Programming Fundamentals - I

Basic Concepts

Fall-Semester 2016

Prepared By:

Rao Muhammad Umer

Lecturer,

Web: raoumer.github.io

Department of Computer Science & IT,

The University of Lahore.

What is computer?

The term "computer" was originally given to humans who performed numerical calculations using mechanical calculators, such as the abacus and slide rule. The term was later given to a mechanical device as they began replacing the human computers.

Today's computers are electronic devices that accept data such as numbers, text, sound, image, animations, video, etc., (input), process that data (converts data to information) , produce output, and then store (storage) the results.

A basic computer consists of 4 components:

1. Input devices
2. Central Processing Unit or CPU
3. Output devices
4. Memory

Input Devices are used to provide input to the computer basic input devices include keyboard, mouse, touch screens etc.

Central Processing Unit acts like a brain, it processes all instructions and data in the computer, the instructions are computer commands, these commands are given to CPU by input devices, some of the instructions are generated by the computer itself

Output devices are used to receive computer output, the output, some basic output devices are hard drive disk (HDD, commonly known as hard disk), printers, computer screens (Monitors and LCDs)

The computer memory is a temporary storage area. It holds the data and instructions that the Central Processing Unit (CPU) needs. Before a program can be run, the program is loaded from some storage device such as into the memory, the CPU loads the program or part of the program from the memory and executes it.

Difference between memory and storage

A storage device keeps the data for long times, the data is not lost when the storage device or computer is off, however the memory holds data for short intervals and this data is lost if the computer is turned off. Storage devices are much slower than computer memory. To clarify the difference consider storage devices as file cabinets (common in offices) and memory as work desk (table usually). The file cabinet provides storage for all the files and information you need in your office. When you come in to work, you take out the files you need from storage and put them on your desk for easy access while you work on them. The desk is like memory in the computer: it holds the information and data you need to have handy while you're working. Imagine what it would be like if every time you wanted to look at a document or folder you had to retrieve it from the file drawer. It would slow you down tremendously, same is the case with storage and memory, as CPU works at high speeds it needs fast access to programs, that is why programs and data are first loaded to computer memory and then CPU works on them.

Basic Components of Computer

- Hardware
- Software
    - Operating System
    - Application Programs

What is computer hardware?

Computer hardware refers to the physical parts or components of a computer such as the monitor, mouse, keyboard, computer data storage, hard drive disk (HDD), system unit (sound cards, memory, motherboard and chips), etc. all of which are physical objects that can be touched (known as tangible).

What are computer software / programs?

Software is any set of instructions that tells the hardware what to do. It is what guides the hardware and tells it how to accomplish each task. Some examples of software include web browsers, games, and word processors such as Microsoft Word. Program or computer program is a commonly used term for software.

What are Application Software / Programs?

An application program (sometimes shortened to application) is any program designed to perform a specific function directly for the user or, in some cases, for another application program. Examples of application programs include word processors; calculator, calendar, database programs; Web browsers; development tools; drawing, paint, and image editing programs; and communication programs. Application programs use the services of the computer's operating system and other supporting programs.

What is Operating System?

A software or program that manage computer resources including software and hardware such as hard drive disk, memory, printer, or application programs, it allows software and hardware to communicate with each other. Most of the time, there are many different programs running at the same time, and they all need to access your computer's central processing unit (CPU), memory, and storage. The operating system coordinates all of this to make sure each program gets what it needs. Without the operating system, the software wouldn't even be able to talk to the hardware, and the computer would be useless. The operating system acts as an interface between the hardware and the programs and allows you to communicate with the computer without knowing how to speak the computer's "language."

In simple terms, an operating system is a manager. It manages all the available resources on a computer. These resources can be the hard disk, a printer, or the monitor screen. Even memory is a resource that needs to be managed. The three most common operating systems for personal computers are Microsoft Windows, Mac OS X, and Linux however there are hundreds of operating system in the world. Some operating systems are for personal computers while others are for different types of computers and devices such as mobile phones, main-frame computers, super computers etc.

