C++ Programming

Lecture Notes for the Course CSI 2172

István T. Hernádvölgyi
1998, 1999

artwork by *Sarah Ouerd*

Lecture notes for the course **2172{A,B}** given at the **S**chool of **I**nformation **T**echnology and **E**ngineering (**SITE**) at the University of Ottawa for the Winter terms of 1998 and 1999.

C++ Programming ______________________ 1997, 1998, 1999

Author: István T. Hernádvölgyi

Format:

This document was typesetted with LaTeX.

School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada
K1N 5N5

```
@booklet{CSI2172A,
    title={C++ Programming},
    author={Istv{\'a}n T. Hern{\'a}dv{\"o}lgyi},
    year={1997,1998,1999},
    note={Lecture notes for the course CSI2172\{A,B\}},
    howpublished={University of Ottawa}
}
```

Contents

Foreword

These notes were first used in the course CSI2172A in the Winter semester of 1998. This print is the first revision, which I believe, has most of my mistakes corrected and the examples simplified. The sole purpose of these notes is to teach the core language and the fundamental elements, mechanisms and programming patterns that characterize "good" C++ programs. Neither in depth nor in breadth these notes cover every aspect of C++ . Specifically, I excluded functions from libraries (*stdio, stdlib, math ...*) and even many of those classes which make up the *Standard Template Library* (*STL*). Flavors of C++ (`Visual C++ , μC++, ...`) are also excluded. For this reason, the students should also obtain a *reference book* and a *manual* for the particular platform. The material is mostly concerned with the semantics of *creating, destroying, passing, writing* or in one word *using* objects in C++ .

Unfortunately, today (*1998*), the *1997* ANSI draft of C++ is not 100% supported by compilers uniformly across platforms. For example, some older C++ compilers do not have a `delete[]` operator, many do not support `namespace`s, logical operators return `int` as opposed to `bool` or instantiate templates in an inconsistent manner to a degree that even *STL* components do not port and special flags are required to turn on such mechanisms as *exception handling*. I tried the examples on Microsoft Windows 95, Microsoft Windows NT with Borland and Microsoft compilers and on Unix (HP-UX, Solaris) with gcc and native Sun CC compilers. Some special settings of the compiler flags and switches are needed to compile the *exception handling*, *template* and *namespace* examples. It is also assumed that the standard C and C++ libraries (*math, stdio, stdlib, iostream, string, ...*) are installed.

C++ is an unusually complex language, which supports *procedural*, *object oriented* and *parametric* styles of programming. C++ is also unique to provide static or compile time created instances which requires special mechanisms for initializing, deleting, passing and returning objects by value. While C++ is almost literally a superset of the C language, the time available in a one semester lab course is insufficient to clearly draw a line between the two languages. The first two lectures are mainly involved with the chore of demonstrating C and C++ syntax, which is followed by a brief introduction into C++ classes and objects. These lectures should be complemented with ample amount of lab exercises and examples. The rest of the lectures concentrate on demonstrating *Object Oriented Programming* and some *Object Oriented Technology* (*streamed I/O, abstraction, polymorphism, design patterns, associations, parametric programming*).

The assumed background of the student is "good understanding" of the procedural paradigm and some practice with such a language (*like* Pascal). The material should also be reinforced with 6-8 assignments or 2-3 assignments and a considerable strength project. I prefer the latter option, because it gives more of an opportunity for the students to do their own research of the libraries available, to make decisions of what frameworks or design patterns to use and to exercise *Object Oriented Design* at least to a limited extent.

István T. Hernádvölgyi
April 1998

At the end of most lectures a new section *Notes on Syntax and Semantics* has been added. Its purpose is to draw attention to some of the caveats and dangers of implicit mechanisms present in C++ and to explain some of the confusion which may be due to the overwhelming notation of C++ . Some comments and suggestions I make on design and usage present my view which may be different from what some authors propose. There are many issues in C++ and object oriented programming still debated with no hope of consensus.

István T. Hernádvölgyi
June 1999

Chapter 1

Object Oriented Programming and C++

1.1 Object Oriented Programming

C++ is a general purpose programming language which is suitable to implement a wide variety of applications. It supports *procedural*, *object oriented* and *generic* styles of programming. All programming languages support some styles of programming. The *style*s supported by a language define the *paradigm* in which the application should be designed. *Pure* languages support only one particular paradigm. For example Prolog supports the *logic*, Scheme and Lisp support *functional* programming, Pascal and C support the *procedural* or *imperative* style of programming and SmallTalk and Java support the *object oriented* paradigm. Non of these paradigms in general is superior to the other although some may be better suited for particular applications than the others. Programming paradigms are techniques of abstracting an application and they define the fundamental building blocks of problem decomposition.

The *procedural* or *imperative* paradigm abstracts the tasks performed by the application. Tasks are decomposed into subtasks. Each task is implemented as a procedure which may call other procedures. Good decomposition should identify general tasks which can be used in many contexts (*for example, maximum of a list*).

The *functional paradigm* is similar to the *procedural paradigm* in that tasks are decomposed into subtasks. However, these subtasks are implemented as functions which always return the value of the computation. Unlike procedural code, information is not saved in variables but propagated. The function is a *first class* object so it can be manipulated by other functions much the same way as if it were data. Because of this flexibility this paradigm is most popular in *Artificial Intelligence.*

In the *logic* paradigm a program is defined by *rules* and *facts*. The *rules* are formulated as *horn clause*s and define the inductive step of the algorithm. The facts, in essence, are the stopping criteria for recursion. Logic programs require a more complicated *execution engine* which uses matching and backtracking. This way applications are *specified* rather than *implemented* (*declarative programming*). This paradigm is also very popular in *Artificial Intelligence.*

The basic unit of decomposition in the *object oriented* paradigm is the *object.* The object is a software module which includes both data and functionality. In this sense an object is similar to an *abstract data structure.* However an object provides more than just combining data and operations. An object belongs to a class and the class is part of the class hierarchy. Every object is an *instance* of a *class.* The instance-class relationship is analogous to the variable-type relationship. A class defines common attributes, operations and associations but does not instantiate them. As an example *rusty* is an *instance* of *class cat* and *class cat* is a *derived class* of *class animal.*

A programming language is *object oriented*[1] if it supports:

- **encapsulation**: the internal members of the class can only be accessed through a *public* interface. This interface is the set of operations defined for the class.
- **inheritance**: classes are arranged into hierarchies. The parent of a class in the hierarchy is its *base class* and a descendant of a class is a *derived class.* A derived class *inherits* the data members and the operations of its base class. The *is a* relationship between the derived class and the base class should always hold, that is, derived class *is a* base class. Instances of the derived class are also instances of the base class. The routines that implement operations are called *methods* or *member functions* of the class.
- **polymorphism**: multiple routines can have the same name. Methods with the same name represent the same operation. It is the task of the *compiler* or the *interpreter* to resolve which implementation of the operation should be executed.

