LIX, École Polytechnique

C++ Notes

Leo Liberti

Last update: January 12, 2008

Contents

1 Introduction

This is a very brief introduction to the C++ programming language, to be used in courses held at the Département d'Informatique and the Département de Mathématiques Appliquées of the École Polytechnique. The aim of the course is to teach the basics of the C++ language in a practical, hands-on way, without many of the more technical details concerning the more advanced topics. These notes should be paired with the associated presentation slides:

```
http://www.lix.polytechnique.fr/~liberti/teaching/c++/online/course.pdf
http://www.lix.polytechnique.fr/~liberti/teaching/c++/online/course-javausers.pdf
```

but most importantly with the exercise book (`http://www.lix.polytechnique.fr/~liberti/teaching/c++/online/exercises`), where most of the material is analyzed in depth. All the didactical content can be found online at `http://www.lix.polytechnique.fr/~liberti/teaching/c++/online`.

The main sources for these notes are:

1. Bjarne Stroustrup, *The C++ Programming Language*, 3rd edition, Addison-Wesley, Reading (MA), 1999
2. Stephen Dewhurst, *C++ Gotchas: Avoiding common problems in coding and design*, Addison-Wesley, Reading (MA), 2002
3. Herbert Schildt, *C/C++ Programmer's Reference*, 2nd edition, Osborne McGraw-Hill, Berkeley (CA).

2 Generalities

This C++ course is based on the GNU C++ compiler v. 4.0 and above running under the Unix operating system. Essential Unix tools, beside the compiler, are: `make`, `tar`, `gzip`/`gunzip`. As debuggers, we use `gdb` with the `ddd` front-end, and `valgrind`.

By "computer", here, we do *not* mean a theoretical computer science model such as a Turing machine, but a real, physical desktop or laptop computer. Although any architecture capable of running a Unix and the above-mentioned tools will do, the actual C++ course is given on desktop PCs with a 32 bit Intel processor running a mainstream Linux distribution.

2.1 Definitions

The CPU (Central Processing Unit) of a computer is where all the logical and arithmetic tests, loops and decisions take place, and where control commands and data exchanges are issued to devices such as memory, disk, screen, etc. The behaviour of the CPU is determined by its state, which is described by the content of all its registers and internal memory caches. Letting S be the set of possible CPU states, the CPU acts like a deterministic function $f : S \to S$. According to this model, to each state $s \in S$ there corresponds a next state $s' = f(s)$. The rate at which the CPU changes state is governed by the system clock (usual rates are between 1 and 3 GHz). Thus, around every billionth of a second, the CPU changes its state.

The form of the function f obviously depends on the CPU make and model. CPUs usually contain some extremely fast but very small memory chunks called "registers" which are specifically designed to store either values or memory addresses. The state of the CPU at each clock tick is then determined

by the values contained in each of its registers. The CPU is designed in such a way that at each clock tick the memory address contained in a certain register will automatically be incremented, and the value contained at the new address is read and interpreted as a "machine code instruction". This allows us to interpret the function f in a different way: we can consider the next state s' of the CPU as given by a function $p : I \times S \to S$ with $s' = p(i, s)$, where i is a machine code instruction in the set I of all possible CPU instructions. Although each basic instruction in I is rather simple, this interpretation of f makes it possible to group several simple instructions into more complex ones[1]. As some of the instructions concern logical tests and loops, it becomes apparent that the full semantics of any modern computer language (including C++) can indeed be exploited by a CPU after a suitable transformation of the complex, high-level language into the simple machine code instruction set I.

Loosely speaking, the set I can be partitioned in the following instruction categories.

- *Input*: transfer data from external device to processor
- *Output*: transfer data from processor to external device
- *Storage*: transfer data from processor to memory
- *Retrieval*: transfer data from memory to processor
- *AL operation*: perform arithmetic/logical operation on data
- *Test*: verify condition on data and act accordingly
- *Loop*: repeat a sequence of operations

In practice, these instructions are encoded in machine language, i.e. sequences of bits. The length of each instruction depends on the width of the CPU registers. The width of each register is measured in terms of the amount of BInary digiTs $\{0, 1\}$ (bits) it can contain. Traditionally, on Intel 16-bit architectures (32- and 64- bit architectures are evolutions thereof, and each new version is guaranteed to retain backward compatibility) there are four general-purpose registers: AX (accumulator), BX (base), CX (counter), DX (data); four pointer registers: SI (source index), DI (destination index), BP (base pointer), SP (stack pointer); four segment registers: CS (code segment), DS (data segment), ES (extra segment), SS (stack segment); and finally, one instruction pointer IP. The machine code instruction i loaded at each clock tick to compute $s' = p(i, s)$ is the value found at the address CS:IP. More information can be found at `http://www.ee.hacettepe.edu.tr/~alkar/ELE414/` and `http://ourworld.compuserve.com/homepages/r_harvey/doc_cpu.htm`.

Consider now the following (informal) definitions:

- *Program*: set of instructions that can be interpreted by a computer
- *Instructions*: well-formed sequences of characters (syntax)
- *Interpretation*: sequence of operations performed by the computer hardware (semantics)
- *Programming language*: set of rules used to form valid instructions
- *Algorithm*: a program which terminates (though sometimes find "non-terminating algorithm" with abuse of notation)

The well-formedness of the sequence of characters in each instruction corresponds to the C++ syntax which is one of the subjects of these notes, and will therefore be explained in more detail later. The same holds for the semantics of each C++ program.

[1] We use the terms "simple" and "complex" here with their natural English language meanings. In CPU-related technical language, "simple instruction set" and "complex instruction set" have a very specific meaning (look for "RISC" and "CISC" on Google for more information on this point).

2.2 Data

Computers can only perform arithmetic and logic operations in the field $\mathbb{F}_2 = \{0, 1\}$. A set of 8 bits is called a *byte*. The 8086 and 80286 had a 16-bit bus (the *bus* is the data transfer capacity per clock tick); modern Intel-based processors have 32- or 64-bit wide buses. An n-bit wide memory chunk can hold 2^n different values. These are either indexed from -2^{n-1} to $2^{n-1} - 1$ or from 0 to $2^n - 1$. An integer in C/C++ on an Intel architecture is usually stored in 32 bits (=4 bytes). Floating point values are stored (usually in 8 bytes) according to a specific code. Any type of data that can be represented by an encoding of finite-sized integers can be stored in a computer memory. In C/C++ data are categorized according to their *types*. Data types can be *elementary*: `bool` (C++ only), `char`, `short`, `int`, `long`, `float`, `double` or *user-defined*: `union`, `struct`, `class` (C++ only). A user-defined data type is a finite sequence of existing data types and occupies either the sum (`struct`, `class`) or the maximum (`union`) of the sizes of the sequence elements. Memory addresses, by contrast, normally occupy a fixed amount of storage (32 bits in 32-bit CPUs).

2.3 Memory

Computers rely on different types of memory to store data (this means: values and addresses). Read-Only Memory (ROM) can only be read. Random-Access Memory (RAM) can be read and written to. Other types of memory include EEPROMs, Flash RAM, disks, CD-ROMs, tapes and so on. Within the scope of these notes, we shall only be concerned with RAM type memory. This is very fast (in modern PCs RAM chips are refreshed at frequencies that are around one tenth as that of the CPU), comparatively expensive, and loses information if the power is switched off. The model of memory that best fits these C++ notes is that of a linear array, as depicted in Fig. 1.

Figure 1: A linear array model of memory: the i-th box has address a_i and contains value v_i.

RAM can be seen as a long (finite) list of boxes; the i-th box in the list is addressed by a value a_i and contains a value v_i. The size of an address is usually the same as the processor bus (normally 32 or 64 bit). The size of the value, on the other hand, can be anything depending on the type of value being stored.