Types of Operating Systems

1. **Time-sharing operating systems**

Time sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Processor's time which is shared among multiple users simultaneously is termed as time-sharing. Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently that the user feels an immediate response.

Advantages of Timesharing operating systems are following

- Provide advantage of quick response.
- Avoids duplication of software.
- Reduces CPU idle time.

Disadvantages of Timesharing operating systems are following.

- Problem of reliability.
- Question of security and integrity of user programs and data.
- Problem of data communication.

2. Distributed operating System

Distributed systems use multiple central processors to serve multiple real time application and multiple users. Data processing jobs are distributed among the processors accordingly to which one can perform each job most efficiently.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers and so on.

The advantages of distributed systems are following.

- With resource sharing facility user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.

- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

3. Network operating System

Network Operating System runs on a server and provides server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. Examples of network operating systems are Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are following.

- Centralized servers are highly stable.
- Security is server managed.
- Upgrades to new technologies and hardwares can be easily integrated into the system.
- Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are following.

- High cost of buying and running a server.
- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

4. Real Time operating System

Real time system is defines as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. Real time processing is always on line whereas on line system need not be real time. The time taken by the system to

respond to an input and display of required updated information is termed as response time. So in this method response time is very less as compared to the online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. Real-time operating system has well-defined, fixed time constraints otherwise system will fail. For example Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, and home-appliances controllers, Air traffic control system etc. Windows CE, OS-9, Symbian and LynxOS are some of the commonly known real-time operating systems.

Multi-user and Single-user Operating Systems:

Computer operating systems of this type allow multiple users to access a computer system simultaneously. Time-sharing systems can be classified as multi-user systems as they enable a multiple user access to a computer through time sharing. Single-user operating systems, as opposed to a multi-user operating system, are usable by only one user at a time. Being able to have multiple accounts on a Windows operating system does not make it a multi-user system. Rather, only the network administrator is the real user. Windows 95, Windows 2000, Mac OS and Palm OS are examples of single-user operating systems. Unix and OpenVMS are examples of multi-user operating systems.

Multi-tasking and Single-tasking Operating Systems:

When a single program is allowed to run at a time, the system is grouped under the single tasking system category, while in case the operating system allows for execution of multiple programs or tasks at a time, it is classified as a multi-tasking operating system. Palm OS for Palm handheld is a single-task operating system. Windows 9x support multi-tasking. DOS Plus is a relatively less-known multi-tasking operating system. It can support the multi-tasking of a maximum of four CP/M-86 programs.

Embedded System:

The operating systems designed for being used in embedded computer systems are known as embedded operating systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design.

Windows CE, FreeBSD and Minix 3 are some examples of embedded operating systems. The use of Linux in embedded computer systems is referred to as Embedded Linux.

Mobile Operating System:

Though not a functionally distinct kind of operating system, mobile OS is definitely an important mention in the list of operating system types. A mobile OS controls a mobile device and its design supports wireless communication and mobile applications. It has built-in support for mobile multimedia formats. Tablet PCs and smartphones run on mobile operating systems.

Blackberry OS, Google's Android and Apple's iOS are some of the most known names of mobile operating systems.

Batch Processing and Interactive Systems:

Batch processing refers to execution of computer programs in 'batches' without manual intervention. In batch processing systems, programs are collected, grouped and processed on a later date. There is no prompting the user for inputs as input data are collected in advance for future processing. Input data are collected and processed in batches, hence the name batch processing. IBM's z/OS has batch processing capabilities. As against this, interactive operating requires user intervention. The process cannot be executed in the user's absence.

How computer programs are created?

Computer programs / software are created by programming languages, a programming language provides set of instructions that human beings can understand, these instructions can be used to instruct computer to perform some useful work, such as mathematical calculations. To create a

program these instructions are written according to certain rules and in some order. A special program is used to convert these instruction into machine-readable form so that computer can understand and execute these commands. There are two types of such conversion programs i.e. compiler and interpreter.

What programs are used to develop a program?