The following example defines two shapes in *object oriented pseudo code*

```
abstract class 2D-closed-shape derived from shape {
    private variable color
    abstract public operation Area
    public method setColor(C) {
      color := C
    }
    public method getColor {
      return color
    }
```

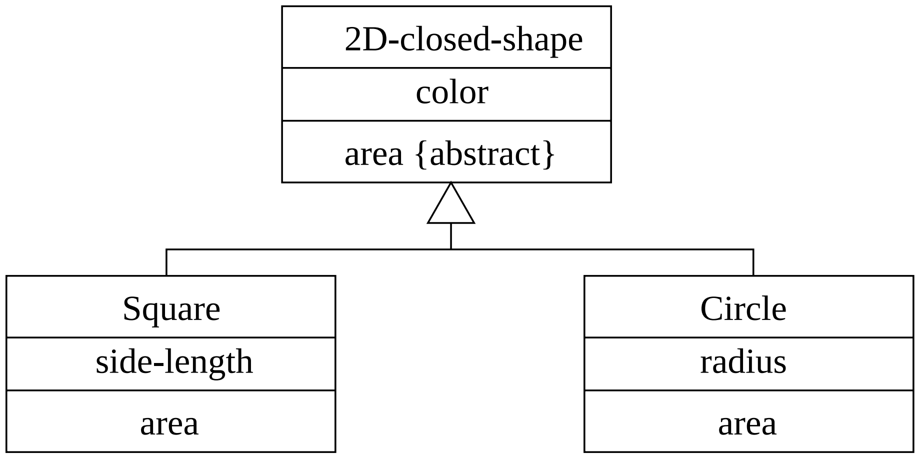
[1]some people also require an additional criterion, namely *all constructs are objects* which include *iterations, conditionals* and even *classes.*

```
}
class square derived from class 2D-closed-shape {
    private variable side-length
    public method Area {
      return side-length $\times$ side-length
    }
    public method setSideLength(L) {
      side-length := L
    }
    public method getSideLength {
      return side-length
    }
}
class circle derived from class 2D-closed-shape {
    private variable radius
    public method Area {
      return radius $\times$ radius $\times$ $\pi$
    }
    public method setRadius(R) {
      radius := R
    }
    public method getRadius {
      return radius
    }
}
```

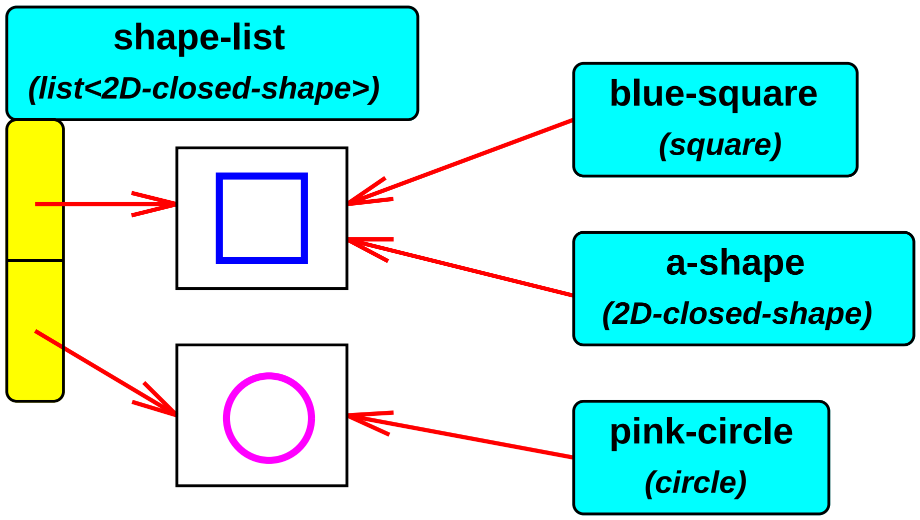
```
object shape-list of class list := new list of 2D-closed-shape
object pink-circle of class circle := new circle
object blue-square of class square := new square
object a-shape of class 2D-closed-shape := blue-square
a-shape setColor(blue)
blue-square setSideLength(5)
pink-circle setColor(pink)
pink-circle setRadius(3)
shape-list add(blue-square)
shape-list add(pink-circle)
shape-list for each value ({ getColor, getArea })
```

The expected output is:

```
blue 25
pink 28.27431
```



The run-time topology of these objects is shown on the figure below. There are only two actual instances: a pink circle and a blue triangle, however they are associated with several variables.



Line 1 declares a new instance `shape-list` of class `list`. *Line 2* declares a new instance `pink-circle` of class `circle` and initializes it with a new instance of class `circle`. *Line 3* declares a new instance `blue-square` of class `square`. *Line 4* declares an object whose name is `a-shape`, but it is not a new instance, just a different name for `blue-square`. `a-shape` is of class `2D-closed-shape`. Because instances of the derived class are instances of the base class, the assignment is valid, but `a-shape` does not understand the methods `setSideLength` and `getSideLength` because it is treated as a `2D-closed-shape` as opposed to a `square`. `blue-square` and `a-shape` are one and the same object with different names. This is called *aliasing*. *Line 5* sends the message `setColor` with argument blue to the object called `a-shape`. This also sets the color of `blue-square` because they are one and the same object. *Line*

6 sends the message `setSideLength` to the object called `blue-square`. *Line 7* sends the message `setColor` to `pink-circle`. This method is not implemented for class `circle`, but it is available through *inheritance*. *Lines 8* is analogous to *line 6*, but for `pink-circle`. *Line 9-10* add the objects `pink-circle` and `blue-square` to the list object `shape-list` *Line 11* calls methods `getColor` and `getArea` on each element of `shape-list`. Although `getArea` has different implementations for instances of class `circle` and instances of class `square`, the *compiler/interpreter* knows which version to call. We also declared the operation `getArea` in class `2D-closed-shape abstract`. This means, that instances of `2D-closed-shape` can only be instantiated by the derived classes, who implement operation `getArea`. The instance variables are declared to be `private`, so they can only be accessed through `public` methods. We also made a distinction between `operation` and `method`. An *operation* is implemented by methods, too, but it applies to the subclasses and possibly to other classes as well. Most object oriented programming languages, *like* C++ , do not make this distinction in *syntax*.

The above example demonstrates all three features of object oriented programming. If this example can be implemented this elegantly and with ease, then the programming language most certainly supports *object oriented* programming. If a programming language can implement the example, but does not support this style of programming, then the language is not *object oriented*.

1.2 C++ and Other Languages

C++ is one of many languages available for implementing applications. It is natural to ask if C++ contributes anything special or it is just syntactically different. It is true that there are many languages which can all encode arbitrarily complex algorithms. This however does not justify the use of a single universal language or paradigm. In fact, it is often more advantageous to use the paradigm, platform and language which is *best suited* for the application. As an extreme example, C++ is no contender of Prolog to implement inductive logic programs, while C++ is obviously better suited to implement applications that directly interact with the operating system. In this sense, without much religious debate, one can conclude that C++ should not be used as an alternative to

- *perl, awk, sh, csh, ksh* which amongst other related applications are mainly used to implement *system administration and management scripts*

- *Visual Basic, Tcl/Tk* which are mainly intended to implement Graphical User Interfaces (*GUI*) to applications (*such as data bases*)

- *Lisp, Prolog, Scheme, ML* which are mainly used in *Artificial Intelligence* because of their strong semantics and ability to treat "procedures" as first class objects

Pascal does not support Object Oriented Programming, but otherwise it is very similar to C . Delphi is the object oriented version of Pascal , but is only available on PC operating systems. While Pascal and Delphi are popular in Academia, they are hardly used elsewhere. The reason is that C and C++ provide direct access to the operating system. Arguably Pascal is a *cleaner* language with respect to semantics and according to many it is better suited to teach "good" procedural programming. SmallTalk is an interpreted object oriented language and is often quoted as the *de facto* object oriented language. As opposed to C++ and Java , SmallTalk does not support imperative features like *free functions* (C++) and even iterations (*loops*) are expressed as methods. SmallTalk has also had considerable success in industry, which is now reclaimed by Java . The real contender of C++ is Java . While Java is no match for C and C++ on the operating system level, it undoubtedly scores better in platform independence and library support. Java also provides platform independent windowing libraries (*AWT* and *SWING*) to implement cross platform graphical user interfaces. For this reason, Java is replacing C++ in many domains. However, Java runs on a virtual platform with no particular assumption of the underlying operating system or even its existence. While it is a definite advantage in platform independence, Java is not suitable to implement operating system level tasks. Java also uses a garbage collector and entirely hides the memory layout from the programmer. This also makes Java a poor choice for *real-time* applications. Without getting into religious discussions, it is true that while Java and C++ almost entirely overlap in the application domain, Java is usually better suited to implement *user interfaces*, "*front-end*" and "*top-layer*" portions of applications and C or C++ is needed when hard time constraints must be satisfied in real-time applications and when operating system level tasks must be performed. Fortunately, there are clean and natural methods to implement integrated C++ and Java applications through the Java Native Interface (*JNI*) and the CORBA protocol.