2.4 The operating system

An *operating system* is a set of running programs that handle the interactions between the computer (in all its parts) and its users. The operating system assigns memory to each program and protects each assigned memory segment from unauthorized access (so that a malfunctioning program cannot bring the whole computer to a halt); it shares the CPU times amongst all the running processes; it operates all data exchanges with external devices (keybord, screen, network, printers, disks). Operating systems have three main parts.

- The *user interface*, or *shell*. It can be command-line driven or a Graphical User Interface (GUI).
- The *kernel*, containing the core functionalities.

- The *device drivers*, a set of programs each of which handles the data exchange with a particular device.

A *process* is a program being run. In practice, the computer keeps track of the machine code instruction being executed for each running process. The operating system kernel stores information on currently running processes on its process table (this lists for example the memory segments reserved for and by the process, the amount of CPU time used by the process and so on). Processes may in turn be segmented into *threads*, which can be seen as separate processes which do not have a separate entry on the process table.

Any time a C++ program is compiled and linked, an executable file consisting of the equivalent machine code instructions is created. Upon launching this executable, the operating system creates a new corresponding process in the process table, and then starts executing the machine code instructions during the CPU time slices allocated to it. By calling the operating system function `fork()`, a C/C++ program can create a new process in the process table with a copy of itself (this means that at the clock tick immediately after the `fork` call has finished, the two processes are indistinguishable, save for a different process ID and different allocated CPU time slices — later on, conditional instruction usually differentiates the two processes: this is the standard way of creating new processes from within a C/C++ program). Threads can be dealt with by linking against special libraries, but this is outside the scope of this document.

2.5 Program execution

When[2] a program is loaded into memory, it is organized into three areas of memory, called segments: the text segment, stack segment, and heap segment. The text segment (sometimes also called the code segment) is where the compiled code of the program itself resides. This is the machine language representation of the program steps to be carried out, including all functions making up the program, both user defined and system.

The remaining two areas of system memory is where storage may be allocated by the compiler for data storage. The stack is where memory is allocated for variables within functions. A stack is a Last In First Out (LIFO) storage device where new storage is allocated and deallocated at only one "end" (the top of the stack).

Every C/C++ program begins executing at a function called `main()`: space is allocated on the stack for all variables declared within `main()`. If `main()` calls a function, say `myFunction()`, additional storage is allocated for the variables in `myFunction()` at the top of the stack. Notice that the parameters passed by `main()` to `myFunction()` are also stored on the stack. If `myFunction()` were to call any additional functions, storage would be allocated at the new top of stack. When `myFunction()` returns, storage for its local variables is deallocated, and the top of the stack is resumed at the old position. As can be seen, the memory allocated in the stack area is used and reused during program execution. It should be clear that memory allocated in this area will contain garbage values left over from previous usage.

The heap segment provides more stable storage of data for a program; memory allocated in the heap remains in existence for the duration of a program. Manually allocated memory, global and static variables are allocated on the heap.

On many architectures, including Intel-based PCs, the relative order of text and stack segment is such that the text segment is placed after the stack segment. This means that if one uses a variable stored on the stack to hold an array of bytes longer than the stack itself, one ends up overwriting parts of the text segment. Since the text segment contains the executable machine code instructions, it is theoretically possible to change the behaviour of the computer by simply entering some meaningful data in a variable.

[2]Parts of this section are taken from `http://www-ee.eng.hawaii.edu/~tep/EE160/Book/chap14/subsection2.1.1.8.html`.

This was the technique used by some of the most spectacular network hacks of the eighties and nineties (see `http://insecure.org/stf/smashstack.html`).

The death of a process can occur by explicit program termination (returning from the `main()` function, use of the `exit()` system function) or by an operating system signal occurring because of some extraordinary condition. The most common are user intervention (`SIGINT`, `SIGKILL`) and runtime errors (`SIGSEGV` — segmentation violation — when the process tries to read or write memory that does not belong to it). When the process terminates, it is removed from the process table along with all the memory it occupies, and its ID becomes available for new processes.

2.6 The Unix shell

There are many types of shells in Unix. The most popular are `bash` and `tcsh`. Personally, I prefer the first one, but the École Polytechnique installs the second by default in the didactical computer rooms. In any case one can start either of them by simply opening a terminal and typing the corresponding name. These notes are based on the `bash` shell, but most notions will work on either.

Basic commands:

- `cd` *directoryName*: change working directory
- `pwd`: print the working directory
- `cat` *fileName*: display the (text) file *fileName* to standard output
- `mv` *file position*: move *file* to a new *position*: e.g. `mv /etc/hosts .` moves the file `hosts` from the directory `/etc` to the current working directory (`.`)
- `cp` *file position*: same as `mv`, but copy the file
- `rm` *file*: remove *file*
- `rmdir` *directory*: remove an empty *directory*
- `grep` *string file(s)*: look for a *string* in a set of *files*: e.g. `grep -Hi complex *` looks in all files in the current directory (`*`) for the string `complex` ignoring upper/lower case (`-i`) and displays the name of the file (`-H`) as well as the line where the match occurs.

Most Unix commands can be "chained": the output of a command is read as the input of the next.

- By default, unix tools send their output messages to the *standard output* stream (`stdout`) and their error messages to the *standard error* stream (`stderr`)
- Both streams can be redirected. E.g., to redirect both `stdout` and `stderr`, use:
  `sh -c 'command options arguments > outFile 2>&1'`
- The output stream of a command can become the input stream of the next command in a chain: e.g. `find ~ | grep \.cxx` finds all files with extension `.cxx` in all subdirectories of the home directory; the first command (`find`) sends a recursive list across subdirectories of the home directory (denoted by `~`) to `stdout`. This stream is transformed by the pipe character (`|`) in the standard input (`stdin`) stream of the following command (`grep`), which filters out all lines *not* containing `.cxx`.

3 Basic C++

A C++ code consists of symbols denoting types, variables and functions linked by logical, arithmetical, conditional and repetitional operations.

3.1 Types, objects, variables and pointers

The main C++ program entities are types and objects. As remarked in Sect. 2.2, types can be elementary or user-defined. Each type describes a category of different values, also called *objects*. In practice, a type is a record defining the size and memory layout of the objects of that type. Objects are values belonging to an existing type that reside in a well-defined memory area. Consider the two objects:

```
float myFloat;
int myInt;
```

The `myFloat` object is of type `float`. In most architectures, this occupies 4 bytes of RAM. `myInt` is an object of type `int` that also occupies 4 bytes. This means that both objects could be stored at the very same memory location; yet, the fact that their type is different makes it impossible to misinterpret `myFloat` as an integer or `myInt` as a float.

A *variable* is a symbol that is assigned a type and a memory area of the correct size. Although an object is usually attached to a variable, it need not be so: one could manually create an object in memory and then delete the variable without freeing the memory. The object still resides in memory but is not attached to any variable. A *pointer* is a particular type of variable, whose assigned memory contains an address pointing to another memory area where the actual object is kept. Since pointers allow user access to an arbitrary part of the memory, they wield an unlimited control over the machine. Use with care.

In Fig. 2, a new type `class MyClass` is first declared as consisting of a `char` and an `int`. An object `myObject` of type `MyClass` is defined, and its components `myChar` and `myInt` are assigned values '`x`' and `1` respectively. A pointer `ptrObject` of type `MyClass*` is then simultaneously declared and defined, and assigned the address where the object `myObject` is stored.