There are many programming languages, and for each programming language there are many software that help in writing, testing and executing a program. Following is a list of basic programs that are required to write a program using any programming language

***1-Editor*:** Editors are simple software that allows a user to create simple text files, a computer program can be written in a text file, the program in this form is called "source code".

***2-Compiler*:** Compilers convert source code into machine code, this is required as computer understands only machine code. For every language there must be a compiler or interpreter. After compilation the source code becomes "executable code".

***3-Interpreter*:** An interpreter also converts the source code into machine code however it is different from a compiler, a compiler takes source code and converts all of it to machine code in one translation or conversion, an interpreter takes pieces from source code converts it to machine code, an interpreter does not converts all of the source code to machine code in one translation.

4-Debugger: A debugger is a software program used to test and find bugs (errors) in other programs, this helps the programmer to understand and trace program errors more easily.

Writing or creating, writing a or developing a program have same meanings, the person who creates or writes computer programs is called programmer or software developer (commonly used term is programmer and developer)

What is an IDE?

The term IDE stands for Integrated Development Environment; the term is used for software that provides a collection of necessary programs to develop a computer program. An IDE usually have an editor, compiler or debugger, and other programming tools.

What is Programming Language?

A "programming language" is a language designed to describe a set of consecutive actions to be executed by a computer. A programming language is therefore a practical way for us (humans) to give instructions to a computer.

Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.

Thousands of different programming languages have been created, mainly in the computer field, and many more still are being created every year. Many programming languages require computation to be specified in an imperative form (i.e., as a sequence of operations to perform), while other languages utilize other forms of program specification such as the declarative form (i.e. the desired result is specified, not how to achieve it).

Languages that computers use to communicate with each other, have nothing to do with programming languages, they are referred to as communication protocols, these are two very different concepts. A programming language is very strict:

> "EACH instruction corresponds to ONE processor action."

The language used by the processor is called machine code. The code that reaches the processor consists of a series of 0s and 1s known as (binary data).

Machine code is therefore difficult for humans to understand, which is why intermediary languages, which can be understood by humans, have been developed. The code written in this type of language is transformed into machine code so that the processor can process it.

The assembler was the first programming language ever used. This is very similar to machine code but can be understood by developers. Nonetheless, such a language is so similar to machine code that it strictly depends on the type of processor used (each processor type may have its own machine code). Thus a program developed for one machine may not be ported to

another type of machine. The term "portability" describes the ability to use a software program on different types of machines. A software program written in assembler code, may sometimes have to be completely rewritten to work on another type of computer!

A programming language has therefore several advantages:

- It is much more understandable than machine code;
- It allows greater portability, i.e. can be easily adapted to run on different types of computers.

Programming languages enable users to write programs for specific computations/algorithms.

Imperative and functional programming languages

Programming languages are generally divided into two major groups according to how their commands are processed:

- Imperative languages
- Functional languages

Imperative programming language

An imperative language programs using a series of commands, grouped into blocks and comprising of conditional statements which allow the program to return to a block of commands if the condition is met. These were the first programming languages in use, even today many modern languages still use this principle.

Structured imperative languages suffer, however, from lack of flexibility due to the sequentially of instructions.

Functional programming language

A functional programming language (often called procedural language) is a language which creates programs using functions, returning to a new output state and receiving as input the result of other functions. When a function invokes itself, we refer to this as recursion.

Interpretation and compilation

Programming languages may be roughly divided into two categories:

- Interpreted languages
- Compiled languages

Interpreted language

A programming language is by definition different to machine code This must therefore be translated so that the processor can understand the code. A program written in an interpreted language requires an extra program (the interpreter) which translates the programs commands as needed.

Compiled language

A program written in a "compiled" language is translated by an additional program called a compiler which in turn creates a new stand-alone file which does not require any other program to execute itself, such a file is called an executable.

A program written in a compiled language has the advantage of not requiring an additional program to run it once it has been compiled. Furthermore, as the translation only needs to be done once, at compilation it executes much faster.

However, it is not as flexible as a program written in an interpreted language, as each modification of the source file (the file understandable by humans: the file to be compiled) means that the program must be recompiled for the changes to take effect.

