the sign of operand y

Arithmetic operators — binary

- Additive operators:

  + addition

  – subtraction

```
total = 8 + 5 – 2;
sum = num – 3;
```

- Multiplicative operators:

  ***** multiplication

  / division

  % modulo or remainder

```
int total;
total = 8 * 5;
total = 23 / 4; // now total has the value 5
total = 23 % 4; // now total has the value 3
```

6.2 Arithmetic Operators and Precedence

Arithmetic precedence and associativity

- One operator may take precedence over another.
- When an expression is evaluated, it is broken down into a series of subexpressions, one for each operator.
- The order in which these subexpressions are evaluated is determined by either parentheses or precedence.
- Example:

  ```
  ( 2 + 4 ) * ( 5 – 3 )
  ```

  evaluates to the same as

  ```
  6 * 2
  ```

- If there are no parentheses, precedence determines the order of evaluation
- All operators are ranked according to their precedence.
- Operators with greater precedence are evaluated first.

- Example:

  ```
  2 + 4 * 5 – 3
  ```

  evaluates to the same as:

  ```
  2 + 20 – 3
  ```

Arithmetic Precedence and Associativity (cont')

- If the operators share the same operand, the priority of one operator over the other will be determined by its associativity. Associativity describes how an operator associates its operands.
- Arithmetic operators associate left to right, whereas assignment operators associate right to left.
- Example:

  ```
  2 + 10 – 5
  ```

  evaluates the same as

  ```
  12 – 5
  ```

- The following table lists the precedence and assciativity of operators.

6.3 All Operators and their Precedence

All Operators and their Precedence

See table 1 for a list of all C operators and their precedence and associativity.

Associativity tells you whether, if the precedence is equal, whether the operators be applied from left to right (left associative), or from right to left (right associative).

6.4 Relational Expressions

Comparison Expressions

- Relational, equality, and logical expressions compare their operands. the result of the comparison is the integer value of either one or zero.
- If an operation compares successfully, the result of the expression is an integer value of 1. If an operation compares fails, the result of the expression is an integer value of 0.

Level	*Operator*	*Function*
15L	–>, .	structure member selectors
	[]	array index
	()	function call
14R	**sizeof**	size in bytes
	++, ––	increment, decrement
	~	bitwise NOT
	!	logical NOT
	+, –	unary plus, minus
	*, &	dereference, address-of
	()	type conversion (cast)
13L	*, /, %	multiply, divide, modulus
12L	+, –	arithmetic operators
11L	≪, ≫	bitwise shift
10L	<, <=, >, >=	relational operators
9L	==, !=	equality, inequality
8L	&	bitwise AND
7L	^	bitwise XOR
6L	|	bitwise OR
5L	&&	logical AND
4L	||	logical OR
3L	?:	arithmetic if
2R	=	assignment operator
	*=, /=, %=	compound assignment operators
	+=, –=, ≪=	
	≫=, &=, |=, ^=	
1L	,	comma operator

Table 1: Table of C operators and their precedence

- A relational operator compares two operands and determines whether one is greater or less than the other.

  `<` less than

  `>` greater than

  `<=` less than or equal

  `>=` greater than or equal

Relational expressions — 1

- Relational expressions can be combined with other expressions.
- Example:

```
num = 3;
abc = ( num < 5 );
```

  As $3 < 5$, the resulting value of *abc* is 1.

- Example:

```
num = 8;
abc = 5 + ( num < 5 );
```

  As $8 > 5$, the value of *abc* is $5 + 0 = 5$.

Relational expressions — 2

- The equality operators test a relationship between two operands
- result is 1 or 0.

  == equal,

  != not equal

- equality operator is a double equal sign ==
- assignment operation is a single equals sign = e.g

```
while ( test == 1 ) // comparison operation for equality
test = 1;           // assignment operation
```

6.5 Logical Expressions

Logical expressions — 1

- The logical operators compare the truth or false of their operands. determined by whether or not it has a zero value
  - If an expression evaluates to zero, the expression is false
  - If an expression evaluates to a non-zero, it is true.
- The operands of a logical operation are often relational expressions.

  `&&` logical AND

  `||` logical OR

  `!` logical NOT

Logical expressions — 2

- Here is the truth table for the logical AND operation:

expr1		*expr2*		(*expr1*) && (*expr2*)	
T	non-zero	T	non-zero	T	1
T	non-zero	F	0	F	0
F	0	T	non-zero	F	0
F	0	F	0-zero	F	0

- Truth table for the logical OR operation:

expr1		*expr2*		(*expr1*) || (*expr2*)	
T	non-zero	T	non-zero	T	1
T	non-zero	F	0	T	1
F	0	T	non-zero	T	1
F	0	F	0-zero	F	0

- Truth table for the logical NOT operation:

expr		!(*expr*)	
T	non-zero	F	0
F	0	T	1

Program `countdig.cpp`

```
/* Counts only the numeric characters '0' - '9'
   read from standard input */

#include <iostream>
int main( void )
{
    int n = 0;
    char c;

    while ( std::cin >> c ) {
        if ( c >= '0' && c <= '9' )
            n = n + 1;
    }
    std::cout << "Count of digits = " << n << '\n';
}
```


6.6 Assignment Expressions

Assignment expressions

- assignment operation is an expression.
- resulting value of the assignment expression is the value assigned to the variable in the assignment operation.
- assignment operation can be combined with other operators to form a complex expression:

```
total = ( num = ( 4 + 2 ) );
```

- Parentheses can be left out since assignment evaluate from right to left.

```
total = num = 4 + 2;
```

which is identical to

```
total = 4 + 2;
num = 4 + 2;
```

Arithmetic assignment operators

- provide a shorthand applying an arithmetic operation to a variable
  - *j* += 3; is equivalent to *j* = *j* + 3;
  - *j* *= 3; is equivalent to *j* = *j* * 3;
- List of arithmetic assignment operators:

+=	add and then assign
−=	subtract and then assign
*=	multiply and then assign
/=	divide and then assign
%=	modulo; assign remainder

6.7 Increment, Decrement Operators

Increment and decrement assignment operators

- an assignment operation in which 1 is added/subtracted to a variable and the result assigned to that variable.

- The increment or decrement operator can operate in two ways: Prefix places the increment or decrement operator before its operand. Postfix places the increment or decrement operator after its operand.
- Example:

```
x = 32;
y = ++x;
```

imply $x = 32 + 1 = 33$ and $y = x = 33$

- But

```
x = 32;
y = x++;
```

imply $y = x = 32$ and $x = 32 + 1 = 33$

Program `plusequl.cpp`

```
#include <iostream>
int main( void )
{
    int n, j, k;
    k = j = n = 4;
    std::cout << "n = " << n << ", j = " << j
         << ", k = " << k << '\n';
    n += j = 3;
    std::cout << "n = " << n << ", j = " << j
         << '\n';
}
```

Output for Program `plusequl.cpp`:

```
n = 4, j = 4, k = 4
n = 7, j = 3
```

6.8 Comma Operator

Comma operator expressions

- The comma operator expression is an expression that consists of a list of other expressions.
- The comma does not perform any operation on these expressions.
  - they are simply evaluated sequentially as if they were a series of statements.