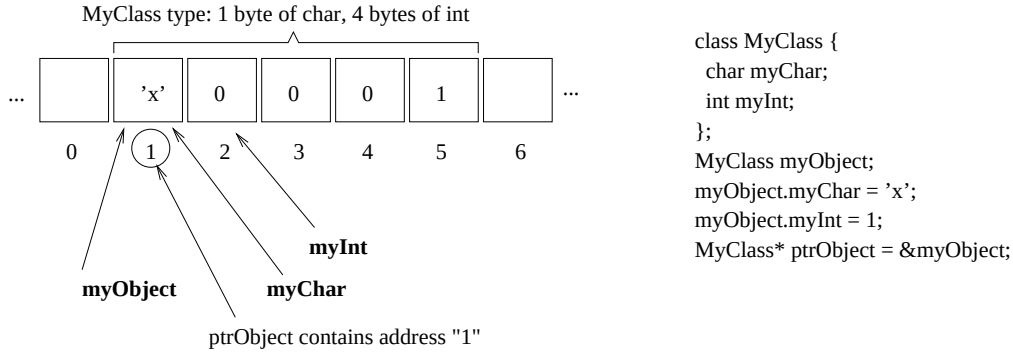

Figure 2: Types, objects, variables and pointers.

3.2 Preprocessing directives

A C++ program consists of several types of *preprocessing directives* and *statements*. Preprocessing directives are interpreted before compilation, must be limited to one line of text, are all prepended by

a hash character '#' and do *not* necessarily terminate with a semicolon character ';'. The purpose of preprocessing directives is mostly to enable conditional compilation and include other source files in the current compilation. Some examples:

```
#define MYCONSTANT 10
#include<iostream>
#include "myheader.h"
#ifdef DEBUG
  const char version[] = "Debug";
#else
  const char version[] = "Release";
#endif
```

The first line defines a constant called `MYCONSTANT` to take the value `10`. Each time the string "MYCONSTANT" appears in the rest of the program being compiled, it will be replaced by the string "10". The purpose of isolating constants to the beginning of the source file, where all the `#define`s are usually kept, is to avoid having to hunt for constants within the whole file each time the constant is changed. Although `#define`s are very popular in the C language, in C++ one usually uses equivalent constructs which are not preprocessing directives, such as

```
const int MYCONSTANT = 10;
```

which is a normal C++ statement. The second line instructs the compiler to look for a file called `iostream` within the compiler's *include search path* (i.e. a predetermined list of paths within the filesystem where a lot of include files are stored; typically, on Unix systems, this is `/usr/include:/usr/local/include`), read it, and compile it as if it was part of the source code being compiled. The third line is similar to the second but the slightly different syntax indicates that the file `myheader.h` is a user-created file and probably resides in the current directory. Lines 4-8 are a conditional preprocessing directive, instructing the compiler to define the character array `version` differently according as to whether the preprocessing symbol `DEBUG` was defined or not when the compiler was called. It is possible to define such symbols either explicitly, by including a preprocessor directive `#define DEBUG` before the conditional statement, or implicitly by launching the compiler from the shell with the `-DDEBUG` option.

3.3 Statements

The actual C++ program resides in the statements. Each statement can span multiple lines of texts. All statements are terminated by a semicolon character ';'. A sequence of statements within braces '{', '}' is a *block*. The *scope* of an instruction is the extent of the block it belongs to. Statements and block can be part of a *declaration* or a *definition*. Declarations specify a syntax, whereas definitions specify a semantics.

A declaration is used to create a new type or make the calling syntax of a function explicit. Examples:

```
class MyClass {
  char myChar;
  int myInt;
};
```

defines a new class called `MyClass`.

```
int myFunction(int myArgument);
```

defines calling syntax (also called *prototype*) of a function called `myFunction`, which is passed an `int` and returns an `int`. Declarations do not use memory and do not generate code; rather, they instruct the compiler on how handle variable symbols appearing subsequently in the code and how to issue function calls. Declaration blocks are terminated by a semicolon.

A definition specifies the meaning attached to a symbol. For a computer, the meaning of a symbol is given by what happens to the computer when the code relating to that particular symbol is executed. In other words, virtually all C++ code is part of a definition. For example:

```
MyClass* myPointer = &myObject;
```

is simultaneously a declaration and a definition. It is a declaration insofar as `myPointer` is a symbol being declared a pointer of type `MyClass*`. It is a definition because some memory of the correct size (i.e. enough to hold an address in memory, which on 32 bit architectures would be 4 bytes) is set aside and assigned to the symbol `myPointer`, and because that memory is filled with the address of memory where the object `myObject` is stored (see Fig. 2). As another example,

```
int myFunction(int myArgument) {
  int ret = myArgument * myArgument;
  return ret;
}
```

is a definition of the semantics of the function `myFunction`: it takes an integer argument, squares it, and returns it.

3.4 Memory allocation

The allocation of memory (i.e. obtaining some memory from the operating system) can be automatic or manual. Automatic allocation is done transparently by the program; in this case, deallocation (giving the memory back to the operating system) is automatic too, and occurs at the end of the scope. Automatic allocation always occurs on the stack. Manual allocation is done explicitly by the user, and deallocation must be done manually too. Manual allocation always occurs on the heap. Variables are always allocated automatically on the stack. The memory pointed to by pointers, however, is manually assigned to each pointer and may be manually allocated on the heap.

Examples.

1. The following code declares that `myChar` is a variable of type `char`, reserves 1 byte on the stack, and fills that byte with the 8-bit number 120 (the ASCII code for '`x`'). The allocated stack byte is deallocated (released) at the closing brace.

   ```
   {
     ...
     char myChar = 'x';
     ...
   }
   ```

2. The following code declares that `ptrChar` is a pointer of type `char*`, reserves 4 bytes on the stack, and fills those bytes with the 32-bit address of the memory where the value corresponding to the `myChar` variable is kept. The allocated stack space is released at the closing brace.

   ```
   {
   ```

```
  ...
  char* ptrChar = &myChar;
  ...
}
```

3. The first line of the following code declares that `ptrDouble` is a pointer of type `double*` and reserves 4 bytes on the stack (enough for a 32-bit memory address). The second line obtains 8 bytes of heap memory (enough to contain a `double`) and assigns its 32-bit address to the pointer `ptrDouble`. The third line writes the floating point number 0.2 into the heap memory area whose address is stored in `ptrDouble`. Notice that in order to actually use manually allocated heap memory, one needs to allocate some stack space to hold the address. The deallocation of the stack space occurs at the closing brace, but the deallocation of the heap space does not occur. In fact, this is an example of memory leak: we allocate some heap space, assign the address to a pointer, but the pointer is deallocated (and the address lost) before the heap space is deallocated; at this point, there is no way to deallocate the heap space, apart from the operating system automatically reclaiming the memory at the end of the execution.

```
{                              // 1
  double* ptrDouble;           // 2
  ptrDouble = new double;      // 3
  *ptrDouble = 0.2;            // 4
}                              // 5
```

4. This is like the example above, but revised so that memory leak does not occur. In line 2, a pointer `ptrOther` of type `double` is declared, and 4 bytes are allocated on the stack for holding a 32-bit address. Lines 4-6 are as above. In line 7 we copy the 32-bit address held in `ptrDouble` to `ptrOther` (this means that the heap memory address where the number 0.2 is stored is held in two different pointers). At line 8, as above, the stack space for `ptrDouble` is released and the address within it is lost. However, since `ptrOther` was allocated outside the scope being closed at line 8, `ptrOther` still exists after line 8: since `ptrOther` contains the address for the heap memory allocated in line 5, we can issue a manual `delete` command to release the heap memory, at line 9. Although the heap memory is released, the stack memory used for storing the `ptrOther` pointer is not: this is released at line 10 (the end of the scope).

```
{                              // 1
  double* ptrOther;            // 2
  {                            // 3
    double* ptrDouble;         // 4
    ptrDouble = new double;    // 5
    *ptrDouble = 0.2;          // 6
    ptrOther = ptrDouble;      // 7
  }                            // 8
  delete ptrOther;             // 9
}                              // 10
```

3.5 Bugs

When programming, it is virtually impossible not to make mistakes. Programming errors are called *bugs*. The etymology of the term "bug" is discussed in `http://en.wikipedia.org/wiki/Computer_bug`. Some of the bugs concern the syntax, and can be caught by the compiler during compilation (i.e. at *compile-time*). Others change the semantics of the program but do not invalidate the syntax. These bugs only manifest themselves during execution (i.e. at *run-time*) and are the hardest to trace.