C and C++ compilers are also available on *parallel* computers. Parallel compilers for specific architectures (*SIMD, MISD and MIMD*) extend C++ syntax with few keywords and constructs, but otherwise do not deviate much from the original language. On these platforms, C and C++ are often the *only* choice. Finally, C is just as capable of performing system level and and real-time tasks as C++ , however it has no support for object oriented programming.

1.3 Architecture of a C++ Application

C and C++ software projects are usually developed by more than one programmers. For this reason the language supports multi file development and separate compilation of modules.

There are two kind of source files in a C or C++ software project: *header file* or *.h* files and C *source* or *.c* files. Header files contain definitions, macros, references to external variables and function prototypes. By convention, in both C and C++ , a header file's name ends with the `.h` suffix[2]. *.c* files contain the implementations of class methods, *stand alone* functions and global variable declarations. *.c* files in C should have the *.c* suffix, but C++ files may have *.cc*, *.cpp* or *.c++* suffices. The suffices are not important on PC platforms, because each vendor forces a particular convention and most likely they are all different and proprietary. Development on Unix workstations, on the other hand, relies on suffix conventions[3]. Header files should not contain implementation, only interface. The implementation of the functions and methods defined in header files is implemented in the *.c* files which can be separately compiled into *object files* and linked together to an executable.

There are several advantages of separating interface from implementation into separate files. A programmer using another developer's work should only be concerned about the interface to the package, which is separately stored in a *.h* file. The implementation of the interface can be compiled separately into an object file. In a big project, where the application is composed of dozens of object files, it is not necessary to recompile the entire project if modifications are made locally to one file. If the interface (*.h file*) changes, only those files must be recompiled which include the interface file. On the other hand, if a *.c file* is changed, only this file must be recompiled. In either case, *linking* must still be done. *Linking* is the last step in the process of creating an executable application from source code. After the software modules have been compiled into object files, the *link editor* creates the executable. Another advantage of separating coherent modules of an application into separately compiled files is that object files can be reused in other projects. Those object files that implement reusable generic components could also be archived in *libraries*. The following example is the low level architecture of a *digit recognition* application using *neural networks*.

package	**file**	**depends on**	
math	`random.cc`	`random.h`	implementation of random
	`calc.cc`	`calc.h`	implementation of calc
`math.h`	`matrix.cc`	`matrix.h`	implementation of matrix
	`ode.cc`	`ode.h calc.h`	differential calculus
neural `nnet.h`	`nnet.cc`	`nnet.h math.h`	neural networks
digit recognition	`digitrec.cc`	`nnet.h math.h digitrec.h`	implementation of digit recognition
	`main.cc`	`digitrec.h`	`main` routine

The *math* package contains routines for random number generators, differential calculus and linear algebra. The entire package definition is included in `math.h`, which wraps

[2]some vendors on PC platforms ignore this convention

[3]see the appendix on `make` files

`random.h`, `calc.h`, `ode.h` and `matrix.h` into one header file. The neural networks routine and class definitions are accessible via `nnet.h`. This package uses the `math` library. These two packages are generic and can be used in many applications. Such packages ought to be archived into a library. The last package `digitrec` is a specific application using the `nnet` package to perform digit recognition. Because it is an application, the routines should form an executable as opposed to a library. The next section discusses how to create the object files, libraries and the executable application program.

1.4 Process of Compilation

The process of compilation for a C or C++ project consists of the following steps.

- **preprocessing**: all source files are first passed to the *preprocessor*. The preprocessor expands macros, definitions and may include files or exclude parts of the source code depending on the preprocessing directives defined in the source code[4].

- **compilation**: *.c* files are compiled into object code. Object code is similar to the format of the executable, but external references to objects in *other* files and libraries are not resolved. The object files on a Unix system will have the suffix *.o*[5]

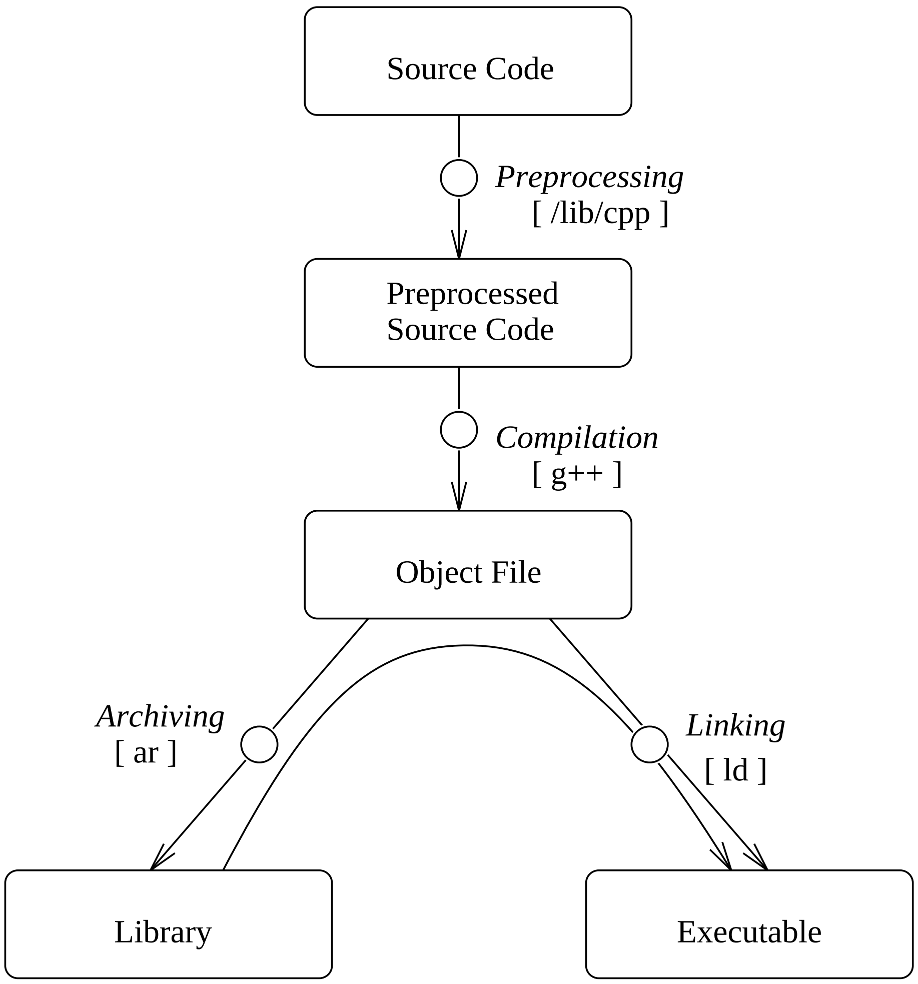
- **archiving**: Object or *.o* files can be bundled to form a library. For generic routines it is advantageous to create libraries, because routines may be linked on demand and the link editor can *pull* the routines the executable needs from one location.

- **linking**: Object files and routines from libraries are *linked* to form an executable application.

The process is shown on the figure below:

[4]for details, see the appendix on `cpp`

[5]on a PC platform they usually suffixed by *.obj*



On a Unix system, the GNU preprocessor, link editor, archiving program and compiler are freely available. On PC platforms, a C++ compiler would probably be part of an *integrated development environment* or IDE which bundles the above programs and a text editor into one single package. We use the Unix programs to demonstrate the creation of the project, because they are more explicit. To *build* the digit recognition project we would need the following steps and programs.

- `g++` : is the GNU C++ compiler. `g++` is an alias with the proper flags and options to the GNU C compiler `gcc` . If the `-c` flag is specified `g++` generates an object file from the source file. For example:
  ```
  g++ -c blah.cc
  ```
  generates the object file `blah.o`. `g++` automatically calls the preprocessor `cpp` before it generates the object code.