On the other hand, a compiled program has the advantage of guaranteeing the security of the source code. In effect, interpreted language, being a directly legible language, means that anyone can find out the secrets of a program and thus copy or even modified the program. There is therefore a risk of copyright violation. On the other hand, certain secure applications need code confidentiality to avoid illegal copying (bank transactions, on-line payments, secure communications...).

Intermediary languages

Some languages belong to both categories (LISP, Java, Python...) as the program written in these languages may in certain cases undergo an intermediary compilation phase, into a file written in a language different to the source file and non-executable (requiring an interpreter). Java applets, small programs, often loaded in web pages, are compiled files, which can only be executed from within a web browser (these are files with the .class extension). Basic Concepts of any Programming Language

There are five basic concepts of any programming languages, as given below:

1. **Variables**

   In computer programming, a variable is a storage location and an associated symbolic name which contains some known or unknown quantity or information, a value.

2. **Control Structures**

   A control structure is a block of programming that analyzes variables and chooses a direction in which to go based on given parameters. The term flow control details the direction the program takes (which way program control "flows"). Hence it is the basic decision-making process in computing; flow control determines how a computer will respond when given certain conditions and parameters.

3. **Data Structures**

   In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

4. **Syntax**

   In computer science, the syntax of a programming language is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language.

5. **Tools**

   In the real world, a tool is something (usually a physical object) that allows you to get a certain job done in a more timely manner. Well, this holds true with the programming world too. A tool in programming is a piece of software that, when used while you code, allows you to get your program done faster!

Basic Concepts of C++ Language

C is a general-purpose high level language that was originally developed by Dennis Ritchie for the UNIX operating system. It was first implemented on the Digital Equipment Corporation PDP-11 computer in 1972. C++ is the improved version of C. Since C has only support of procedural language, while, C++ has support of both Procedural and Object Oriented Programming language.

The UNIX operating system and virtually all Unix applications are written in the C as well as C++ language. C++ has now become a widely used professional language for various reasons.

- Easy to learn
- OOP language
- It produces efficient programs.
- It can handle low-level activities.
- It can be compiled on a variety of computers.

Why to use C++?

C++ was initially used for system development work, in particular the programs that make-up the operating system. C++ was adopted as a system development language because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C++ might be:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Modern Programs
- Data Bases
- Language Interpreters
- Utilities

C++ Program File

All the C++ programs are written into text files with extension ".cpp" for example hello.cpp.

C++ Compilers

When you write any program in C++ language then to run that program you need to compile that program using a C++ Compiler which converts your program into a language understandable by a computer called machine language.

Variables

A variable is a symbolic name for (or reference to) information. The variable's name represents what information the variable contains. They are called variables because the represented information can change but the operations on the variable remain the same. In general, a program should be written with "Symbolic" notation, such that a statement is always true symbolically. For example if I want to know the average of two grades, We can write "average = (grade_1 + grade_2) / 2.0;" and the variable average will then contain the average grade regardless of the scores stored in the variables, grade_1 and grade_2.

Variables in a computer program are analogous to "Buckets" or "Envelopes" where information can be maintained and referenced. On the outside of the bucket is a name. When referring to the bucket, we use the name of the bucket, not the data stored in the bucket.

Variables are "Symbolic Names". This means the variable "stands in" for any possible values. This is similar to mathematics, where it is always true that if given two positive numbers (lets use the symbols 'a' and 'b' to represent them):

$a + b > a$

(i.e., if you add any two numbers, the sum is greater than one of the numbers by itself).

This is called Symbolic Expression, again meaning, when any possible (valid) values are used in place of the variables, the expression is still true.

Another example, if we want to find the sum of ANY TWO NUMBERS we can write:

result = a + b;

Both 'a' and 'b' are variables. They are symbolic representations of any numbers. For example, the variable 'a' could contain the number 5 and the variable 'b' could contain the number 10. During execution of the program, the statement "a + b" is replaced by the Actual Values "5 + 10" and the result becomes 15. The beauty (and responsibility) of variables is that the symbolic operation should be true for any values.