Compile-time bugs are easy to detect (the program does not compile) but it may be hard to pinpoint the line of code where the bug actually is: although compile-time bugs invalidate the language syntax,

they may not invalidate the syntax at the line where they appear. For example, inserting a spurious open brace '{' on a line by itself is often perfectly legal. It is only at the end of the file that the compiler detects a mismatched number of brackets, and may not be able to trace the even to the spurious open brace. Compiler error messages often refer to the implications of the bug rather than the bug itself, so may be next to useless (or rather, they must be interpreted in the light of experience).

The run-time bugs that are hardest to trace are those that manifest themselves rarely and in nondeterministic ways. Since the computer is a deterministic machine, this sentence needs an explanation. One source of nondeterminism, for example, is given by the values stored in unused memory. Since memory is never implicitly initialized, these may be values left over from a terminated process, or simply random values from the computer power-on procedure. Since C++ pointers make it possible to read the whole computer memory, an erroneous use of pointers may yield random values nondeterministically. Nondeterministic runtime bugs are hard to trace because it is often impossible to recreate the conditions by which they occur.

I personally had two nightmarish nondeterministic runtime bug experiences, and both took me months to eradicate. The first one had something to do with the C++ Standard Template Library (STL) `sort()` algorithm with a user-defined "less than" relation. I had erroneously defined my less than as a $\leq$ relation instead of a strict ordering $<$ relation. Depending on the contents and memory position of the array, then, in a completely irreproducible fashion, sorting succeeded or terminated in a SIGSEGV abort. In the second one I was iterating over an STL `vector<int> v` as follows: `for(vector<int>::iterator vi = v.begin(); vi < v.end(); vi++) {...}`, mimicking the usual integer iteration `for(int i = 0; i < n; i++) { ...}`. STL iterators, however, are more like pointers than like integers, and STL vectors are not necessarily organized linearly in memory: which means that although logically the array element corresponding to the iterator `vi` may be *before* the end of the array (signalled by `v.end()`), the actual memory address contained in `vi` might be *after* it in the physical memory organization. Consequently, depending on the values contained in `v`, the iterator loop sometimes aborted before reaching the end of the vector. Save yourselves a lot of trouble by employing the correct syntax with a "different" (`!=`) instead of a "less than" (`<`) operator: `for(vector<int>::iterator vi = v.begin(); vi != v.end(); ++vi) {...}`.

As a last piece of advice when confronted with a bug that just wouldn't go away: if you spent weeks looking for a runtime bug without finding it, it probably means it's so simple it has escaped your attention. Question your simplest statements.

3.6 C++ Syntax

See `http://www.csci.csusb.edu/dick/c++std/cd2/gram.html` for an ANSI C++ grammar.

- boolean value: `bool` (1 bit), true or false
- ASCII character: `char` (1 byte), integer between -128 and 127
- integer number:
  - `int` (usually 4 bytes), between -2^{31} and $2^{31}-1$
  - `long` (usually 8 bytes)
  - can be prefixed by `unsigned`
- floating point: `double` (also `float`, rarely used)
- arrays:
  typeName variableName`[`*constArraySize*`] ;` `char myString[15];`

- pointers (a pointer contains a memory address): *typeName* `*` *pointerName* `;` `char* stringPtr;`

3.6.1 Declarations, assignments, tests, arithmetic/logical operations

- declaration: *typeName variableName* `;` `int i;`
- assignment: *variableName* `=` *expression* `;` `i = 0;`
- test:

```
if ( condition ) {
   statements ;
} else {
   statements ;
}
```

```
if (i == 0) {
   i = 1;
} else if (i < 0) {
   i = 0;
} else {
   i += 2;
}
```

- logical operators: and (`&&`), or (`||`), not (`!`)
  condition1 logical_op condition2 `;` `if (!(i == 0 || (i > 5 && i % 2 == 1))) { ...`
- arithmetic operators: `+`, `-`, `*`, `/`, `%`, `++`, `--`, `+=`, `-=`, `*=`, `/=`, ...

3.6.2 Loops

- loop (while):

```
while ( condition ) {
   statements ;
}
```

```
while (i < 10) {
   i = i + 1;
}
```

- loop (for):

```
for ( initial_statement ; condition ; itn_statement ) {
   statements ;
}
```

```
for (i = 0; i < 10; i++) {
   std::cout << "i = " << i << std::endl;
}
```

3.7 Functions

- function declaration: *typeName functionName*`(`*typeName1 argName1*, ...`);`

```
double performSum(double op1, double op2);
```

- function call: *varName* `= functionName(`*argName1*, ...`) ;`

```
double d = performSum(1.0, 2.1);
```

- return control to calling code: `return` *value* `;`

```
double performSum(double op1, double op2) {
   return op1 + op2;
}
```

3.7.1 Argument passing

- Arguments are passed from the calling function to the called function in two possible ways:
  1. by *value*
  2. by *reference*
- Passing by value (default): the calling function makes a copy of the argument and passes the copy to the called function; **the called function cannot change the argument**

```
double performSum(double op1, double op2);
```

- Passing by reference (prepend a &): the calling function passes the argument directly to the called function; **the called function can change the argument**

```
void increaseArgument(double& arg) { arg++; }
```

3.7.2 Overloading

- Different functions with the same name but different arguments: *overloading*
- Often used when different algorithms exist to obtain the same aim with different data types

```
void getInput(int theInput) {
   std::cout << "an integer" << std::endl;
}
void getInput(std::string theInput) {
   std::cout << "a string" << std::endl;
}
```

- Can be used in recursive algorithms to differentiate initialization and recursive step

```
void retrieveData(std::string URL, int maxDepth, Digraph& G,
        bool localOnly, bool verbose);
void retrieveData(std::string URL, int maxDepth, Digraph& G,
        bool localOnly, bool verbose,
        int currentDepth, VertexURL* vParent);
```

3.8 Pointers

- retrieve the address of a variable:

pointerName = &*variableName* ;

```
int* pi;
pi = &i;
```

- retrieve the value stored at an address:

variableName = **pointerName* ;

```
int j;
j = *i;
```

- using pointers as arrays:

```
const int bufferSize = 10;
char buffer[bufferSize] = "J. Smith";
char* bufPtr = buffer;
while(*bufPtr != ' ') {
   bufPtr++;
}
std::cout << ++bufPtr << std::endl;
```

3.8.1 Warnings

As already remarked, pointers allow direct access to the computer memory, and hence deliver an enormous power into the programmer's hands.

- Pointers allow you to access memory directly, hence can be very dangerous
- Attempted memory corruption results in "segmentation fault" error and abort, or garbage output, or unpredictable behaviour
- Most common dangers:
  1. writing to memory outside bounds
     ```
     char buffer[] = "LeoLiberti";
     char* bufPtr = buffer;
     while(*bufPtr != ' ') {
        *bufPtr = ' ';
        bufPtr++;
     }
     ```
  2. deallocating memory more than once
- Pointer bugs are usually very hard to track

3.9 Human-side C++ syntax

There are many different ways to write perfectly legal C++ programs. Some of these ways make the program clear when read by humans, others are less perspicuous. Within this document, we propose a programming style that should be **clear to read**. Code clarity is the main weapon against bugs.