- `cpp` : is the preprocessor, and is automatically called by the compiler. The program usually resides in `/lib/cpp`.

- ld : is the link editor and it combines libraries and object files into executable programs. For example:
  `ld -o app main.o -lmath`
  creates the application program `app` by liking the object file `main.o` and the routines it needs from the `math` library. The file name after the `-o` flag is the name of the application, and the symbolic name `math` after `-l` instructs the link editor that it needs to link routines from the math library. By naming conventions, it is looking for the library `libmath.a`

- ar : is used to create libraries from object files. Creating the library `math` from object files `calc.o` and `matrix.o`:
  ```
  ar rc libmath.a calc.o matrix.o
  ranlib libmath.a
  ```
  rc instructs ar to create a new archive. The library is called `libmath.a`, but the link editor automatically recognizes it is as the *math* library, because it is preceded by `lib` and has suffix `.a`. The ranlib program creates a symbol table for the archive which helps the link editor to find the routines.

Finally the following steps are needed to build the digit recognition project (*$ represents the prompt*):

Compile each source file of the math library:

```
$ g++ -c random.c
$ g++ -c calc.c
$ g++ -c matrix.c
$ g++ -c ode.c
```

Create the math library:

```
$ ar rc libmath.a random.o calc.o matrix.o ode.o
$ ranlib libmath.a
```

Compile each source file of the neural networks project:

```
$ g++ -c nnet.c
```

Create the neural networks library:

```
$ ar rc libnnet.a nnet.o
$ ranlib libnnet.a
```

Compile each source file of the digit recognition application:

```
$ g++ -c digitrec.c
$ g++ -c main.c
```

Create the executable file:

```
$ ld -o digitrec digitrec.o main.o -lnnet -lmath
```

Chapter 2

Basics I.

2.1 The “hello world” Program

First let us take a look at the (in)famous “hello world” program. This program prints the message “hello world” on the screen and then it terminates.

```
#include <iostream.h>
// include the input output library

int main(void) {
   // cout is the OBJECT associated with the
   // standard output stream
   cout << "hello world" << endl;

   return 0;
   /*
     a zero return value is interpreted by the
     shell which started this program that
     it has terminated with no errors
   */
  }
```

The first line instructs the compiler to include the header file *iostream.h.* This header file contains the interface for the *iostream* library. An interface bundled in a header file consists of *constants, type declarations, class definitions, function prototypes, external references to global variables and macros.* If we did not include this library, the program would not have access to the output stream.

Every C and C++ program starts executing at the *main* function, hence a program can have only one *main*. *main* returns an integer and takes no arguments. A *void* argument or return type indicates that the function does not take arguments or does not have a return value. There are two forms of *main*: one that takes arguments from the command line, and one that ignores them. Later we discuss passing parameters at the command line. For now we use the form that disregards the command line arguments (*void*). The curly braces { and } enclose a *block*. In this case, the block is the body of *main*.

cout is a global object of class *ostream*, declared in *iostream.h*. *cout* represents the default output stream, which in most cases is the *monitor screen*. The arguments can be written to the stream using the "$\ll$" insertion operator. *endl* is a constant which stands for the end of line character. Arguments can be chained using the insertion operator; hence `cout` $\ll$ `"hello world"` $\ll$ `endl;` is equivalent to `cout` $\ll$ `"hello"` $\ll$ `" wo"` $\ll$ `"rld"` $\ll$ `endl;`

The *return* statement is an unconditional transfer of control. When this statement is reached the function terminates and the specified value is returned.

There are two ways to put comments into a C++ program. Everything starting at // until the end of that line is ignored by the compiler. The alternative is to enclose comments between /* and */. The latter may span multiple lines.

2.2 Literals

Literals are anonymous constants which represent themselves. Every programming language has some literal representation for integers, floating point numbers, characters and character strings. These literals become part of the program text (*or executable image of the program*) when the program is compiled.

Integer Literals

In C++ integer literals can be specified in *decimal*, *octal* and *hexadecimal*. A decimal representation is just a sequence of digits with an optional minus or plus sign at the front. Octals can only use the digits *0-7* and an integer literal which starts with a 0 (*zero*) and is followed by an octal digit is assumed to be an octal integer. Hexadecimal literals are prefixed by *0x* and made up of decimal digits (*0-9*) and the letters *a*, *b*, *c*, *d*, *e* and *f* – *a* being *10* and *f 15*. Integer literals can also be suffixed by the letters *U* or *L* to represent that the literal is to be stored as a `short` or `long` integer.

octal	*hexadecimal*	*decimal*
026	0x16	22
0177	0x7f	127
0377	0xff	255

Literals specified in octal or hexadecimal form usually represent bit patterns. One must be careful with a number like *0xffffffff*, as it represents *-1* for a compiler which uses 32 bits to store integers.

Character Literals

Character literals are enclosed in *single quotes* and can be assigned to variables of type `char`. A character is always stored in one byte, and the `char` data type also represents the *byte* in C and C++ . Of the 256 possible 8 bit patterns most are not printable. The ASCII standard uses only the 7 low order bits.

ASCII Characters							
0	NUL	1	SOH	2	STX	3	ETX
4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT
12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3
20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC
28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	”	35	#
36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;
60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K
76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[
92	\	93	]	94	ˆ	95	_
96	‘	97	a	98	b	99	c
100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{
124	|	125	}	126	˜	127	DEL

The quote characters `'` and `"` have to be *escaped* by \. Some non-printable characters also have an escaped literal form:

newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert (bell)	\a
backslash	\\
single quote	\'
double quote	\"
null byte	\0

Arbitrary 8-bit characters can also be represented by specifying them in *octal* or *hexadecimal*. For example, '\x30' = '\60' = '0' ($30_{16} = 60_8 = 48_{10}$).

String Literals

String literals represent character sequences and are enclosed in double quotes. The compiler automatically trails quoted string literals with the *null byte* '\0', because character strings in C and C++ are *null* terminated. Characters which otherwise have to be escaped must be prefixed by a \ in the string literal as well. For example

```
"\'hello\tworld\'\n"
"\"hi mom!\""
```

represent

```
'hello       world'

"hi mom!"
```

Floating-Point Literals

Floating-point literals are of type `double` (*or double precision floating point number*). They consist of an optional minus or plus sign followed by 0 or more digits an optional decimal point 0 or more decimal digits and an optional exponent. The exponent is represented by the character *e* followed by an optional minus or plus sign followed by digits. For example, the following are floating-point literals:

```
 3.14   .004    1.   -5.4e10    -1.e-23
```

2.3 Scope

There is no meaningful program without variables and functions. In the object oriented paradigm functions belong to objects or classes. C++ supports both; *free functions* which do not belong to any particular class, while instance or class methods belong to objects or classes. Variables can belong to free functions and instance methods or they can be instance variables. Variables may be local to the function or method, to the instance or to a class, they can belong to one particular file or be shared with other files. Because C++ inherited the imperative features of C while it supports object oriented concepts, it has unusually versatile and complex scoping rules. Every identifier belongs to a scope and this scope defines where that identifier can be used. In other words, the scope of an identifier is the logical block where the identifier is visible.

An identifier is the name of a variable, function, class, method, etc ... assigned by the programmer and it is introduced into its scope by a declaration. The following line declares an integer variable:

```
int a;
```

The declaration of a variable, in C and C++ , has the following syntax:

`type` *identifier*

It is also possible to initialize the variable when it is declared, with the following syntax:

`type` *identifier* `=` *expression*

expression must evaluate to the same type as the *identifier*. In the above example, *int* is a *primitive type* and stands for fixed storage size integer. The storage requirements for an *int* most often coincide with the computer's word size (*or 32 bits for most personal computers*). "a" is the identifier which refers to the integer. In C and C++ , identifier names can be composed from underscores and alphanumeric characters, with the restriction that a digit cannot start an identifier. Examples of syntactically correct identifiers:

hello, hello2, _blah, array_counter

C and C++ identifiers are *case sensitive*, which means that `b` and `B` are different identifiers. A variable is uniquely identified by its name, type and scope. A function is uniquely identified by its name, return type and parameter types[1].

```
int foo(int,int);
```

The line above is the prototype of a function. The function is called "foo" and it takes two integer arguments and returns an integer.