- Example:

```
r = ( 3 * 5, 8.00 + 2.5, num = 5 );
```

- The result is the value of the last expression in its list i.e.,

```
r = num = 5
```

- The main use of comma is in the headers of **for** loops:

```
for ( i = 0, j = n; i < n; ++i, --j )
```

6.9 Arithmetic if Expressions

Arithmetic if Expressions

- A conditional expression implements an if-else decision format.
- The conditional expression consists of three subexpressions: test expression and two alternative result expressions

  ⟨*expression1*⟩ **?** ⟨*expression2*⟩ : ⟨*expression3*⟩
- If the test is true, the result will be the value of the second expression.
- If the test is false, the result will be the value of the third expression.

Arithmetic if expressions — 2

- When used in an assignment operation, the arithmetic if expression works like an if-else statement.

```
max = ( a > b ) ? a : b;
```

- equivalent to

```
if ( a > b )
        max = a;
else
        max = b;
```

6.10 Bitwise Operators

Bitwise operators — 1

- The bitwise operations allow the programmer to manipulate specific bits.
- The bitwise operations can be combined with *masks* to turn specific bits on and off.
- The bitwise *AND* operation, &, is used to *clear* specific bits in an integer operand, leaving the other bits unchanged.
- The bitwise *OR* operation, |, is used to *set* specific bits in an integer operand, leaving the other bits unchanged

Bitwise operators — 2

- Here is the truth table for the bitwise AND operation:

bit n	bit m	m & n
T 1	T 1	T 1
T 1	F 0	F 0
F 0	T 1	F 0
F 0	F 0	F 0

- Truth table for the bitwise OR operation:

bit n	bit m	m | n
T 1	T 1	T 1
T 1	F 0	T 1
F 0	T 1	T 1
F 0	F 0	F 0

Bitwise operators — 3

- The one's complement operator, ∼, is a unary operator.
- The resulting value is set to the opposite of that of the operand's bit.

bit m	∼m
0	1
1	0

- The bitwise exclusive OR operator, ^, results in 1 if the corresponding two bits are different:

bit m	bit m	m ^ n
0	0	0
0	1	1
1	0	1
1	1	0

Shift Operators

- There are two shift operators:

  ≪ left shift operator

  ≫ right shift operator

- useful for accessing individual parts of a bit pattern
- shift the bits of the left operand some number of positions to the left or right.

```
unsigned char bits = 1;      // 0000 0001
bits = bits << 1;            // 0000 0010
bits = bits << 2;            // 0000 1000
bits = bits >> 3;            // 0000 0001
```

Right Shift Operator

- the right shift operator will fill negative signed numbers with '1's from the left, but will shift 0 into the MSb (most significant bit) of unsigned numbers.
- Program `shiftright-demo.cpp`

```
#include <iostream>
#include <iomanip>

int main( void )
{
        int test_numbers[] = { -16, 16, -1, 1 };
        const int len = sizeof( test_numbers ) / sizeof( test_numbers[ 0 ] );
        for ( int i = 0; i < len; ++i ) {
                int n = test_numbers[ i ];
                unsigned u = n;
                std::cout << showbase;
                std::cout <<   "dec n: " << dec << n << "; n >> 2: " << ( n >> 2 )
                     << "\thex n: " << hex << n << "; n >> 2: " << ( n >> 2 )
                     << '\n';
                std::cout <<   "dec u: " << dec << u << "; u >> 2: " << ( u >> 2 )
                     << "\thex u: " << hex << u << "; u >> 2: " << ( u >> 2 )
                     << '\n';

        }
}
```

Right Shift Operator: example output

- output of program `shiftright-demo.cpp` (folded to fit):

```
dec n: -16; n >> 2: -4
        hex n: 0xfffffff0; n >> 2: 0xfffffffc
dec u: 4294967280; u >> 2: 1073741820
        hex u: 0xfffffff0; u >> 2: 0x3ffffffc
dec n: 16; n >> 2: 4    hex n: 0x10; n >> 2: 0x4
dec u: 16; u >> 2: 4    hex u: 0x10; u >> 2: 0x4
dec n: -1; n >> 2: -1
        hex n: 0xffffffff; n >> 2: 0xffffffff
dec u: 4294967295; u >> 2: 1073741823
        hex u: 0xffffffff; u >> 2: 0x3fffffff
dec n: 1; n >> 2: 0     hex n: 0x1; n >> 2: 0
dec u: 1; u >> 2: 0     hex u: 0x1; u >> 2: 0
```

- Notice that when shifted right:
  - unsigned values have the upper two bits zero
    - so the *unsigned* value 0xfffffff0 shifted right by two is 0x3ffffffc
  - signed values have the upper two bits the same as the previous value of the MSb (most significant bit)
    - so the *signed* value 0xfffffff0 (−16) shifted right two places is 0xfffffffc

Use of bitwise operators

- Use the AND operator '&' to *clear* individual bits, leaving the others unchanged
  - For example,

    `x = x & 0xf;`

    will clear all but the least significant four bits of `x`
- Use the OR operator '|' to *set* individual bits, leaving the others unchanged
  - For example,

    `y = y | ~0xf;`

    will set all bits except for the least significant four bits of `y`
- Use the Exlusive OR operator '^' to *toggle* (flip) individual bits, leaving the others unchanged
  - For example,

    `z = z ^ 0xf;`

    will toggle the least significant four bits of `z`, i.e., make 1s ↔ 0s

6.11 Casts

Cast expressions

- The cast operation returns the value of an expression, converting it to the type in the brackets
- Example:

  `( float ) 7`

  - This converts the integer 7 to a floating point value, 7.00.
- The operand may be any expression
  - Examples:

    ```
    ( int ) ( 5.8 * 2.7 )
    ( double ) ( k = 10 )
    ( float ) num
    ```

- If a floating point value is cast to an integer, the floating point fraction is lost.
  - Example:
    (**int**) 3.75
  - resulting value is 3
- Casts overide the compiler's concept of correctness — *use rarely*

Conversions

- Expressions can include operands of different number types
- Example: An integer can be multiplied by a float.
- C handled operands of different types by converting one of them into the same type as that of the other operand.
- Conversion determines which operand to convert by a process of *promotion* and *demotion*.

Conversions: promotion

- In expressions with two different types of operands, the operand with the smaller type is always *promoted* to that of the largest type.
- Example:

```
int num = 6;
float cost;
cost = num * 5;
```

- The expression *num* * 5 results in the integer 30 and will be promoted to a **float**, 30.0 before assigned into the *cost* variable.

Conversions: demotion

- *Demotion* from a floating point type to an integer type results in the loss of the floating point's fraction.
- Example:

```
int num;
float cost = 62.65;
num = cost;
```

- The fraction will be cut off, leaving just the integer, 62. The value 62 is then assigned to the variable *num*.

7 Statements

Statements: an introduction

- A C program is a sequence of *declarations* and *statements*.
- We have seen:
  - how to declare variables, and
  - how to create expressions using operators
  - examples of putting this all together.
- Now let's look at ***statements.***

Statements: intro — 2

- We can turn an expression such as *x* = 0 or *std*::*cout* ≪ "a" into a statement simply by putting a semicolon at the end:

```
x = 0;
std::cout << "a";
```

- We can join these into a *compound statement* by putting braces **{ }** around them.
- There are some which are used to create *loops* and make *decisions*. These are sometimes called *control-flow statements*.