Examples of particulary obscure code stylings can be found in the International Obfuscated C Code Contest (`http://www.ioccc.org/`). A 2004 finalist reads as follows:

```
                                #include\
                                <stdio.h>
                    #include                <stdlib.h>
                    #include                <string.h>

                   #define w "Hk~HdA=Jk|Jk~LSyL[{M[wMcxNksNss:"
                  #define r"Ht@H|@=HdJHtJHdYHtY:HtFHtF=JDBIl"\
                 "DJTEJDFIlMIlM:HdMHdM=I|KIlMJTOJDOIlWITY:8Y"
                #define S"IT@I\\@=HdHHtGH|KILJJDIJDH:H|KID"\
               "K=HdQHtPH|TIDRJDRJDQ:JC?JK?=JDRJLRI|UItU:8T"
              #define _(i,j)L[i=2*T[j,O[i=O[j-R[j,T[i=2*\
             R[j-5*T[j+4*O[j-L[j,R[i=3*T[j-R[j-3*O[j+L[j,
            #define t"IS?I\\@=HdGHtGIDJILIJDIItHJTFJDF:8J"

    #define y                   yy(4),yy(5),                yy(6),yy(7)
  #define yy(               i)R[i]=T[i],T[i ]            =O[i],O[i]=L [i]
#define Y _(O           ], 4] )_ (1 ], 5] )_ (2       ], 6] )_ (3 ], 7] )_=1
#define v(i)(      (( R[ i ] * _ + T [ i ]) * _ + O [ i ]) * _ + L [ i ]) *2
double b = 32  ,l ,k ,o ,B ,_ ; int Q , s , V , R [8 ], T[ 8] ,O [8 ], L[ 8] ;
#define q( Q,R ) R= *X ++ % 64 *8 ,R |= *X /8 &7 ,Q=*X++%8,Q=Q*64+*X++%64-256,
# define  p      "G\\QG\\P=GLPGTPGdMGdNGtOG1OG"   "dSGdRGDPGLPG\\LG\\LHtGHtH:"
#  define W         "Hs?H{?=HdGH|FI\\II\\GJlHJ"    "lFL\\DLTCMlAM\\@Ns}Nk|:8G"
# define   U           "EDGEDH=EtCElDH{^H|AJk}"       "Jk?LSzL[|M[wMcxNksNst:"
#  define u                  "Hs?H|@=HdFHtEI"             "\\HI\\FJLHJTD:8H"
char  *   x                   ,*X , ( * i )[               640],z[3]="4_",
*Z = "4,8O4.8O4G" r U "4M"u S"4R"u t"4S8CHdDH|E=HtAIDAIt@IlAJTCJDCIlKI\\K:8K"U
 "4TDdWDdW=D\\UD\\VF\\FFdHGtCGtEIDBIDDI1BIdDJT@JLC:8D"t"4UGDNG\\L=GDJGLKHL\
FHLGHtEHtE:"p"4ZFDTFLT=G|EGlHITBH|DIlDIdE:HtMH|M=JDBJLDKLAKDALDFKtFKdMK\
\\LJTOJ\\NJTMJTM:8M4aGtFGlG=G|HG|H:G\\IG\\J=G|IG|I:GdKGlL=G|JG|J:4b"W
S"4d"W t t"4g"r w"4iGlIGlK=G|JG|J:4kHl@Ht@=HdDHtCHdPH|P:HdDHdD=It\
BIlDJTEJDFIdNI\\N:8N"w"4lID@IL@=HlIH|FHlPH|NHt^H|^:H|MH|N=J\\D\
J\\GK\\OKTOKDXJtXItZI|YIlWI|V:8^4mHLGH\\G=HLVH\\V:4n" u t t
"4p"W"IT@I\\@=HdHHtGIDKILIJLGJLG:JK?JK?=JDGJLGI|MJDL:8M4\
rHt@H|@=HtDH|BJdLJTH:ITEI\\E=ILPILNNtCNlB:8N4t"W t"4u"
p"4zI[?Il@=HlHH|HIDLILIJDII|HKDAJ|A:JtCJtC=JdLJtJL\
THLdFNk|Nc|\
:8K"; main (
int C,char**        A) {for(x=A[1],i=calloc(strlen(x)+2,163840);
C-1;C<3?Q=_=       0,(z[1]=*x++)?((*x++==104?z[1]^=32:--x), X =
strstr(Z,z))      &&(X+=C++):(printf("P2 %d 320 4 ",V=b/2+32),
V*=2,s=Q=0,C     =4):C<4?Q-->O?i[(int)((l+=o)+b)][(int)(k+=B)
]=1:_?_-=.5/    256,o=(v(2)-(l=v(0)))/(Q=16),B=(v(3)-(k=v(1)
))/Q:*X>60?y   ,q(L[4],L[5])q(L[6],L[7])*X-61||(++X,y,y,y),
Y:*X>57?++X,  y,Y:*X >54?++X,b+=*X++%64*4:--C:printf("%d "
,i[Q][s]+i[Q ][s+1]+i[Q+1][s]+i[Q+1][s+1])&&(Q+=2)<V||(Q=
0,s+=2)<640
||(C=1));}
```

When compiled with the command `cc -o anonymous anonymous.c` and executed with:

```
./anonymous "ash nazg durhbatuluhk, ash nazg gimbatul, ash nazg thrakatuluhk, agh burzhumh-ishi krimpatul."
> anonymous.pgm
```

it produces an output file `anonymous.pgm` containing the graphics below.

3.9.1 Indentation

Indentation is a coding style that emphasizes the logical structure of blocks. It is the easiest way to prevent bugs. More details can be found in the exercise book.

- Not necessary for the computer
- **Absolutely necessary for the programmer / maintainer**
- After each opening brace {: new line and tab (2 characters)

- Each closing brace } is on a new line and "untabbed"

```
double x, y, z, epsilon;
...
if (fabs(x) < epsilon) {
   if (fabs(y) < epsilon) {
      if (fabs(z) < epsilon) {
         for(int i = 0; i < n; i++) {
            x *= y*z;
         }
      }
   }
}
```

Proper indentation can be obtained by editing source files with the GNU Emacs text editor.

- Traditional GNU/Linux text editor: `emacs` `emacs` *programName*`.cxx`
- Many key-combination commands (try ignoring menus!)
- Legenda: `C-`*key*: CTRL+*key*, `M-`*key*: ALT+*key* (for keyboards with no ALT key or for remote connections can obtain same effect by pressing and releasing ESC and then *key*)
- Basics:
  1. `C-x C-s`: save file in current buffer (screen) with current name (will ask for one if none is supplied)
  2. `C-x C-c`: exit (will ask for confirmation for unsaved files)
  3. `C-space`: start selecting text (selection ends at cursor position)
  4. `C-w`: cut, `M-w`: copy, `C-y`: paste
  5. `tab`: indents C/C++ code
  6. `M-x indent-region`: properly indents all selected region

3.9.2 Comments

- Not necessary for the computer
- **Absolutely necessary for the programmer / maintainer**
- One-line comments: introduced by `//`
- Multi-line comments: `/* ... */`
- Avoid over- and under-commentation
- Example of over-commentation

```
// assign 0 to x
double x = 0;
```

- Example of under-commentation

```
char buffer[] = "01011010 01100100";
char* bufPtr = buffer;
while(*bufPtr &&
   (*bufPtr++ = *bufPtr == '0' ?  'F' : 'T'));
```

3.10 Structure of a C++ program

Example.

```
/*********************************************
* Name:       helloworld.cxx
* Author:     Leo Liberti
* Source:     GNU C++
* Purpose:    hello world program
* Build:      c++ -o helloworld helloworld.cxx
* History:    060818 work started
*********************************************/

#include<iostream>

int main(int argc, char** argv) {
  using namespace std;
  cout << "Hello World" << endl;
  return 0;
}
```

- Each executable program coded in C++ must have one function called **main()**
  `int main(int argc, char** argv);`
- The `main` function is the entry point for the program
- It returns an integer *exit code* which can be read by the shell
- The integer `argc` contains the number of arguments on the command line
- The array of character arrays `**argv` contains the arguments: the command `./mycode arg1 arg2` gives rise to the following storage:
  `argv[0]` is a `char` pointer to the string `./mycode`
  `argv[1]` is a `char` pointer to the string `arg1`
  `argv[2]` is a `char` pointer to the string `arg2`
  `argc` is an `int` variable containing the value 3
- C++ programs are stored in one or more text files
- Source files: contain the C++ code, extension `.cxx`
- Header files: contain the declarations which may be common to more source files, extension `.h`
- Source files are compiled
- Header files are included from the source files using the *preprocessor directive* `#include`
  `#include<standardIncludeHeader>` `#include "userDefinedIncludeFile.h"`

3.11 The building process

All source code is held in ASCII text files. The process by which these files are transformed into an executable program consisting of machine code instructions is called `building`. A program may involve many source files written in many different languages. Some of the code may already be pre-compiled into libraries, against which the program must be linked. Handling the whole process may be a nontrivial task, and is therefore an important part of the programming knowledge. To a program we associate a *project*, which loosely speaking is a workspace containing everything that is needed to build the program.