[1]methods of a class also need the classname to be uniquely identified

The following example demonstrates the scoping rules applied for identifiers declared in a file.

```
int a;
int foo(int,int);
```

file: test.cc

EXAMPLE 1

```
int a;
int foo(int,int);


int foo(int,int)
 {
   int a;
  a = 1;
  :: a = 2;
 }
```

file: test.cc

EXAMPLE 2

```
int a;
static int b;
```

file: test1.cc

```
extern int a;
extern int b; // no external b!
int foo(int,int);

int foo(int,int)
 {
   int a;
  a = 1;
  :: a = 2;
 }
```

file: test2.cc

EXAMPLE 3

In *Example 1*, both “a” and “foo” are visible in all function bodies in *test.cc*. In *Example 2*, the first “a” is visible everywhere in *test.cc*, however “foo” declares its own local “a”. As scopes can be nested, the most local scope overrides the outer scopes in case of an identifier’s name clash. The *assignment statement*, `a = 1;` sets the value of the *local “a”*. C++ supplies the scope resolution operator “::”, so the programmer can explicitly specify which scope’s variable is dereferenced. “::a” hence refers to the *global* "a". In *Example 3*, there are two files. *test1.cc* defines the integer variable "a". *test2.cc* declares an integer variable "a", which is defined in another file (*external*). The compiler is unable to check whether "a" is really defined in an external file, so it leaves resolution to the link editor. In this example, "a" is shared between *test1.cc* and *test2.cc*. Variables can be shared amongst more files which have an appropriate *extern* declaration, but each variable can only be defined only once (*as in, test1.cc, without the extern keyword*). It is also possible to prevent a variable declared outside of a function's or class' scope being shared with another file.

The *static* keyword for a free[2] declaration protects the variable from being shared with external files (*example:* `static int b;`). Such an identifier is local to the file, so *extern* declarations in other files cannot see them. It also works for static functions, which are like regular functions but are only visible in the file where they are declared. Of course one should not provide a prototype for such a function in a header file because the linker would not find it.

In fact, every pair of { and } defines a new scope in C++ .

```
 .....

  { // scope 1
   int i=2;

       { // scope 2
        int i = 3;

        cout << i <<  endl; // i = 3
       } // scope 2

   cout << i << endl;  // i = 2
  } // scope 1

  cout << i << endl; // i is not defined!

  ....
```

The `static` keyword has many meanings in C++ . Another one is the concept of a static local variable. A variable declared static in a function remembers its previous value. What really happens is that a static local variable is not on the *stack*[3] but is allocated in the program text.

[2]not bound to a class or explicit namespace

[3]we will discuss stack layout and variable allocation in more detail later

```
void foo() {
   int a = 0;
   static int b = 0;

   a = a+1;
   b = b+1;

   cout <<  a << ' ' << b << endl;
}

int main() {
   for(int i=0; i<3; i++) {
      foo();
   }
   return 0;
}
```

would print

```
1 1
1 2
1 3
```

In other words, `b` is a global variable but is only visible inside `foo` and hence its life time is the entire duration of the program and its scope is `foo`.

2.4 Namespaces

*Namespace*s provide a mechanism to logically group declarations. In other words, *classes*, *functions*, *variables* and *types* which logically belong together can be put into a single namespace which expresses this coherence. Namespaces nicely facilitate the *total* separation of *interface* from *implementation* together with *information* hiding. A library is ought to be defined in terms of an interface to what it provides. This interface is composed of the class definitions, function prototypes, constants, global resources and type definitions that are *necessary* to effectively use the library. Implementation details which may also include class definitions, method and function implementations, global resources and type definitions should be hidden from the user of the library! Unfortunately, *as of today*, namespaces are not uniformly supported across all C++ compilers, and most C++ development available today is *not* using the namespace mechanism.

Namespaces are declared by the `namespace` keyword. The body of the declaration is enclosed in curly braces. The following is the interface definition of the `university` library.

```
// File: university.h
// interface definition for the university library

namespace university {
   class Student {  // class definition
      /* ... */
   };

   class Course {  // class definition
      /* ... */
   };

   extern Database* db; // global resource

   extern const double minimum_salary; // constant

   void register(Student&,Course&,Database*); // function

};
```

Namespaces are *open*. This means that a namespace can be extended by the user to add additional classes, functions, *etc.* to both the interface and its implementation. The implementation of the interface should reside in a different file. (*or files*).

```
// File: university.cc
// implementation

#include "university.h"

// extend namespace with implementation specific
// classes, functions, ...

// extension or 'hidden' interface
// the user of the library does not know about this
// these classes help to implement the university library
namespace university {
   class Registry {
     /* ... */
   };
```

```
    const max_string_length = 1024;

    void parse_student_file(istream&,Student**,int);
};

// and now the implementation:

const double university::minimum_salary = 17000;

university::Student::Student() { // default constructor
    /* ... */
}

...

void university::register(Student& s, Course& c, Database* b) {
    /* ... */
}
```

The scope resolution operator `::` is needed to qualify the identifiers. For example, class `Student` may also be a part of a different namespace and library with a different purpose and meaning. The qualified identifier `university::minimum_salary` refers to the `minimum_salary` of `university` namespace as opposed to (*let's say*) the `company` namespace. Namespaces also define a scope where the identifiers can be referenced. Application programs can only use the public interface.

```
// File: univapp.cc
// university application program

#include "university.h" // public interface

int main(void) {
  university::Student students[1000]; // array of 1000 university students
  /* ... */

  return 0;
}
```

It is often cumbersome to use the scope resolution operator to refer to identifiers. The `using` directive causes the compiler to search in the specified namespaces for the identifier if it is not explicitly qualified.

```
  // File: univapp.cc
  #include "university.h" // public interface

  using namespace university;

  int main(void) {
    Student students[1000]; // array of 1000 university students
                            // Student is from 'university'
    /* ... */

    return 0;
  }
```

Namespaces can also be composed from other namespaces by the *using* directive.

```
// ode also includes namespaces polynomial and complex, and the
// two symbol manipulation libraries. the methods sym_differentiate
// overlap, so use Attila's instead of Istvan's

namespace ode {
  // implementation of numeric integration
  using namespace polynomial;
  using namespace complex;

  using istvan_symbolmanip;
  using attila_symbolmanip;
  using sym_polynomial attila::sym_differentiate(const sym_polynomial&);

  complex integrate(const polynomial&);
  polynomial differentiate(const polynomial&);
}
```

Namespaces provide hooks for code improvement and reusability by maintaining a standard interface and replacing it with improved implementations using *aliasing* of namespaces.

```
// OLD ODE
namespace ode1 {
  // simple implementation of numeric integration
  // using trapezoid method
  using namespace polynomial;
  using namespace complex;
  complex integrate(const polynomial&);
  polynomial differentiate(const polynomial&);
}

// NEW ODE
namespace ode2 {
  // sophisticated implementation of numeric integration
  // using variable step size adjustable order Runge-Kutta methods
  // with hundreds of lines of optimized code
  using namespace polynomial;
  using namespace complex;

  complex integrate(const polynomial&);
  polynomial differentiate(const polynomial&);
}

namespace ode = ode2; // aliasing a namespace
                     //  used to be ode = ode1

...
   complex c = ode::integrate(p); // magically works better
...
```

If the compiler supports namespaces, they should be used to develop libraries and applications, as they are included in the ANSI definition of the C++ language since 1997.