7.1 Simple Statements

Expression statements

- An *expression statement* consists of any valid expression, followed by a semicolon.
- Often the expression is an assignment operation or a function call.
- Example:

```
count = 8;
num = 3 + 4;
calc();
```

- However, the expression could just as easily be an arithmetic expression or relational expression.
- Example:

```
4 + 5; // nothing done with the result, 9.
( n < 3 );
```

Null statement (empty statement)

- If there is no expression in the expression statement, nothing happens. This is called as the *null statement*.
- Example:

```
; // just a semicolon
```

7.2 Compound Statements

Compound Statements

- A compound statement is a statement composed of one or more statements.
- A compound statement consists of opening and closing braces within which statements are placed.
- Example:

```
{
    num = 6;
    fact = ( 5 - 3 );
    std::cout << fact << ", " << num << '\n';
}
```

Blocks — 1

- Variables can be declared at the beginning of compound statement. A compound statement with variable declarations is referred to as a *block*.
- The body of a function is a compound statement itself, and is often referred to as the *function block*.

Program `blocks.cpp`


```
#include <iostream>
int main( void )
{
        int num = 10;
        float cost = 100.0;
        {
                float cost = 50.0;
                num = 5;
                std::cout << "Inside: " << cost
                     << ", " << num << '\n';
        }
        std::cout << "Outside: " << cost
             << ", " << num << '\n';
}
```

- Output of program `blocks.cpp`:

```
Inside: 50, 5
Outside: 100, 5
```

7.3 Scope

Scope

- A new *scope* is created in each block
- the compiler searches for an identifier defined in the innermost scope first
- then searches the scopes that the enclose current scope...
- ... until it reaches global scope
  - global scope is outside of any block
- identifiers defined in an inner scope "hide" identifiers defined in an outer scope
- For example, in program `blocks.cpp`, there are two variables called *cost* in different nested scopes
  - The inner output statement prints the value of the *cost* variable defined in the inner scope
  - The outer output statement prints the *cost* defined in the outer scope.

7.4 Looping Statements

Iteration statements

- Loops are implemented with three *iteration statements*:
  - **while**
  - **do**
  - **for**
- Each statement repeats a statement called the *body of the loop*
- The body of the loop can be any statement, including:
  - *compound* statement, or
  - another loop, or
  - a null statement.

7.5 while Statement

while statement

- **while** loop consists of the keyword **while**, a test expression in parentheses, and a statement.
- statement is repeated for as long as the test expression evaluates to a non-zero value.

  while (⟨*test expression*⟩)
  ⟨*statement*⟩;

- Example:

```
i = 0;   // initialize the loop counter
while ( i < 5 ) {
   std::cout << "ABC ";
    ++i; // update the loop counter
         // very important
}        // the loop becomes infinite
         // without this statement that
         // changes the loop counter
```


7.6 do statement

Iteration Statements — do loop

- The **do** statement is a variation on the while statement.
- Instead of the test occurring at the beginning of the loop, it occurs at the end,
- Example:

```
i = 0;
do {
    std::cout << "abc\n";
    ++i;          // loop counter
} while ( i < 4 );
```

Program `square-1.cpp`

```
#include <iostream>

int main( void )
{
        int num;
        std::cout << "Enter a number: ";
        std::cin >> num;

        while ( num != 0 ) {
                int square = num * num;
                std::cout << "Square of " << num
                     << " = " << square << '\n';
                std::cout << "Enter a number (0 to quit): ";
                std::cin >> num;
        }
}
```

- This program has a major problem.
- What happens if we input a character that is not part of an integer?

7.7 Avoid Confusing == with =

Test Expression

- The test expression for the **while**, **for**, and **if** statements can be any valid expression,
- Example:

  - assignment operation
  - simple primary expression consisting of a variable or a constant

- A zero result evaluated is considered to be false while any non-zero result is true.

Traps with = and ==

- Don't confuse comparison with assignment in a test expression.

```
while ( i = k ) { ... }
```

- If *k* equals 0, the test will always be false
- If *k* is not equal to zero, the test will always be true.
- Examples of *incorrect* test expressions:

```
while ( n = 0 ) { /* always false */ ... }
while ( n = 3 ) { /* always true, an infinite loop */ ... }
```

- The correct way to write these test expressions is:

```
while ( n == 0 ) {
    ...
}

while ( n == 3 ) {
    ...
}
```

Program `quit-1.cpp`


```
#include <iostream>

/* This program shows problems that come from
   confusing assignment with comparison */

int main( void )
{
        int quit = 0, num = 1, square;
        while ( quit = 0 ) // Oh dear; always false
        {
                // the loop body will never be executed
                square = num * num;
                std::cout << "Square of " << num
                     << " = " << square << '\n';

                if ( num = 10 ) // Oh dear; always true
                        quit = 1;
                num++;
        }
}
```

7.8 Using a constant or single variable as a test condition

Using a constant or single variable as a test condition

- Constants are often used to write infinite loops.
- Variables are used as a shorthand for comparing the value of the variable to zero.
- Example:

```
while ( 1 )   // infinite loop
while ( 0 )   // never execute loop
while ( i )   // is equivalent to
              //       while ( i != 0 )
```

Program `square-2.cpp`

```
#include <iostream>

int main( void )
{
        int num;
        while ( 1 ) {        // or for (;;)
                std::cout << "Please enter a number: ";
                if ( ! ( std::cin >> num ) )
                        break;
                int square = num * num;
                std::cout << "Square of " << num
                     << " = " << square << '\n';
                char ch;
                std::cout << "Another square? (y/n) ";
                if ( std::cin >> ch && ch != 'y' )
                        break;
        }
}
```

7.9 while and the null statement

while test and null statement

- A **while** statement can be written in which the test expression does all the work. The statement following the test expression is simply a null statement.
- Example:

```
while ( std::cin.get( ch ) && ch != '\n' )
    ; // do nothing
```

- Three actions take place in the above example:
  - function call to the `istream` member function *get*()
  - *std*::*cin.get*(*ch*) will return false if reach end of input file;
  - inequality operation in which character value obtained by *std*::*cin.get*() is tested against a newline constant.
- End result is that we wait till we get a newline character, or end of file

7.10 for Statement

The for Statement

- The for statement consists of three expressions followed by a statement.

```
for ( ⟨expression 1⟩; ⟨expression 2⟩; ⟨expression 3⟩ )
        ⟨statement⟩;
```

- ⟨*expression 1*⟩ is executed once, before loop begins. It is often used to *initialize* variables used in the test expression.
- ⟨*expression 2*⟩ is the *test* expression for the loop. When it evaluates as false, the loop stops.
- ⟨*expression 3*⟩ *update* expression. It is executed within the loop and is usually used to update variables used in the test expression.

Example of for loop

```
for ( int i = 1; i < 3; ++i ) {
        std::cout << "OK\n";
}
```

7.11 Comparing while and for

for and while: a comparison

for loop:

```
for ( ⟨init⟩; ⟨test⟩; ⟨update⟩ ) {
        ⟨body of loop⟩
}
```

example:

```
for ( int i = 0; i < 3; ++i )
        std::cout << "loop " << i
                  << '\n';
```

while loop:

```
⟨init⟩;
while ( ⟨test⟩ ) {
        ⟨body of loop⟩
        ⟨update⟩;
}
```

example:

```
int i = 0;
while ( i < 3 ) {
        std::cout << "loop " << i
                  << '\n';
        ++i;
}
```

Nested loops — loop within a loop

- The inner loop executes fully within each iteration of the outer loop
- Example:

```
for ( int k = 0, i = 0; i < 3; ++i ) {
        for ( int j = 0; j < 3; ++j ) {
                ++k;
                std::cout << k << ' ';
        }
        std::cout << '\n';
}
```

- See example program `nested-for.cpp`
- Output:

```
1 2 3
4 5 6
7 8 9
```

7.12 if and switch Statements

Conditions: if statement

- The **if** and **switch** statements are decision making structures that determine which statements are to be executed and which are not.
- **if** is a condition placed on a statement's execution.
  - if the condition is true, the statement is executed
  - if the condition is false, the statement is not executed

```
if ( ⟨test expression⟩ )
        ⟨statement⟩;
```

if and else

- The **if** statement can choose between several choices using **else**:

```
if ( ⟨condition⟩ ) {
        doThis();
} else if ( ⟨condition 2⟩ ) {
        doThat();
} else {
        doTheOther();
}
```


- Only *one* action is performed
- the first that matches is done.