The stages for building a project are as follows.

- Creating a directory for your project(s) `mkdir` *directoryName*
- Entering the directory `cd` *directoryName*
- Creating/editing the C++ program
- Building the source
- Debugging the program/project
- Packaging/distribution (Makefiles, READMEs, documentation...)

3.11.1 Compilation and linking

The translation process from C++ code to executable is called *building*, carried out in two stages:

1. *compilation*: production of an intermediate object file (`.o`) with unresolved external symbols
2. *linking*: resolve external symbols by reading code from standard and user-defined libraries

```
int
getReturnValue(void);
int main() {
   int ret = 0;
   ret =
getReturnValue();
   return ret;
}
```

Compilation → OBJECT CODE: dictionary associating function name to machine language, save for undefined symbols (`getReturnValue`)

```
main:  0010 1101
...getReturnValue
```

Linking → looks up libraries (`.a` and `.o`) for unresolved symbols definitions, produces executable

3.11.2 File types

- C++ *declarations* are stored in text files with extension `.h` (*header files*)
- C++ source code is stored in text files with extension `.cxx`
- Executable files have no extensions but their "executable" property is set to on (e.g. `ls -la /bin/bash` returns `'x'` in the properties field)
- Each executable must have exactly *one* symbol `main` corresponding to the first function to be executed
- An executable can be obtained by *compiling* many source code files (`.cxx`), *exactly one of which* contains the definition of the function `int main(int argc, char** argv);`, and linking all the objects together
- Source code files are compiled into object files with extension `.o` by the command `c++ -c` *sourceCode.cxx*

3.11.3 Object files

- An object file (`.o`) contains a table of symbols used in the corresponding source file (`.cxx`)
- The symbols whose definition was given in the corresponding source file are *resolved*

- The symbols whose definition is found in another source file are *unresolved*
- Unresolved symbols in an object file can be resolved by *linking* the object with another object file containing the missing definitions
- An executable cannot contain any unresolved symbol
- A group of object files `file1.o`, ..., `fileN.o` can be linked together as a single executable `file` by the command `c++ -o file file1.o ... fileN.o` only if:
  1. the symbol `main` is resolved exactly once in exactly one object file in the group
  2. for each object file in the group and for each unresolved symbol in the object file, the symbol must be resolved in exactly one other file of the group

3.11.4 Debuggers

- GNU/Linux debugger: `gdb`
- Graphical front-end: `ddd`
- Designed for Fortran/C, not C++
- Can debug C++ programs but has troubles on complex objects (use the "insert a print statement" technique when `gdb` fails)
- Memory debugger: `valgrind` (to track pointer bugs)
- In order to debug, compile with `-g` flag: `c++ -g -o helloworld helloworld.cxx`
- More details in the exercise book

3.11.5 Packaging and distribution

- For large projects with many source files, a `Makefile` (detailing how to build the source) is essential
- Documentation for a program is **absolutely necessary** for both users and maintainers
- Better insert a minimum of help within the program itself (to be displayed on screen with a particular option, like `-h`)
- A `README` file to briefly introduce the software is usual
- There exist tools to embed the documentation within the source code itself and to produce Makefiles more or less automatically
- UNIX packages are usually distributed in tarred, compressed format (extension `.tar.gz` obtained with the command `tar zcvf` *directoryName*`.tar.gz` *directoryName*

4 Classes

4.1 Basic class semantics

4.1.1 Classes: motivations

1. Problem analysis is based on data and algorithm break-down structuring ⇒ hierarchical design for data and algorithms

2. Fewer bugs if data inter-dependency is low ⇒ design data structure first, then associate algorithms to data (not the reverse)
3. Data structures are usually complex entities ⇒ need for sufficiently rich expressive powers for data design
4. Different data objects may share some properties ⇒ exploit this fact in hierarchical design

4.1.2 The class concept

A *class* is a user-defined data type. It contains some data *fields* and the *methods* (i.e. algorithms) acting on them.

```
class TimeStamp {
 public:  // can be accessed from outside
  TimeStamp(); // constructor
  ~TimeStamp(); // destructor
  long get(void) const; // some methods
  void set(long theTimeStamp);
  void update(void);
 private:  // can only be accessed from inside
  long timestamp; // a piece of data
};
```

4.1.3 Objects of a class

- An *object* is a piece of data having a class data type
- A class is declared, an object is defined
- In a program there can only be one class with a given name, but several objects of the same class
- Example:
```
TimeStamp theTimeStamp; // declare an object
theTimeStamp.update(); // call some methods
long theTime = theTimeStamp.get();
std::cout << theTime << std::endl;
```

4.1.4 Referring to the current object

- Occasionally, we may want to know the address of an object within one of its methods
- Each object is endowed with the `this` pointer `cout << this << endl;`

4.1.5 Constructors and destructors

- The class constructor defines the data fields and performs all user-defined initialization actions necessary to the object
- The class constructor is called only once when the object is defined
- The class destructor performs all user-defined actions necessary to object destruction
- The class destructor is called only once when the object is destroyed
- An object is destroyed when its *scope* ends (i.e. at the first brace } closing its level)

4.1.6 Lifetime of an object

```
int main(int argc, char** argv) {
   using namespace std;
   TimeStamp theTimeStamp; // object created here
   theTimeStamp.update();
   long theTime = theTimeStamp.get();
   if (theTime > 0) {
      cout << "seconds from 1/1/1970:  "
           << theTime << endl;
   }
   return 0;
} // object destroyed before brace (scope end)
```

Constructor and destructor code:

```
TimeStamp::TimeStamp() {
   std::cout << "TimeStamp object constructed at address "
        << this << std::endl;
}
TimeStamp::~TimeStamp() {
   std::cout << "TimeStamp object at address "
        << this << " destroyed" << std::endl;
}
```

Output:

```
TimeStamp object constructed at address 0xbffff24c
seconds from 1/1/1970:  1157281160
TimeStamp object at address 0xbffff24c destroyed
```

4.1.7 Data access privileges

```
class ClassName {
  public:
    members with no access restriction
  protected:
    access by: this, derived classes, friends
  private:
    access by: this, friends
} ;
```

- a *derived* class is a class which inherits from this (see inheritance below)
- a function can be declared *friend* of a class to be able to access its protected and private data

```
class TheClass {
   ...
   friend void theFriendMethod(void);
};
```

4.1.8 Namespaces

- All C++ symbols (variable names, function names, class names) exist within a *namespace*

- The complete symbol is `namespaceName::symbolName`
- The only pre-defined namespace is the *global namespace* (its name is the empty string `::varName`)
- Standard C++ library: namespace `std` `std::string`

```
namespace WET {
   const int maxBufSize = 1024;
   const char charCloseTag = '>';
}
```

```
char buffer[WET::maxBufSize];
```

```
using namespace WET;
for(int i = 0; i < maxBufSize - 1; i++) {
   buffer[i] = charCloseTag;
}
```