Another example, if we want to find out how many centimeters tall a person is, we could use the following formula: height in centimeters = height in inches * 2.54.

This formula is always true. It doesn't matter if its Joe's height in inches or Jane's height in inches. The formula works regardless. In computer terminology we would use:

```
height_in_centimeters = height_in_inches * 2.54;

// the variable height_in_centimeters is assigned a
// new value based on the current value of "height_in_inches"
// multiplied by 2.54
```

Variable Properties

There are 6 properties associated with a variable. The first three are very important as you start to program. The last three are important as you advance and improve your skills (and the complexity of the programs you are creating).

Memorize These!

1. A Name
2. A Type
3. A Value
4. A Scope
5. A Life Time

6. A Location (in Memory)

Clarification of Properties

1. A Name

The name is Symbolic. It represents the "title" of the information that is being stored with the variable.
The name is perhaps the most important property to the programmer, because this is how we "access" the variable. Every variable must have a unique name!

2. A Type

The type represents what "kind" of data is stored with the variable. (See the Chapter on Data Types).

3. A Value

A variable, by its very name, changes over time. Thus if the variable is jims_age and is assigned the value 21. At another point, jims_age may be assigned the value 27.

4. A Scope

Good programs are "Chopped" into small self contained sections (called functions) much like a good novel is broken into chapters, and a good chapter is broken into paragraphs, etc. A variable that is seen and used in one function is NOT available in another function. This allows us to reuse variable names, such as age. In one function 'age' could refer to the age of a student, and in another function 'age' could refer to the vintage of a fine wine.

Further this prevents us from "accidentally" changing information that is important to another part of our program.

5. A Life Time

The life time of a variable is strongly related to the scope of the variable. When a program begins, variables "come to life". Variables "die" when the program leaves the "Scope" of the variable.

6. A Location (in Memory)

Luckily, we don't have to worry too much about where in the computer hardware the variable is stored. The computer does this for us. But you should be aware that a "Bucket" or "Envelope" exists in the hardware for every variable you declare.

Legal Variable Names

- Start with a letter
- Use _ (underscores) for clarity
- Can use numbers
- Don't use special symbols
- Don't have spaces
- Have meaningful Names

Good Variable Names

Good variable names tell you, your teammates, (and your TA) what information is "stored" inside that variable. They should make sense to a non computer programmer. For example:

```
g = 9.81; // bad

gravitational_constant = 9.81;  //good
```

The different kinds of files

Compiling C++ programs requires you to work with four kinds of files:

1. Regular **source code** files. These files contain function definitions, and have names which end in "`.cpp`" by convention.
2. **Header** files: These files contain function declarations (also known as function prototypes) and various preprocessor statements. They are used to allow source code files to access externally-defined functions. Header files end in "`.h`" by convention.
3. **Object** files: These files are produced as the output of the compiler. They consist of function definitions in binary form, but they are not executable by themselves. Object files end in "`.o`" by convention, although on some operating systems (*e.g.* Windows, MS-DOS), they often end in "`.obj`".
4. **Binary executables**: These are produced as the output of a program called a "linker". The linker links together a number of object files to produce a binary file which can be directly executed. Binary executables have no special suffix on Unix operating systems, although they generally end in "`.exe`" on Windows.

There are other kinds of files as well, notably libraries ("`.a`" files) and shared libraries ("`.so`" files), but you won't normally need to deal with them directly.

The preprocessor

Before the C++ compiler starts compiling a source code file, the file is processed by a *preprocessor*. This is in reality a separate program (normally called "cpp", for "C++ preprocessor"), but it is invoked automatically by the compiler before compilation proper begins. What the preprocessor does is convert the source code file you write into another source code file (you can think of it as a "modified" or "expanded" source code file). That modified file may exist as a real file in the file system, or it may only be stored in memory for a short time before being sent to the compiler. Either way, you don't have to worry about it, but you do have to know what the preprocessor commands do.