The switch statement — 1

- The **switch** statement provides a convenient way to choose among several alternatives.
- It is a conditional statement (selection)

```
switch ( ⟨integer expression⟩ ) {
    case ⟨integer⟩:
        ⟨statements⟩;
        break;
    case ⟨integer⟩:
        ⟨statements⟩;
        break;
    default:
        ⟨statements⟩;
        break;
}
```

The `switch` statement — 2

- The **switch** compares an *integer value* against a set of integer constants.
- The execution will continue unless a **break** is encountered.

7.13 break, continue, goto

The jump Statements

- The jump statements are non-structured control statements that allow the program to jump across statements.
- Strickly speaking, these are not allowed in structured programming.
- However, **break** and **continue** are especially useful.
- **break** and **continue** statements are used with **while**, **for**, and **switch** statements.
- **break** provides an exit condition other than that of the statement's test expression.

Example of use of continue

- Kernighan and Ritchie provide this example of using **continue**:

```
for ( i = 0; i < n; ++n ) {
        if ( a[ i ] < 0 ) // skip negative elements
                continue;
        // do positive elements
}
```

- See also my example program `cat.cpp`

goto: use it seldom

- **goto** and *label* statements allow the program to jump to any statement.
- Using **goto** can cause the program to become very hard to understand
- Use it only when you really have to
  - An example is to break out of a nested loop from the inner loop

```
for ( i = small; i < big; ++i )
        for ( j = small2; j < bigger; ++j )
                if ( i == j )
                        goto equal;
equal:
```

- *equal* here is a label

7.14 Exercises

1. Write a program using the following code fragment as a guide:

```
int i = 10;
if ( i > 0 )
        std::cout << "i > 0\n";
if ( i > 1 )
        std::cout << "i > 1\n";
if ( i > 2 )
        std::cout << "i > 2\n";
// ...
if ( i > 10 )
        std::cout << "i > 10\n";
else
        std::cout << "something else\n";
```


2. Save your source to a different file name, but replace all the **if**s (except the first) to **else if**s.
3. Run your two programs and compare the output.
4. Write a program contianing a **while** loop and a **for** loop that both output data and do the same thing.

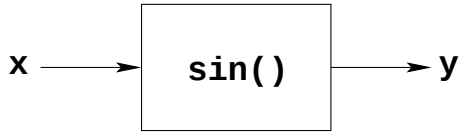
8 Functions

Functions — 1

- A big program may be too complex to hold in your head.
- I need a way to break a big problem into many small, easy-to-understand problems.
- One way to break a problem into small problems is to divide a problem into small parts that can be written as *functions*.
- A *function* is a number of statements that:
  - Perform one easily-understood job
  - are given a single name.
- A function is a little bit like a simple IC, with input pins and output pins.

Functions — 2

- A function may have inputs and outputs:



- The function call:

  `y = sin( x );`

  can be represented by the block diagram above.
- The inputs go in the parentheses: ()
- The output of the function can be assigned, as above.

Functions — 3

- To write and use functions in your program, there are two things to consider:
- When you *define* the function, you write the statements that the function will perform when it is *called*.
- When you want to *use* the function, we say we *call* the function.

8.1 Defining Functions

Function definition

- A function definition consists of a *header* and a *body*.
- *header* contains the function name, its return type, and parameter types.
- The *body* of a function is a block, which usually contains variable declarations and statements.

```
⟨return type⟩ ⟨function_name⟩( ⟨parameter list⟩ )
{
        ⟨variable definitions⟩;
        ⋮
        ⟨statements⟩;
}
```

- Example:

```
void calc( void )
{
        int num;
        num = 5;
}
```

8.2 Calling Functions

Function definition and call Program `nowincal.cpp`


```
// Program to call a function

#include <iostream>
// function definition:
void calc( void )
{
        std::cout << "Now in Calc\n";
}

int main( void )
{
        std::cout << "Hello World\n";
        calc();
        std::cout << "Now in Main\n";
}
```

8.3 Using return Value from Functions

Functions as expressions

- A function call is an expression whose value is the function's return value.
- Program `calc-2.cpp`

```
#include <iostream>
float calc( void )
{
        return 8.0 * 5.35;
}

int main( void )
{
        float res1 = calc();
        float res2 = 7 + 5 * calc();
        if ( calc() > 5 )
                std::cout << "Larger\n";
        std::cout << "res1 = " << res1
             << ", res2 = " << res2 << '\n';
}
```

return statement and function return value

- The **return** statement consists of the keyword **return**, an expression, and a semicolon.
- Syntax: **return** ⟨*expression*⟩ **;**
- The ⟨*expression*⟩ is called the *return* expression.
- **return** statement will:
  - end the processing of a function
  - make execution continue from where the function was called, and
  - specify the function's return value.

Function return value: example Program `calc-3.cpp`

```
#include <iostream>
// Example of a function that takes
// parameters
float calc( int num, float calc_rate )
{
    return calc_rate * num;
}

int main( void )
{
        float rate = 2.0;
        float res = calc( 5, rate );
        std::cout << "res = " << res << '\n';
}
```

Return inconsistencies

- Return value inconsistencies: occur when the *return expression* has a type different from the function's *return type*.
- The *return expression* is what comes between the keyword **return** and the semicolon.
- The *return type* is what is written before the name of the function in a function definition.

Program `retbad-1.cpp`

```
#include <iostream>
short getnum( void )
{
        long num = 2147483647L;
        std::cout << "Number is " << num << '\n';
        return num;
}

int main( void )
{
        long res = getnum(); // return value inconsistency
        std::cout << "result is " << res << '\n';
}
```

- Output for program `retbad-1.cpp`:

```
Number is 2147483647
result is -1
```

Program `retbad-2.c`

```
#include <stdio.h>

calc( void )        /* return type missing */
{                   /* gives return type inconsistency */
                    /* legal in C */
                    /* Illegal in C++, won't compile */
    float cost = 8.0 * 5.35;
    return cost;
}

int main( void )
{
    float res1;
    res1 = calc();
    printf( "%f\n", res1 );
}
```

- Output from program `retbad-2.c`:

```
42.000000
```

8.4 Function Parameters

Function parameters

- Functions may have zero or more *parameters*.
- Parameters are usually the **inputs** to the function.
- Parameters appear in the parentheses after the name of the function, both in the function definition and in the function call.
- The *type* and *number* of parameters must match in:
  - function definition, and
  - function call.

Function parameters — 2

- In this example the parameters match
- Program `funcmult.cpp`

```
#include <iostream>

float mult( int a, float b )
{
    return a * b;
}

int main( void )
{
    int x = 3;
    float y = mult( x, 4.5 ); // function call
    std::cout << "y = " << y << '\n';
}
```

Function parameters — 3

- In the function definition of *mult*():

```
float mult( int a, float b )
{
    return a * b;
}
```

  - the first parameter is called *a*.
  - the second parameter is called *b*.