4.1.9 Exceptions

- Upon failure, a method may abort its execution
- We do not wish the whole program to abort
- Mechanism:
  1. method *throws* an *exception*
  2. caller method *catches* it
  3. called method handles it if it can
  4. otherwise it re-throws the exception
- Exceptions are passed on the method calling hierarchy levels until one of the method can handle it
- If exceptions reaches `main()`, the program is aborted

An *exception* is a class. Exceptions can be thrown and caught by methods. If a method throws an exception, it must be declared: *returnType methodName*`(`*arguments*`) throw (`*ExceptionName*`)`

- The `TimeStamp::update()` method obtains the current time through the operating system, which is outside the program's control
- `update()` does not know how to deal with a failure directly, as it can only update the time; should failure occur, control is delegated to higher-level methods

```
class TimeStampException {
 public:
   TimeStampException();
   ~TimeStampException();
}
```

```
void TimeStamp::update(void) throw (TimeStampException) {
   using namespace std;
   struct timeval tv;
   struct timezone tz;
   try {
      int retVal = gettimeofday(&tv, &tz);
      if (retVal == -1) {
         cerr << "TimeStamp::updateTimeStamp():  "
              << "could not get system time" << endl;
         throw TimeStampException();
      }
   } catch (...)  {
      cerr << "TimeStamp::updateTimeStamp():  "
           << "could not get system time" << endl;
      throw TimeStampException();
   }
   timestamp = tv.tv_sec;
}
```

4.1.10 Overloading operators in and out of classes

- Suppose you have a class `Complex` with two pieces of private data, `double real;` and `double imag;`
- You wish to overload the `+` operator so that it works on objects of type `Complex`
- There are two ways: (a) declare the operator outside the class as a friend of the `Complex` class; (b) declare the operator to be a member of the `Complex` class
- (a) declaration:

```
class Complex {
 public:
   Complex(double re, double im) :  real(re), imag(im) {}
   ...
   friend Complex operator+(Complex& a, Complex& b);
 private:
   double real;
   double imag;
}
```

definition (out of the class):

```
Complex operator+(Complex& a, Complex& b) {
   Complex ret(a.real + b.real, a.imag + b.imag);
   return ret;
}
```

- (b) declaration:

```
class Complex {
 public:
   Complex(double re, double im) :  real(re), imag(im) {}
   ...
   Complex operator+(Complex& b);
 private:
   double real;
   double imag;
}
```

definition (in the class):

```
Complex Complex::operator+(Complex& b) {
   Complex ret(this->real + b.real, this->imag + b.imag);
   return ret;
}
```

- `this->` is not strictly required, but it makes it clear that *the left operand is now the object calling the* `operator+` *method*

4.1.11 The stack and the heap

- Executable program can either refer to near memory (the *stack*) or far memory (the *heap*)
- Accessing the stack is **faster** than accessing the heap
- The stack is **smaller** than the heap
- Variables are allocated on the stack `TimeStamp tts;`
- Common bug (but hard to trace): **stack overflow** `char veryLongArray[1000000000];`
- Memory allocated on the stack is deallocated automatically at the end of the scope where it was allocated (closing brace })
- Memory on the heap can be accessed through *user-defined memory allocation*
- Memory on the heap must be deallocated explicitly, otherwise *memory leaks* occur, exhausting all the computer's memory
- Memory on the heap **must not be deallocated more than once** (causes unpredictable behaviour)

4.1.12 User-defined memory allocation

- Operator `new`: allocate memory from the heap *pointerType* pointerName* `= new` *pointerType* `;`
  `TimeStamp* ttsPtr = new TimeStamp;`
- Operator `delete`: release allocated memory `delete` *pointerName*`;` delete ttsPtr;
- Commonly used with arrays in a similar way:
  pointerType pointerName* `= new` *pointerType* `[`*size*`];`
  `double* positionVector = new double [3];`
  `delete []` *pointerName* `;` `delete [] positionVector;`
- **Improper user memory management causes the most difficult C++ bugs!!**

4.1.13 Using object pointers

- Suppose `ttsPtr` is a pointer to a `TimeStamp` object
- Two equivalent ways to call its methods:
  1. `(*ttsPtr).update();`
  2. `ttsPtr->update();`
- Prefer second way over first

4.2 Input and output

4.2.1 Streams

- Data "run" through *streams*
- Stream types: input, output, input/output, standard, file, string, user-defined

  outputStreamName `<<` *varName or literal ...* `;` `std::cout << "i = " << i << std::endl;`

  inputStreamName `>>` *varName* `;` `std::cin >> i;`

```
stringstream buffer;
char myFileName[] = "config.txt";
ifstream inputFileStream(myFileName);
char nextChar;
while(inputFileStream && !inputFileStream.eof()) {
   inputFileStream.get(nextChar);
   buffer << nextChar;
}
cout << buffer.str();
```

4.2.2 Object onto streams

- Complex objects may have a complex output procedure
- **Example**: we want to be able to say `cout << theTimeStamp << endl;` and get `Thu Sep 7 12:23:11 2006` as output
- Solution: overload the `<<` operator

```
std::ostream& operator<<(std::ostream& s,
TimeStamp& t)
   throw (TimeStampException);
```

```
#include <ctime>
std::ostream& operator<<(std::ostream& s, TimeStamp& t)
   throw (TimeStampException) {
   using namespace std;
   time_t theTime = (time_t) t.get();
   char* buffer;
   try {
      buffer = ctime(&theTime);
   } catch (...)  {
      cerr << "TimeStamp::updateTimeStamp():  "
           "couldn't print system time" << endl;
      throw TimeStampException();
   }
   buffer[strlen(buffer) - 1] = '\0';
   s << buffer;
   return s;
}
```

4.2.3 Overloading the << and >> operators

- How does an instruction like `cout << "time is " << theTimeStamp << endl;` work?
- Can parenthesize is as `(((cout << "time is ") << theTimeStamp) << endl);` to make it clearer

- Each `<<` operator is a binary operator whose left operand is an object of type `ostream` (like the `cout` object); we need to define an operator overloading for each new type that the right operand can take
- Luckily, many overloadings are already defined in the Standard Template Library
- The declaration to overload is:
  ```
  std::ostream& operator<<(std::ostream& outStream,
                           newType& newObject)
  ```
- To output objects of type `TimeStamp`, use:
  ```
  std::ostream& operator<<(std::ostream& outStream,
                           TimeStamp& theTimeStamp)
  ```
- Note: in order for the chain of `<<` operators to output all their data to the same `ostream` object, each operator must return the same object given at the beginning of the chain (in this case, `cout`)
- In other words, each overloading must end with the statement `return outStream;` (notice `outStream` is the *very same name* of the input parameter — so if the input parameter was, say, `cout`, then that's what's being returned by the overloading)

4.3 Inheritance and polymorphism

4.3.1 Inheritance

- Consider a class called `FileParser` which is equipped with methods for parsing text occurrences like `tag = value` in text files
- We now want a class `HTMLPage` representing an HTML page with all links
- `HTMLPage` will need to parse an HTML (text) file to find links; these are found by looking at occurrences like `HREF="url"`
- It is best to keep the text file parsing data/methods and HTML-specific parts independent
- `HTMLPage` can *inherit* the public data/methods from `FileParser`:
  ```
  class HTMLPage :  public FileParser {...} ;
  ```

4.3.2 Nested inheritance

- Consider a corporate personnel database
- Need `class Employee;`
- Certain employees are "empowered" (have more responsibilities): need
  `class Empowered : public Employee;`
- Among the empowered employees, some are managers: need `class Manager : public Empowered;`
- `Manager` contains public data and methods from `Empowered`, which contains public data and methods from `Employee`

```
class Employee {
 public:
   Employee();
   ~Employee();
   double
getMonthlySalary(void);
   void getEmployeeType(void);
};
```