Preprocessor commands start with the pound sign ("#"). There are several preprocessor commands; two of the most important are:

1. **#define**. This is mainly used to define constants. For instance,

2. #define BIGNUM 1000000

   specifies that wherever the character string BIGNUM is found in the rest of the program, 1000000 should be substituted for it. For instance, the statement:

   int a = BIGNUM;

   becomes

   int a = 1000000;

   #define is used in this way so as to avoid having to explicitly write out some constant value in many different places in a source code file. This is important in case you need to change the constant value later on; it's much less bug-prone to change it once, in the #define, than to have to change it in multiple places scattered all over the code.

3. **#include**. This is used to access function definitions defined outside of a source code file. For instance:
4. #include <iostream>

   causes the preprocessor to paste the contents of <iostream> into the source code file at the location of the #include statement before it gets compiled. #include is almost always used to include header files, which are files which mainly contain function declarations and #define statements. In this case, we use #include in order to be able to use functions such as cout and cin, whose declarations are located in the file iostream. C++ compilers do not allow you to use a function unless it has previously been declared or defined in that file; #include statements are thus the way to re-use previously-written code in your C++ programs.

There are a number of other preprocessor commands as well, but we will deal with them as we need them.

Making the object file: the compiler

After the C++ preprocessor has included all the header files and expanded out all the #define and #include statements (as well as any other preprocessor commands that may be in the original file), the compiler can compile the program. It does this by turning the C++ source code into an **object code** file, which is a file ending in ".o" which contains the binary version of the source code. Object code is not directly executable, though. In order to make an executable, you also have to add code for all of the library functions that were #included into the file (this is not the same as including the declarations, which is what #include does). This is the job of the linker.

In general, the compiler is invoked as follows:

```
% gcc  -c  foo.c
```

where % is the unix prompt. This tells the compiler to run the preprocessor on the file foo.c and then compile it into the object code file foo.o. The -c option means to compile the source code file into an object file but not to invoke the linker. If your entire program is in one source code file, you can instead do this:

```
% gcc  foo.c  -o  foo
```

This tells the compiler to run the preprocessor on foo.c, compile it and then link it to create an executable called foo. The -o option states that the next word on the line is the name of the binary executable file (program). If you don't specify the -o, *i.e.* if you just type gcc foo.c, the executable will be named a.out for silly historical reasons.

Note also that the name of the compiler we are using is gcc, which stands for "GNU C compiler" or "GNU compiler collection" depending on who you listen to. Other C compilers exist; many of them have the name cc, for "C compiler". On Linux systems cc is an alias for gcc.

Putting it all together: the linker

The job of the linker is to link together a bunch of object files (.o files) into a binary executable. This includes both the object files that the compiler created from your source code files as well as object files that have been pre-compiled for you and collected into **library files**. These files have names which end in .a or .so, and you normally don't need to know about them, as the linker knows where most of them are located and will link them in automatically as needed.

Like the preprocessor, the linker is a separate program called ld. Also like the preprocessor, the linker is invoked automatically for you when you use the compiler. The normal way of using the linker is as follows:

```
% gcc  foo.o  bar.o  baz.o  -o  myprog
```

This line tells the compiler to link together three object files (foo.o, bar.o, and baz.o) into a binary executable file named myprog. Now you have a file called myprog that you can run and which will hopefully do something cool and/or useful.

This is all you need to know to begin compiling your own C programs. Generally, we also recommend that you use the -Wall command-line option:

```
% gcc  -Wall  -c  foo.cc
```

The -Wall option causes the compiler to warn you about legal but dubious code constructs, and will help you catch a lot of bugs very early. If you want to be even more anal (and who doesn't?), do this:

```
% gcc  -Wall  -Wstrict-prototypes  -ansi  -pedantic  -c foo.cc
```

The -Wstrict-prototypes option means that the compiler will warn you if you haven't written correct prototypes for all your functions. The -ansi and -pedantic options cause the compiler to warn about any non-portable construct (*e.g.* constructs that may be legal in gcc but not in all standard C compilers; such features should usually be avoided).