2.5 Primitive Data Types

Every programming language declares a set of *primitive types*. Variables of these types can be readily used, and the compiler knows how to interpret them. C and C++ provides the following primitive data types:

	type
int	signed integer
float	signed floating point (*real*) number
double	signed double precision real number
char	character type
bool	boolean (*only in ANSI* C++)

With additional keywords more primitive types can be formed. *int* and *char* can be preceded with the keyword `signed` and `unsigned`. The `signed` keyword explicitly declares the types signed, while the unsigned modifier forces the sign bit to be interpreted as part of the number. A *char* type variable can also be declared *unsigned*, even though it is not a numeric type. Every C and C++ compiler internally implements *char* to be 8-bits long. Hence *char* can be used as the *byte* data type. Integers (*int*) can also be declared *long* or *short*. Although compilers may decide on any fixed storage size for integers, the storage size of a *short int* is less or equal to the storage required by an *int*, which in turn requires less or the same amount of storage as a *long int*. A double precision floating point type can also be preceded by the `long` modifier. All in all, the following combinations are possible to declare a primitive type variable: *char, unsigned char, signed char, int, short int, long int, signed int, signed short int, signed long int, unsigned int, unsigned short int, unsigned long int, float, double, long double.*

While the above keywords change the storage size required for the type, or its range, the keywords below instruct the compiler, how to manage or optimize the variables. Variables can be declared *auto*, *register* and *volatile*. *auto* variables are automatically created and destroyed. This is the default, hence the `auto` keyword is redundant. The `register` keyword instructs the compiler to keep the variable in a register. The compiler may ignore this keyword. Optimization at this level is beyond the scope of this course and modern compilers are good in spotting variables that can be kept in registers automatically. A single variable used as a counter (*as in a for loop*) is usually a good candidate to be declared *register*.

Sometimes it is absolutely necessary to protect a variable from being optimized (*being kept in a cache or register*), and these cases are almost always related to processes accessing a shared resource *asynchronously*. Suppose variable `c` is a stopping condition for a loop and is shared with another process (*or thread*). The compiler may spot this variable to be optimized and puts it into a register to spare lookups by address, while another process modifies the value of `c`. The copy kept in the register is inconsistent with the variable `c` stored in memory, and hence the loop will not terminate. To guarantee that a variable is always obtained by an address lookup, it must explicitly be declared *volatile*.

2.6 Conditional and Iterative Constructs

A programming language must support conditional and iterative constructs. At the assembly language level, a conditional is asking whether the contents of a register or flag is zero, while iteration is achieved by “jump instructions”. Although, C and C++ directly support these primitive forms of control flow manipulation, you may not use “labels” and “goto” statements in this course. They can be simulated by safer constructs. Most languages provide three looping constructs: *while, repeat* and *for* loops, while there are languages that prefer recursion over explicit loops (*Prolog, Scheme*). C and C++ support all of the above.

The *for* loop is usually used in a context where the number of iterations is known ahead of time. In C and C++ , the format of the for loop is:

`for(`*initialization expression*`;`*test*`;`*increment expression*`) body`

The *initialization expression* usually initializes the counter variable used in the for loop. The *test* must be an expression which is (*or can implicitly be typecasted*[4]) to an integer. In C and C++ , 0 represents false, and a non-zero value is interpreted as true[5]. The *increment expression* usually increments (*or otherwise modifies*) the value of the counter variable. The word "usually" is used, because both C and C++ allow the programmer to use these expressions anyway he likes, including doing something out of context. The following is an example of a for loop which calculates the value of factorial:

```
int fact=1;
for (int i=2;i<=n;i=i+1)
  fact = fact*i;
```

If we had more than one statement in the body of the *for* loop, they should have been enclosed in { and }. C insists that local variables are declared before the first statement in a function, so the integer `i` must be declared before the *for* loop. C++ allows the variables to be declared as they are needed. In this case the counter variable i is local to the *for* loop. The programmer may leave any or all of the *for* loop expressions blank. For example, `for(;;)` is an *infinite loop.*

while loops have the following form:

`while(`*test*`) body`

As with *for* loops, test may be anything that resolves to an integer. If that integer is zero, *test* evaluates to false, and if it is not zero, then *test* evaluates to true. If *test* is true, the body of the loop is entered. Hence `while(1)` is an infinite loop (*so is* `while(-3.14)`). Our factorial example, using a while loop:

```
int fact=1;
int i=2;
while(i<=n) {
   fact = fact*i;
   i = i + 1; // or i++
  }
```

[4]we address implicit type casting later

[5]more recent compilers (*as of circa 1995*) support the native boolean type `bool` as well

repeat or *do-while* loops are seldom used, because the *while* and *for* constructs can simulate anything the *repeat loop* can do. In C and C++ , repeat loops have the form:

```
do
  body
while(test);
```

If *test* evaluates to true, then the body of the loop is reentered. Unfortunately, they implement the reverse of the Pascal *repeat-until* logic which exits on true. Another danger of this construct is that the loop's body is executed at least once. The factorial example, using the *do-while* loop:

```
int fact = 1;
int i = 1;
do {
  fact =  fact * i;
  i = i + 1;
} while (i<=n);
```

The *if* statement has the following three forms:

```
if (test) body
```

```
if (test1)
  body1
else
  body2
```

test ? *expression1* : *expression2*

The latter form is provided for convenience only. The value of the expression is *expression1* if *test* is true, and it is *expression2* otherwise. For example, in the statement `int max = a>b ?  a :  b;`, *max* is set to the maximum of "a" and "b".

Both C and C++ support recursion, that is, a function can be called within itself. A recursive implementation of factorial is:

```
int fact(int n) {
   if (n==0) return 1;
   else return n*fact(n-1);
  }
```

2.7 Unconditional Transfer of Control

C++ supports mechanisms that transfer control unconditionally. The use of these constructs is often not justified and must be used with care. One of these is the infamous *goto*, which we avoid altogether because it is always possible to find a safer[6] and more elegant solution. The other constructs are practical and have stronger semantics.

The **return** statement has two forms:

```
return;
return expression;
```

return forces unconditional exit from the currently executing function with proper stack clean-up. The first form can be used for functions with *void* return type. The second form can be called in functions which return a variable of the type of the expression after the `return` keyword. After the stack is properly cleaned up, the value which *expression* evaluates to is left on the stack (*to be picked up by an assignment statement or the function which initiated the call*).

break can be used in *while*, *do-while* and *for* loops, as well as in the *switch statement*. When a `break;` appears in a loop, control is transferred to the first statement *after* the loop (*control returns from the loop, or the loop is left unconditionally*). In the case of nested loops, `break;` only aborts the inner most loop which it was called from.

The **switch** statement is the equivalent of Pascal 's case statement and it relies on the appropriate insertions of `break` statements. It has the following form:

```
switch(expression)
  {
  case constant_1 :
    statements_1
  ...
  case constant_n :
    statements_n
  default:
   statements
  }
```

The **switch** statement has somewhat unusual semantics. $constant_{1 \ldots n}$ must be integral *literal constants*. For example, 1 and 32 are literal constants of *int*, and 'a' and 'n' are literal constants of *char*. In the latter case, the characters are implicitly type casted to `ints`. Control descends on the `case` clauses top down until one of the constants matches the value of the *expression*. If such a constant is found, then *all* statements following that line will be executed, including those which seemingly belong to other clauses. The *default* clause is optional; statements beneath it are only executed if no match was found above. This is why, it is almost always necessary to use *break* statements in a *switch* statement. When the `break` statement is encountered, control

[6] *goto* is unsafe because it may leave a scope without *clean up*, that is, it may not release resources, and deallocate variables

leaves the *switch* construct.

```
char c;
switch(c) {
   case 'a':
   case 'b':
           cout << "a and b" << endl;
   case 'c':
           cout << "c" << endl;
           break;
   case 'd':
   case 'e':
           cout << "d and e" << endl;
           break;
   case 'f':
           cout << "f" << endl;
   default:
           cout << "beyond f" << endl;
  }
```

The output of the program above is:

value of c	**output**
a	a and b c
b	a and b c
c	c
d	d and e
e	d and e
f	f beyond f
g,...	beyond f

continue is used to transfer control to the *start* of a *while*, *do-while* or *for* loop (*forces to start the next iteration*). If a `continue;` is issued in a for loop, everything that comes right after it, is ignored and control passes to the increment step.

exit terminates the entire process or thread of execution, which it was called from. Before *exit* terminates the program, it calls routines provided by the programmer or generated by default, to deallocate them. It also waits on pending resources. The prototype of exit is `void exit(int);`. The integer argument is returned to the "system" it was called from. By convention a zero return value indicates successful completion.