←

```
class Empowered :  public Employee
{
 public:
   Empowered();
   ~Empowered();
   bool isOverworked(void);
   void getEmployeeType(void);
};
```

↑

```
class Manager :  public Empowered
{
 public:
   Manager();
   ~Manager();
   bool isIncompetent(void);
   void getEmployeeType(void);
};
```

4.3.3 Hiding

Consider method `getEmployeeType`: can be defined in different ways for `Manager`, `Empowered`, `Employee`: *hiding*

```
void Employee::getEmployeeType(void) {
   std::cout << "Employee" << std::endl;
}
void Empowered::getEmployeeType(void) {
   std::cout << "Empowered" << std::endl;
}
void Manager::getEmployeeType(void) {
   std::cout << "Manager" << std::endl;
}
```

4.3.4 Nested inheritance and hiding

Examples of usage

```
Employee e1;
Empowered e2;
Manager e3;
cout << e1.getMonthlySalary(); // output the monthly salary
cout << e2.getMonthlySalary(); // call to the same fn as above
e1.getEmployeeType(); // output:  Employee
e2.getEmployeeType(); // output:  Empowered (call to different
fn)
e3.getEmployeeType(); // output:  Manager (call to different fn)
e3.Employee::getEmployeeType(); // output:  Employee (forced
call)
cout << e1.isIncompetent(); // ERROR, not in base class
```

4.3.5 Inheritance vs. embedding

- Consider example of a salary object:

```
class Salary {
   Salary();
   ~Salary();
   void raise(double newSalary);
   ...
};
```

- Might think of deriving `Employee` from `Salary` so that we can say `theEmployee.raise();` to raise the employee's salary
- Technically, nothing wrong
- Architecturally, very bad decision!
- Rule of thumb: *derive B from A only if B can be considered as an A*
- In this case, better embed a `Salary` object as a data field of the `Employee` class

4.3.6 Polymorphism

- Hiding provides *compile-time polymorphism*
- Almost always, this is **not** what is desired, and should be avoided!
- Want to be able to choose the class type of an object *at run-time*
- Suppose we want to write a function such as:

```
void use(Employee* e) {
   e->getEmployeeType();
}
```

and then call it using Employee, Empowered, Manager objects:

```
use(&e1); // output:  Employee
use(&e2); // output:  Employee
use(&e3); // output:  Employee
```

- As far as `use()` is concerned, the pointers are all of `Employee` type, so wrong method is called

Run-time polymorphism can be obtained by declaring the relevant methods as `virtual`

```
class Employee {
   ...
   virtual void getEmployeeType(void);
   ...
};
```

```
class Empowered :  public Employee {
   ...
   virtual void getEmployeeType(void);
   ...
};
```

```
class Manager :  public Empowered {
   ...
   virtual void getEmployeeType(void);
   ...
};
```

```
use(&e1); // output:  Employee
use(&e2); // output:  Empowered
use(&e3); // output:  Manager
```

4.3.7 Pure virtual classes

- Get objects to interact with each other: need *conformance* to a set of mutually agreed methods
- In other words, need an *interface*
- All classes derived from the interface implement the interface methods as declared in the interface
- Can guarantee the formal behaviour of all derived objects
- In C++, an interface is known as a *pure virtual class*: a class consisting only of method declarations and no data fields
- A pure virtual class has no constructor — no object of that class can ever be created (only objects of derived classes)
- A pure virtual class may have a virtual destructor to permit correct destruction of derived objects
- All methods (except the destructor) are declared as follows: *returnType methodName(args)* `= 0;`
- All derived classes **must** implement all methods

4.3.8 Pure virtual classes

```
class EmployeeInterface {
 public:
   virtual ~EmployeeInterface() { }
   virtual void getEmployeeType(void) = 0;
};
```

```
class Employee :  public virtual EmployeeInterface {...};
class Empowered :  public Employee, public virtual EmployeeInterface {...};
class Manager :  public Empowered, public virtual EmployeeInterface {...};
```

```
void use(EmployeeInterface* e) {...}
...
use(&e1); // output:  Employee
use(&e2); // output:  Empowered
use(&e3); // output:  Manager
```

- Code behaves as before, but clearer architecture
- `public virtual` inheritance: avoids having many copies of `EmployeeInterface` in `Empowered` and `Manager`

5 Templates

5.1 User-defined templates

5.1.1 Templates

- *Situation*: action performed on different data types
- *Possible solution*: write many functions taking arguments of many possible data types.

- *Example*: swapping the values of two variables

```
void varSwap(int& a, int& b);
void varSwap(double& a, double& b);
...
```

- Potentially an unlimited number of objects ⇒ invalid approach
- Need for *templates*

```
template<class TheClassName> returnType functionName(args);
```

```
template<class T> void varSwap(T& a, T& b) {
   T tmp(b);
   b = a;
   a = tmp;
}
```

Behaviour with predefined types:

```
int ia = 1;
int ib = 2;
varSwap(ia, ib);
cout << ia << ", " << ib << endl; // output:  2, 1
```

```
double da = 1.1;
double db = 2.2;
varSwap(da, db);
cout << da << ", " << db << endl; // output:  2.2, 1.1
```

Behaviour with user-defined types:

```
class MyClass {
 public:
   MyClass(std::string t) :  myString(t) { }
   ~MyClass() { }
   std::string getString(void) { return myString; }
   void setString(std::string& t) { myString = t; }
 private:
   std::string myString;
};
```

```
MyClass ma("A");
MyClass mb("B");
varSwap(ma, mb);
cout << ma << ", " << mb << endl; // output:  B, A
```

5.1.2 Internals and warnings

- Many hidden overloaded functions are created **at compile-time** (one for each argument list that is actually used)
- Very difficult to use debugging techniques such as breakpoints (which of the hidden overloaded functions should get the breakpoints?)
- **Use sparingly**
- But use the Standard Template Library as much as possible (already well debugged and very efficient!)

5.2 Standard Template Library

5.2.1 The STL

- Collection of generic classes and algorithms
- Born at the same time as C++
- Well defined
- Very flexible
- Reasonably efficient
- Use it as much as possible, do not reinvent the wheel!
- Documentation: `http://www.sgi.com/tech/stl/`
- Contains:
  - Classes: `vector`, `map`, `string`, I/O streams, ...
  - Algorithms: `sort`, `swap`, `copy`, `count`, ...

5.2.2 `vector` example

```
#include<vector>
#include<algorithm>
...
using namespace std;
vector<int> theVector;
theVector.push_back(3);
theVector.push_back(0);
if (theVector.size() >= 2) {
   cout << theVector[1] << endl;
}
for(vector<int>::iterator vi = theVector.begin();
      vi != theVector.end(); vi++) {
   cout << *vi << endl;
}
sort(theVector.begin(), theVector.end());
for(vector<int>::iterator vi = theVector.begin();
      vi != theVector.end(); vi++) {
   cout << *vi << endl;
}
```

5.2.3 `map` example

```
#include<map>
#include<string>
...
using namespace std;
map<string, int> phoneBook;
phoneBook["Liberti"] = 3412;
phoneBook["Baptiste"] = 3800;
for(map<string,int>::iterator mi = phoneBook.begin();
      mi != phoneBook.end(); mi++) {
   cout << mi->first << ":  " << mi->second << endl;
}
cout << phoneBook["Liberti"] << endl;
cout << phoneBook["Smith"] << endl;
for(map<string,int>::iterator mi = phoneBook.begin();
      mi != phoneBook.end(); mi++) {
   cout << mi->first << ":  " << mi->second << endl;
}
```