- In the function call,

```
float y = mult( x, 4.5 ); // function call
```

  - the value of *x* is copied to *a*, the value 4.5 is copied into *b*.
  - The *type* of the parameter in the function call matches the type of the parameter *in the same position* in the function definition.
  - This is like pins on an IC plugging into the holes in the IC socket.

9 Arrays

Arrays

- An array is a collection of objects, all of the same data type.
- Any one data type can be used in an array.
  - an array of integers
  - an arrary of characters
  - an array of structures
  - an array of pointers
- The declaration of an array reserves memory, which is then managed by pointers (to be discussed in next section).
- Array objects themselves are actually referenced through pointer indirection.

9.1 Defining Arrays

Array declaration

- An array declaration consists of 4 parts
  - the data type
  - array name,
  - square brackets around the...
  - ... number of objects in the array

Arrays — 2

- The declaration below declares an array of 5 integers. The array name is *mynums*

```
int mynums[ 5 ];
```

- Many different kinds of arrays may be declared, each having its own data type and number objects.

`int total[10];`	an array of ten integers (i.e. `total[0]`, `total[1]`,... `total[9]`)
`char name[40];`	an array of forty characters (i.e. `name[0]`, `name[1]`,... `name[39]`)

Array initialisation

- When a variable is defined it can be *initialised* with a value. In the declaration:

```
char mychar = 'E';
```

- the variable *mychar* is *initialised* with the character `'E'`.
- The elements of an array can also be initialised in an array declaration. The initalisation values are listed within curly braces and separated by commas:

```
int mynums[ 5 ] = { 3, 4, 5, 6, 7, };
```

Array Length

- When using standard C, the initialization part of the array declaration can be used be left out of the array declarations. The number is, instead, determined by the number of values in the initialisation block.
- Program `array-1.cpp`

```
#include <iostream>
int main( void )
{
        char letters[] = { 'A', 'B', 'C', };
        int totals[] = { 23, 8, 11, 31, };
        std::cout << letters[ 1 ] << ' ';
        std::cout << totals[ 3 ] << '\n';
}
```


Array references and array notation

- Once an array has been declared, its objects can be referenced and used in expressions.
- The array name, together with the position of an object in the array is used to reference an object.
- The objects in an array are arranged in sequence, starting from zero. The number of an object's place in that sequence is referred to as either the object's index or subscript.
- In the example above, we can see

```
mynums[ 0 ] = 3;  // first object
mynums[ 1 ] = 4;  // second object
mynums[ 2 ] = 5;  // third object
mynums[ 3 ] = 6;  // fourth object
mynums[ 4 ] = 7;  // fifth object
```

9.2 Arrays and Loops

Array management and loops

- Operations cannot be performed on an array as a whole.
- To assign a set of values to an array, you need to assign a value to each element individually.
- An array is only a collection of objects. It is not an object itself. We use loops to process all these objects.
- Program `arrayprt.cpp`

```
#include <iostream>
int main( void )
{
        int nums[] = { 23, 8, 11, 31 };
        for ( int i = 0; i < 4; ++i ) {
                std::cout << nums[ i ] << ' ';
        }
        std::cout << '\n';
}
```

Array Management and Loops — 2

- A common rule of thumb is that the test for the end of an array is the less than operator, <, tested against the number of objects declared in the array.

- Example: this **for** loop prints each number in the array:

```
int nums[ 4 ] = { 42, 1000, 7, 103 };
for ( int i = 0; i < 4; ++i )
    std::cout << "this num is " << nums[ i ] << '\n';
```

Using constants for array size

- In the listing below, the same symbolic constant, **max**, is used in both the array declaration and the test for the last array object in the **for** statement.

```
#include <iostream>
const int max = 4;
int main( void )
{
    int mynums[ max ] = { 23, 8, 11, 31 };
    for ( int i = 0; i < max; ++i )
        std::cout << mynums[ i ] << ' ';
    std::cout << '\n';
}
```

9.3 Exercise

Write a program that:

1. Reads up to 20 numbers into an array;
2. After reading those numbers, the program calculates and prints the sum of the numbers in the array.
3. Note that your program should stop attempting to read numbers when there are no more numbers to read, i.e., because you have reached end of file, or a character is in the input that cannot be part of a number.

9.4 Strings

Arrays of characters: strings

- A *string* is an array of characters.
- Here are examples of definitions of strings:

```
char name[ 20 ];
char string[] = "This is a string";
char str[ 10 ] = "string";
char letters[ 10 ] = {
 's', 't', 'r', 'i', 'n', 'g', '\0'
};
```


- Note that the string definitions *str* and *letters* are equivalent.
- Note that a string is automatically ended with a special character called the *null* character, `'\0'`

Arrays of characters: strings — 2

- Because the string has the extra `'\0'` character at the end, the array of characters must be long enough to hold it.
- Example:

```
char string2[] = "string";
```

- ... has seven characters, so this would be wrong:

```
char string3[ 6 ] = "string"; // too short!
```

- but these are okay:

```
char string4[ 7 ] = "string";
char string5[ 100 ] = "string";
```

Careful: strings cf. characters

- A common mistake is to confuse a string with a character.
- Example:

```
char c;
// Wrong!  A character has single quotes: 'A'
c = "A";
```

- The string `"A"` is actually *two* characters: `'A'` then `'\0'`.
- One final note: do not assign strings!

```
char str[ 100 ];
str = "this is a string"; /* Oh no, a mistake!
                             Use strcpy() library
                             function instead. */
strcpy( str, "this is a string" ); // OK
```

Working with strings

- The standard C++ `strings` library is the best choice for simplicity, but the Borland 3.1 compiler does not seem to support it
- The standard library that comes with (nearly) every C compiler provides lots of functions for working with strings.
- To use them, put:

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

- at the top of your program.
- Here are some:
- *strlen*() — give the length of a string
- *strcpy*() — copy one string to another string
- *strcmp*() — compare two strings
- *strcat*() — join one string onto the end of another

Working with strings 2

- Example using *strlen*():

```
int len;
len = strlen( "a string" ); // len = 8
```

- Example using *strcpy*():

```
char str1[ 100 ], str2[] = "a string";
strcpy( str1, str2 );
```

10 Pointers

Pointers

- Any object defined in a program can be referenced through its address. A pointer is a variable that has as its value the address of an object.
- A pointer is used as a referencing mechanism.
- A pointer provides a way to reference an object using that object's address.

- There are 3 elements involved in this referencing process:
  - a pointer variable
  - an address
  - and another variable

Pointer holds address

- A pointer variable holds the *address* of another variable of a *particular type*
- Program `pointer-1.cpp`

```
#include <iostream>
int main( void )
{
    int num = 12;
    int *nptr;
    nptr = &num;
    std::cout << "num holds " << num
        << " and nptr points to " << *nptr << '\n';
    std::cout << "The address held in nptr is " << nptr
        << '\n';
}
```

- Output of program `pointer-1.cpp`:

```
num holds 12 and nptr points to 12
The address held in nptr is 0xbfafade8
```

10.1 Pointers as Function Parameters

Pointers as Function Parameters

- If you try to write a function to swap its parameters like this:

```
void swap( int x, int y )
{
        int temp = x;
        x = y;
        y = temp;
}
```

and call it like this: *swap*(*a*, *b*);, the values of *a* and *b* are copied by value, so the final values are not changed.