2.8 Basic I/O

Every programming language provides mechanisms to read and write data. C and C++ very much differ in this respect. For now, we only consider reading input from the keyboard and writing data to the screen C++ style.

First, the `iostream.h` header file must be included. This contains the definitions of the input and output streams, and references to the global objects `cin`, `cout` and `cerr`. `cin` represents the standard input stream or the *keyboard*, `cout` represents the standard output stream or the *monitor screen* and `cerr` represents the standard error stream, which in most cases is also the screen. `cout` and `cerr` are instances of class `ostream` (*output stream*). Each instance of `ostream` understands the binary operator $\ll$. The left hand side of the operator must be an `ostream` object and the right hand side could be a primitive type variable, a literal constant or an object for which this operator is *overloaded*[7]. For example:

```
double pi = 3.14259;
cout << "hello " << (2+5) <<  " pi = " <<  pi <<  endl;
```

prints `hello 7 pi = 3.14259`. `endl` represents the end of line character. Writing to the *standard error stream* is analogous to the example above:

```
cerr << "hello " << (2+5) <<  " pi = " <<  pi <<  endl;
```

`cin` represents the keyboard, and this global object is an instance of class `istream` (*input stream*). Every instance of `istream` understands the $\gg$ binary operator. An instance of class `istream` must be on the left hand side of this operator, while the right hand side is a variable. For example, the following line can be used to read in two integers and a real number:

```
int a,b;
double c;
cin >>  a >> b >> c;
cout << " a = " << a << " b = " << b << " c = " << c << endl;
```

To read a string, we have to declare an array of characters[8].

```
char name[100];
cin >> name;
cout << " name = " << name;
```

[7]operator overloading is discussed later

[8]arrays are covered in the next section

2.9 Notes on Syntax and Semantics

In C and C++ variable declarations the type name always precedes the name of the identifier (*which is the opposite in* Pascal). More than one variables can be declared with one type specifier on a comma separated list. Tokens (*syntax elements and identifiers*) can be separated by an arbitrary number of white spaces (*horizontal tabs, spaces and new lines*). For example

```
int i, j,
    k;

double a,b;
double c;

double


       d;
```

looks awkward but is syntactically correct. The first two lines declare integer variables `i`, `j` and `k`. The second two non-empty lines declare variables `a`, `b` and `c`. And the last declaration of variable `d` is also valid.
Loops in C++ are similar to their counterparts in other languages. The `for` loop may have its initialization, test or increment section left blank. For example

```
for(;;);
```

is an infinite loop. Because of ANSI scoping rules

```
for(int i=0;i<10;i++) {
   ...
}
```

is not equivalent to

```
int i=0;
for(;i<10;i++) {
   ...
}
```

in C++ (*although they are in* C). In C++ , loops have their own scope, so the former version's `i` is only visible inside the loop and it may shadow any `i` in the surrounding context. It is also possible to declare more than one loop variables in a comma separated list:

```
for(int i=0, double j=10; i < 10 && j > 1; i++, j= j/2 ) {
   ...
}
```

In C and C++ a comma separated list of statements evaluates to the value of the last expression but the previous ones are also evaluated.

```
int a,b = 2;

a = b++, 5 + b;
```

`b` is incremented and `a` is set to the new `b` + 5.
Input and output are quite different in the C language. Instead of I/O streams a family of `printf` and `scanf` functions are used. In general, it is not a good idea to mix the two. The differences will be discussed in detail later.
Function calls (*unlike* Pascal) always need the parentheses even if the function takes no arguments.

```
int f() {
   return 5;
}


int main() {
   cout << f() << endl; // as opposed to cout << f << endl;
   return 0;
}
```

`continue, break` and `return` are not functions so they do not need the parentheses.

2.10 Exercises

2.1 When we used the *for*, *repeat* and *while* loops, we had the variable *fact* containing the value of *n!* at the end of the computation. Implement functions *fact1, fact2* and *fact3*, which implement factorial as a function, using a *for*, *while* and *do-while* loops respectively.

2.2 Implement Euclide's famous *greatest common divisor* algorithm. Given two integers "a" and "b", *gcd(a,b)* is defined (*recursively*) as: *gcd(0,0) = 1*, *gcd(a,0) = a*, *gcd(a,b) = gcd(b,a mod b)*. Implement the function *gcd* using all of the iterative constructs we discussed, including recursion. Comment on which construct is the most appropriate. *Hint: The mod operator in* C *and* C++ *is %, that is, a % b stands for a mod b*

2.3 Implement algorithm *lcm* which calculates the *least common multiple* of two integers. *Hint: use* `gcd` *from the previous exercise*

2.4 Using all the loop constructs, write as many different *kind* infinite loops as you can.

2.5 Write a program that reads in integers in a for loop and prints them only if the value is greater or equal to zero. Use *continue.*

2.6 Write a program which reads in two *doubles* and a character in a loop. If the character '+' is read then the two doubles are added up and the result is printed. Similarly, if '-', '/' or '*' is read, the corresponding arithmetic operation is executed. If the character '#' is read, then the program terminates. If a character other than the ones already mentioned is read, the program prints an error message to the standard output stream and continues.

Chapter 3

Basics II.

3.1 Enumerated Types

Enumerations can hold a set of *constant values* specified by the programmer. For example, the following declaration defines `day` to hold any one of the 7 days and refer them to by their names:

```
enum day { Monday=1,Tuesday,
       Wednesday,Thursday,Friday,Saturday,Sunday };
```

If we did not set *Monday* to 1, the enumeration would start at zero. We can also set each of the constants in the enumeration for a specific integral value:

```
#include <iostream.h>

enum foo { a=-21,b=-2,c=1,d=23};

int main(void) {
   foo f1 = a;
   foo f2 = foo(2);
   foo f3 = foo(-22);
   cout << "f1 = " << f1 << " f2 = "
        << f2 << " f3 = " << f3 << endl;

   return 0;
  }
```

The output of the program is:

```
f1 = -21 f2 = 2 f3 = -22
```

The range values of which *foo* can be instantiated is actually bigger than the values we set the elements to. The form *foo(-23)* explicitly requests the variable to hold the value

-23, even though, we cannot access it by a symbolic constant. This usage is rare and if you set or dereference the value of an enumerated type variable, you would always use the symbolic constants. A better example is:

```
switch(d) {
  case Monday   :
  case Tuesday  :
  case Wednesday:
  case Thursday :
  case Friday   : work();
                  break;
  case Saturday :
  case Sunday   : rest();
                  break;
 }
```