- The right way is to pass the *address* of *a* and *b* like this: *swap*(*&a*, *&b*); and define the function like this:

```
void swap( int *x, int *y )
{
        int temp = *x;
        *x = *y;
        *y = temp;
}
```

- This is how to change the value of a parameter.

11 Arrays and Pointers

11.1 Strong relationship between arrays and pointers

C Arrays are very low level

- Pointers and arrays have a strong relationship
- Any operation using array subscripting can be done using pointers
- If we define an array of integers and a pointer to an integer like this:

```
int a[ 10 ];
int *pa;
```

  and if we make *pa* point to the start of the array *a*[ ] like this:

```
pa = &a[ 0 ];
```

  then the value of **pa* is the same as is stored in *a*[0].

- The pointer *pa* + 1 points to the value of *a*[1], so this statement is true: *(*pa* + 1) == *a*[1]
- If we add '1' to a pointer, we point to the address just after the value stored at that pointer.
  - If **sizeof**(**int**) is 4, and if addresses each hold one character, then the address *pa* + 1 is 4 address locations higher than the address *pa*.
  - The *type* of the pointer determines what address you get when you increment a pointer.
- similarly *(*pa* + *i*) == *a*[*i*] is true.

C Arrays and pointers

- The name of an array is the same as the location of the first element, so these two statements are equivalent:

```
pa = &a[ 0 ];
pa = a;
```

- These statements are also true:

```
a[ i ] == *( a + i );
&a[ i ] == a + i;
```

- An expression made of an array and index has an equivalent expression made with a pointer and offset.
- *Important:* we can do

```
++pa; // okay; now pa points to a[ 1 ]
```

- . . . but not:

```
++a;  // compiler error
```

- . . . because an *array name is a constant*, but an ordinary pointer is not.

Passing arrays to functions

- If a parameter is an array name, inside the function it is a pointer to the first element of the array
- If you find the size of an array with **sizeof**, you are given the number of elements × the size of one element
- Inside a function, the size of an array parameter is the size of a pointer.
- See program `array-parameter.cpp`.
- Note that the **sizeof** operator gives the number of characters in either a type or an expression:
- **sizeof**(*T*) gives the number of characters in the type *T*
- **sizeof** *expression* gives the number of characters in the expression *expression*.

Program `array-parameter.cpp`

```
#include <iostream>

int ar[ 10 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
void check_array_parameter( int a[] )
{
        std::cout << "sizeof( a ) = " << sizeof( a ) << '\n';
        std::cout << "sizeof( ar ) = " << sizeof( ar ) << '\n';
}

int main( void )
{
        check_array_parameter( ar );
        int nelements = sizeof( ar ) / sizeof ar[ 0 ];
        std::cout << "sizeof( ar ) = " << sizeof( ar ) << '\n'
             << "number of elements in ar[] is " << nelements
             << '\n';
}
```

- Output of `array-parameter.cpp`

```
sizeof( a ) = 4
sizeof( ar ) = 40
sizeof( ar ) = 40
number of elements in ar[] is 10
```

Passing arrays to functions — 3

- To work properly, you need to *pass the length of an array* as a *separate parameter* together with the array.
- Example:

```
double mean( int nums[], int len )
{
    int sum = 0;
    for ( int i = 0; i < len; ++i ) {
        sum += nums[ i ];
    }
    if ( len == 0 ) return 0;
    return ( double ) sum / len;
}
```

- Note that it makes no difference to write the parameter as **int** *nums*[100], since that length information will not be passed to the function as part of the **int** *nums*[100] parameter

- If the length of the array passed to *mean*() is 100, then that number must be passed as a *separate parameter*.

12 Multidimensional Arrays and arrays of pointers

12.1 Arrays of pointers

Arrays of pointers

- Arrays of pointers are very commonly used in C, because this gives much greater flexibility than alternatives (see slide §71)
- Can easily sort an array of pointers; copy only the address, not the data
- Can define an array of pointers to lines like this:

```
const int maxlines 10000;
char *line[ maxlines ];
```

- We must make sure that we allocate memory using the **new** operator for this as we read lines.

12.2 Memory Allocation

Allocating memory with new

- If we don't know how much data we will read, we need to *allocate* memory as we need it
  - The **new** operator allocates memory as it is needed
  - deallocate (free) memory with the **delete** operator
- If allocating a scalar value, use syntax like this:

  ⟨*pointer*⟩ = **new** ⟨*type*⟩;

  and free the memory like this:

  delete ⟨*pointer*⟩;

- if allocating an *array* or *string*, use syntax like this:

  ⟨*pointer*⟩ = **new** ⟨*type*⟩[⟨*length*⟩];

and free the memory like this:

delete `[]` ⟨*pointer*⟩`;`

- the result from **new** is a null pointer if the memory cannot be allocated.
  - *Always* check the return value of **new**

Example use of new: Program `new-1.cpp`

- Program `new-1.cpp` does the following:
  - dynamically allocate an array of ten integers
  - terminate if allocation doesn't succeed
  - put a value into each element of the array
  - print each value
  - free up the memory

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

int main( void )
{
        const int maxn = 10;
        int *a = new int[ maxn ];
        if ( a == NULL ) {
                std::cerr << "Out of memory!\n";
                exit( 1 );
        }
        for ( int i = 0; i < maxn; ++i )
                a[ i ] = i + 1;
        for ( int i = 0; i < maxn; ++i )
                std::cout << "a[ " << i << " ] = "
                     << a[ i ] << '\n';
        delete [] a;
}
```

Program `new.cpp` — 1


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

// Read all of input into memory.      Normally we would process one line
// at a time, as we read it.

// Read each line into a string
// allocate memory for the string and copy the string into that memory
// add the newly allocated string to an array of pointers

int read_lines( char *lines[], int maxnlines )
{
        const int maxlinelen = 8000;
        char line[ maxlinelen ];
        int nlines = 0;
        while ( std::cin.getline( line, maxlinelen ) ) {
                int len = std::cin.gcount(); // includes space for '\0'
                char *p;
                if ( nlines >= maxnlines
                     || ( p = new char[ len ] ) == NULL ) {
                        return -1;
                } else {
                        strcpy( p, line );
                        lines[ nlines++ ] = p;
                }
        }
        return nlines;
}
```

Program `new.cpp` — 2

```
void write_lines( char *lines[], int nlines )
{
        while ( nlines-- > 0 )
                std::cout << *lines++ << '\n';
}

void free_lines( char *lines[], int nlines )
{
        while ( nlines-- > 0 )
                delete [] *lines++;
}

int main( void )
{
        const int maxlines = 10000;
        char *line[ maxlines ];

        int nlines = read_lines( line, maxlines );
        if ( nlines >= 0 ) {
                write_lines( line, nlines );
                free_lines( line, nlines );
        } else {
                std::cerr << "Input is too big to read\n";
        }
}
```

12.3 Multidimensional Arrays

Multidimensional Arrays

- Not used as much as arrays of pointers
- Usually we allocate memory with new for each string

```
int matrix[ 2 ][ 4 ] = {
    { 1, 2, 3, 4 },
    { 10, 20, 30, 40 }
};
```

  defines a rectangular matrix.

- We can access the entry with the value 30 with

```
int entry = matrix[ 1 ][ 2 ];
```


- Note that this is *wrong*, and just uses the comma operator:

```
    int entry = matrix[ 1, 2 ]; // WRONG!!!
```

  %

12.4 Command Line Arguments: *argc, argv*

argc and *argv*

- The *main*() function takes two optional parameters that are always called *argc* and *argv*:

```
int main( int argc, char *argv[] )
```

- parameter *argc* is the number of arguments on the command line *including the program name*
- parameter *argv* is a pointer to an array of command line arguments
- if the program `echo` is called like this:

```
echo this is a test
```

- then *argc* is 5, *argv*[0] is `"echo"`, *argv*[1] is `"this"`, *argv*[2] is `"is"`, *argv*[3] is `"a"`, *argv*[4] is `"test"` and finally *argv*[5] is the null pointer.

Program `echo.cpp`

- The program `echo.cpp`:

```
#include <iostream>

int main( int argc, char *argv[] )
{
        for ( int i = 1; i < argc; ++i )
                std::cout << argv[ i ] << ' ';
        std::cout << '\n';
}
```

- prints its parameters, like the echo command in the Windows `CMD.EXE` shell, or like the echo command built into the `bash` shell that is popular with Linux.

13 Structures

Structures — 1

- A structure consists of a set of data objects that can be referenced as one object.

```
struct ⟨tag name⟩
{
        ⟨list of declarations⟩
};
```

- A tag can be used to label a structure type declaration. In the structure type declaration, the keyword struct may be followed with a tag placed before the opening brace of the declaration list,
- Example:

```
struct employee {
    int id;
    float salary;
};
```

Defining struct variables

- We can define a variable *person* of the type **struct** *employee* like this:

```
struct employee {
    int id;
    float salary;
};
// somewhere else:
struct employee person;
```

- We can now refer to the two values in the variable *person* as *person.id* and *person.salary*

Initialising struct variables

- We can initialise a structure when it is defined by putting a list of values in braces, as we do for arrays
- The first item in that list initialises the first element of the structure,
- the second item initialises the second element of the structure,
- ...

- We could initialise our **struct** *employee* like this:

```
struct employee person = { 8, 80000 };
```

- After this, *person* will contain the same values as in the next slide.

Structures: Program `struct.cpp`

```
#include <iostream>
int main( void )
{
        struct employee {
                int id;
                float salary;
        };
        struct employee person;
        person.id = 8;
        person.salary = 80000;
        std::cout << "ID = " << person.id << '\n';
        std::cout << "Salary = $" << person.salary << '\n';
}
```

13.1 Passing Structures to Functions

Accessing a structure through a pointer

- Given a pointer to a struct declared as:

```
struct employee {
        int    id;
        float salary;
};
struct employee person;
struct employee *p = &person;
```

- we could access the members with the *arrow operator* '–>' like this:

```
p->id = 8;
p->salary = 80000;
std::cout << "ID = " << p->id << '\n';
std::cout << "Salary = $" << p->salary << '\n';
```

- There is nothing magic about the "–>" operator; it is just a shorthand used, because we often access members of structures through pointers
- Note that "*p*–>*id*" is equivalent to "(**p*).*id*", and "*p*–>*salary*" is equivalent to "(**p*).*salary*".

Passing Structures to Functions

- Unlike arrays, structures are passed to functions *by value*
- That means that your function only gets a copy of the structure.
- If you want to modify the original structure, you need to either:
  - return the modified structure, or
  - pass a pointer to the structure.

Passing Structures: example

Here we use the **struct** *employee* defined previously. Passing structure by value

```
struct employee raise_salary( struct employee p, float raise )
{
    p.salary += raise;
    return p;
}
// in another function:
struct employee manager = { 50, 100000 };
manager = raise_salary( manager, 20000 );
```

Passing a pointer to structure:

```
void raise_salary( struct employee *p, float raise )
{
    p->salary += raise;
}
// in another function:
struct employee manager = { 50, 100000 };
raise_salary( &manager, 20000 );
```

Structures: Program `complex.cpp`


```
#include <iostream>
struct complex {
        int re;
        int im;
};

complex cadd( complex z1, complex z2 )
{
        complex zt;
        zt.re = z1.re + z2.re;
        zt.im = z1.im + z2.im;
        return zt;
}

int main( void )
{
        complex za;
        za.re = 1;
        za.im = 2;
        complex zb = za;
        complex zc = cadd( za, zb );
        std::cout << "zc.re = " << zc.re
             << ", zc.im = " << zc.im << '\n';
}
```

Output of program `complex.cpp`:

```
zc.re = 2, zc.im = 4
```

13.2 typedef

typedef

- A **typedef** is used to allow the programmer to give another name to a type.
- **typedef** ⟨*type*⟩ ⟨*NAME*⟩**;**
  defines ⟨*NAME*⟩ as a new name for the existing type ⟨*type*⟩
- Example:

```
typedef float FF; // FF is now a type
FF abc;           // equivalent to float abc;
```

- Often **typedef** is used with **struct** variables to avoid needing to type the word "**struct**"

– I do not encourage you to do this.
– See `http://www.kroah.com/linux/talks/ols_2002_kernel_coding` and search for "`typedef` is evil" at section 3.5.

14 Reading and Writing Files

14.1 `fstream`: file input and output

Text Files — `ifstream` and `ofstream`

- We work with text files much the same way as with standard input and standard output
- Open a file for input — **`ifstream`**
- Open a file for output — **`ofstream`**
- when the `fstream` object goes out of scope, the file is *automatically closed*

14.2 Error Handling

If cannot open a file, what next?

- If you fail to create an `ifstream` object (perhaps the file does not exist), the object evaluates as *false*
- It is *always essential* to test the result of things that can go wrong, and provide an error message or return an error code
- If things have gone wrong, it may be better to stop the program using the *exit*() command.
- Note: **#include** <*stdlib.h*> when you use *exit*().

Text Files — 7


```
#include <iostream>
#include <fstream>
// ...
std::ifstream fin( "data.in" );
if ( ! fin ) {
        std::cerr << "error: unable to open file "
             << "'data.in' for reading\n";
        exit( 1 );
}
std::ofstream fout( "data.out" );
if ( ! fout ) {
        std::cerr << "error: unable to open file "
             << "'data.out' for writing\n";
        exit( 1 );
}
```

Appending to files

- To append to a file, use the extra parameter *std*::*ios*::*app* when defining the `ofstream` object:

```
#include <iostream>
#include <fstream>
// ...
std::ofstream fout( "data.out", std::ios::app );
if ( ! fout ) {
        std::cerr << "error: unable to open file "
             << "'data.out' for appending\n";
        exit( 1 );
}
```

- There are a number of other parameters available.
- OR them together with the bitwise OR operator "|"

14.3 Binary files

Binary files — 1

- To open a file for binary input or output, use the extra parameter *std*::*ios*::*binary* when defining the `ofstream` or `ifstream` object:

```
#include <iostream>
#include <fstream>
// ...
std::ofstream fout( "data.out", std::ios::app | std::ios::binary );
if ( ! fout ) {
        std::cerr << "error: unable to open binary file "
             << "'data.out' for appending\n";
        exit( 1 );
}
```

- Here we open a binary file for input:

```
#include <iostream>
#include <fstream>
// ...
std::ifstream fin( "data.in", std::ios::in | std::ios::binary );
if ( ! fin ) {
        std::cerr << "error: unable to open binary file "
             << "'data.in' for reading\n";
        exit( 1 );
}
```

Binary files — 2

- Here we open a binary file for output:

```
#include <iostream>
#include <fstream>
// ...
std::ofstream fout( "data.out", std::ios::out | std::ios::binary );
if ( ! fout ) {
        std::cerr << "error: unable to open binary file "
             << "'data.out' for writing\n";
        exit( 1 );
}
```

14.4 Character I/O

Character Input and Output

- We can treat a file as a stream of characters. The `istream` member function *get*() and the `ostream` member function *put*() read and write one character at a time.

- Program `copy-file-to-output.cpp`:

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
int main( void )
{
        char ch;
        std::ifstream fin( "abc.txt" );
        if ( ! fin ) {
                std::cerr << "Cannot open file abc.txt\n";
                exit( 1 );
        }
        // skips whitespace.
        // while ( fin >> ch )
        //         std::cout << ch;
        while ( fin.get( ch ) )
                std::cout.put( ch );
}
```

Character I/O with Text Files — 2

- Program `copyfile.cpp`

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
int main( void )
{
        char ch;
        std::ifstream fin( "abc.txt" );
        std::ofstream fout( "mmm.txt" );
        if ( ! fin || ! fout ) {
            std::cerr << "Problem opening files\n";
            exit( 1 );
        }
        // The following skips white space:
        // while ( fin >> ch )
        //       fout << ch;
        while ( fin.get( ch ) )
                fout.put( ch );
}
```

14.5 Reading a Line at a time: `getline()`

Working with Lines in Text Files

- We often want to work with one line of a text file at a time
- the `istream` member function *getline*() reads from a file and places it in a string or character array, without the ending newline.

  std::*cin*.*getline*(⟨*string or character array*⟩, ⟨*maximum length*⟩);

- *getline*() returns false if there is any error.

Program `copy-lines-from-file.cpp`

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

int main( void )
{
        const int maxline = 100;
        char line[ maxline ];
        std::ifstream fin( "abc.txt" );
        if ( ! fin ) {
                std::cerr << "Cannot open file abc.txt\n";
                exit( 1 );
        }
        while ( fin.getline( line, maxline ) )
                std::cout << line << '\n';
}
```

14.6 I/O of other data to/from Text Files

Reading other data from Text Files — 1

- We can write formatted text to text files just as we can to standard output with *std*::*cout*
- Program `fileout.cpp`

```
#include <iostream>
#include <fstream>
int main( void )
{
        int num = 127;
        std::ofstream fout( "abc.txt" );
        fout << num;
        std::cout << "Wrote '" << num
             << "' to abc.txt\n";
}
```


Reading other data from Text Files — 2

- We can read formatted text from text files just as we can from standard input with *std*::*cin*
- Program `filein.cpp`:

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
int main( void )
{
        int num;
        std::ifstream fin( "abc.txt" );
        if ( ! fin ) {
                std::cerr << "Cannot open abc.txt\n";
                exit( 1 );
        }
        fin >> num;
        std::cout << "we got '" << num
             << "' from abc.txt\n";
}
```

15 Guidelines

15.1 Style Guidelines

Program layout: rules of thumb

- Use spaces after commas, around operators,
- Example:

  printf("%d", *i*);

  not

  printf("%d",*i*);

  and

  x = *x* + 3;

  not

  x=*x*+3;

- I suggest you put your *main*() function last.
  - avoids the need to put "function prototypes" that need unnecessary extra maintainance
- Use modern books about C, not very old ones.
- Indent your program to make it easy to follow.
- Indent the body of loops and **if** statements.

15.2 Program Design

Program design

- "Real" programs need to be designed; they may be too complicated to hold in your head all at one time.
- Sitting at the keyboard and typing a program as you think it up may work for small programs, but bigger programs written this way will become very messy and expensive to maintain.
  - The result is rather like a rough sketch to try out ideas.
  - You may want to start again after the experience you get from this "sketch"
- The greatest cost for a program is usually in maintaining it.
- Flowcharts:
  - are okay for simple programs
  - are good for representing complicated, unstructured looping and branching
  - **But:** a flowchart can easily become more complicated than the program itself!

Pseudocode: a basic design tool

- *Pseudocode* is a more practical way to show how your program will work.
- Pseudocode is a mixture of: English and structured C programming statements, such as **if**, **while**, **do**.
- Pseudocode should be simpler and easier to understand than the final program source code
- Pseudocode should be written *before* you type in your new program!

Program design: top-down

- Top-down design looks at the big picture first, forgetting the details.
- Write pseudocode including only these important, big steps, leaving out small steps at first. This is like your *main*() function.
- Write more pseudocode for each of these big steps, giving more detail, but not the smallest details. These are written rather like function definitions.
- For each step, write out more pseudocode like more function definitions, until you have enough detail to begin writing your source code.
- Check your design carefully before you move on.

15.3 Modules

Making an application from many modules

- Most useful C or C++ programs are written in separate *modules*
- Each module corresponds to:
  - one `.c`, `.C`, `.cc`, `.cpp`, or `.cxx` file (lets call this the *source file*), and
  - one `.h`, `.H`, `.hh`, or `.hpp` file (we call this the *header file*).
- Each header file lists the publicly exported names: type definitions, global variables, and function prototypes
  - Avoid defining variables or functions in header files
- It makes this set of names as small as possible to *reduce the interaction between modules*
- All non-public functions defined in the source files are defined with the keyword **static** so that they cannot be linked to from other modules.

Modules — silly example

main.cpp:

```
#include <iostream>
#include "calc.h"
#include "main.h"

int glob;

int main( void )
{
        glob = 10;
        int sum = calc( 15 );
        std::cout << "sum = " << sum << '\n';
}
```

main.h:

```
#ifndef MAIN_H
#define MAIN_H

extern int glob;

#endif
```

Modules — silly example (continued)

calc.cpp:

```
#include "main.h"
#include "calc.h"

int calc( int n )
{
        return glob + n;
}
```

calc.h:

```
#ifndef CALC_H
#define CALC_H

extern int calc( int n );

#endif
```


static variables

- A variable defined with the keyword **static** is visible only within its file, and does not conflict with a variable with the same name defined in another file
- the **static** keyword can (and should) be used with functions that are to be used only within one file
- The **static** keyword can be used inside functions and blocks
  - It is initialised once, and its value remains even between function calls. See Program `static.cpp`:

```
#include <iostream>
void show_times_called( void )
{
        static int called = 0;
        std::cout << "called " << ++called << " times\n";
}
int main( void )
{
        for ( int i = 0; i < 5; ++i )
                show_times_called();
}
```

- output of `static.cpp`:

```
called 1 times
called 2 times
called 3 times
called 4 times
called 5 times
```

16 Some Things to Read

References

[K&R] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall 1988.

[1] Stanley B. Lippman and Josée Lajoie and Barbara E. Moo. *C++ Primer, Fourth Edition*. Addison-Wesley 2005.

[2] Bjarne Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley, 2004, ISBN 0201889544.

[3] Tom Adamson and James L. Antonakos and Kenneth C. Mansfield Jr. *Structured C for Engineering and Technology, Third Edition*. Prentice Hall, 1998.

[4] Steve Oualline. *Practical C Programming*. O'Reilly 1993.

[5] Paul Davies. *The Indispensable Guide to C With Engineering Applications*. Addison-Wesley 1995.

[6] H. M. Deitel and P. J. Deitel. *C How to Program, Second Edition*. Prentice Hall 1994.

[7] Vincent Kassab. *Technical C Programming*. Prentice Hall 1989.

[8] Marshall Cline. *C++ FAQ LITE*. `http://www.parashift.com/c++-faq-lite/`

[9] Bjarne Stroustrup. *A Tour of the Standard Library*. Chapter 3 of *The C++ Programming Language*. `http://public.research.att.com/~bs/3rd_tour2.pdf`

License covering this document