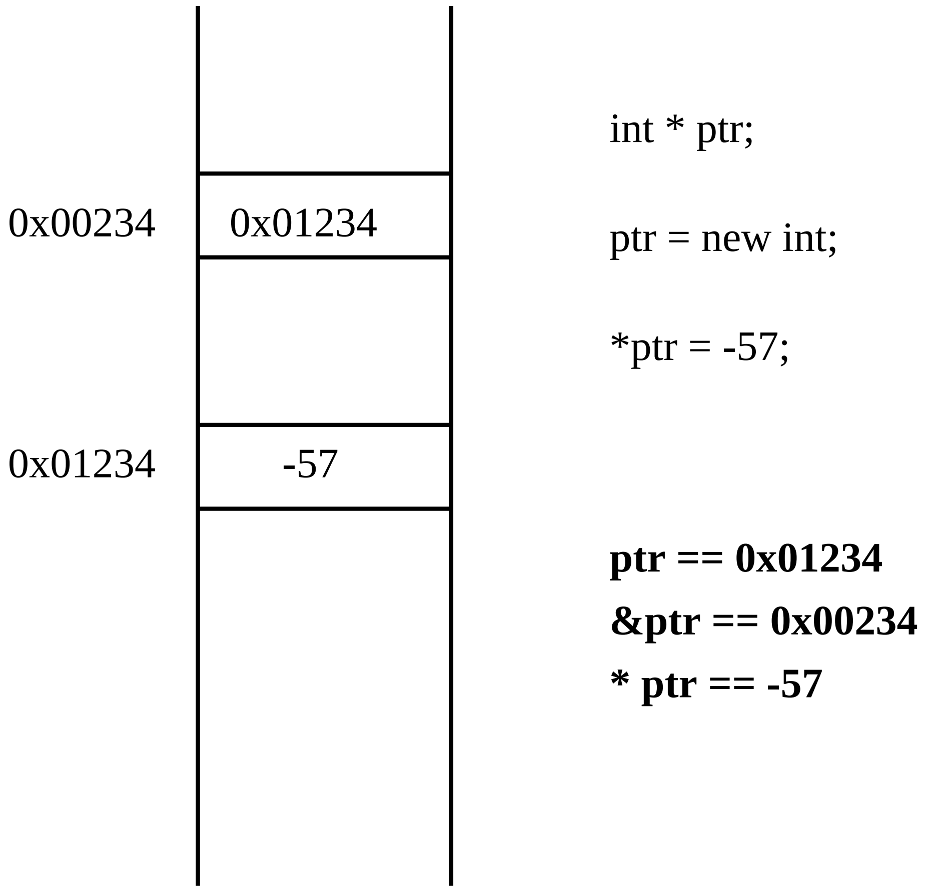
3.2 Pointers

A *pointer* is a *variable* which holds the address of another variable[1]. This allows two *pointer variables* to indirectly modify the value of the variable they are both pointing to. Pointers are also one of the major sources of errors in a program; if the address stored in the variable is invalid, the contents of that part of memory can be corrupted. If the address is outside of the *address space* allocated by the operating system for the program, it is likely that the program will terminate with an error, or if it does not, it can cause even more damage. Syntactically a pointer variable is declared like a variable it is pointing to, preceded by an *.

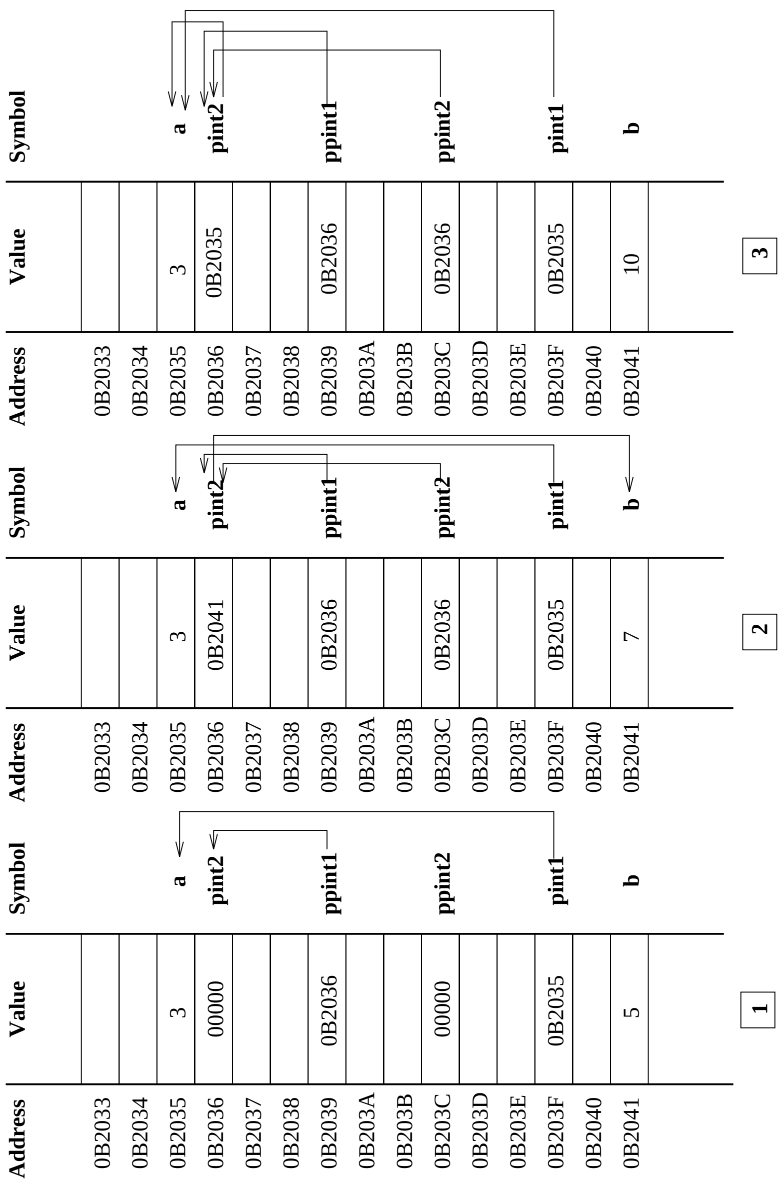
```
int a;
int *pa;
```

`a` is an integer variable and `pa` is a pointer variable which holds the address of an integer variable. The address of a variable can be accessed through the & operator (*address operator*); `&a` is the address of variable `a`. Since `pa` is a pointer variable which can hold the address of an integer variable, `pa = &a;` is a valid assignment statement. If the pointer variable is preceded by an *, then it refers to the variable it is pointing to: `*pa`, which in this case is the variable `a`. Pointers can also point to pointers.

[1]Contrary to popular belief, a pointer is not only an address. In fact there are two addresses associated with each pointer: the address it holds and the address of the pointer variable itself.



It is important not to confuse what `&ptr`, `*ptr` and `ptr` stand for. The *value* of a pointer is the address it holds. `&ptr` is the address of the pointer variable. Every variable has an address which can be queried by the `&` operator. Pointers are variables, hence the `&` operator can be applied. `*ptr` is the actual value which the pointer is pointing to. Two pointers can have the same value – *or store the same address* – but every variable (*hence every pointer*) itself has a different address. The case when two or more pointers point to the same location in memory is also referred to as *aliasing*. In that case, the value can be changed via either of the pointers. Besides aliasing, pointers also play an important role in parameter passing, dynamic memory allocation and run-time polymorphism as we shall see later. It is virtually impossible to write a "useful" C or C++ program without using pointers.



```
short int a = 3;
short int b = 5;
short int *pint1, *pint2;
short int **ppint1, **ppint2;

pint1= &a;
ppint1 = &pint2;        1

ppint2 = &pint2;
pint2 = &b;
*pint2 = 7;             2

**ppint1 = *pint2 + a;
*ppint2 = pint1;
*pint1 = **ppint1;      3
```

$\boxed{1}$

Address	Value	Symbol
0B2033		
0B2034		
0B2035	3	a
0B2036	00000	pint2
0B2037		
0B2038		
0B2039	0B2036	ppint1
0B203A		
0B203B		
0B203C	00000	ppint2
0B203D		
0B203E		
0B203F	0B2035	pint1
0B2040		
0B2041	5	b

$\boxed{2}$

Address	Value	Symbol
0B2033		
0B2034		
0B2035	3	a
0B2036	0B2041	pint2
0B2037		
0B2038		
0B2039	0B2036	ppint1
0B203A		
0B203B		
0B203C	0B2036	ppint2
0B203D		
0B203E		
0B203F	0B2035	pint1
0B2040		
0B2041	7	b

$\boxed{3}$

Address	Value	Symbol
0B2033		
0B2034		
0B2035	3	a
0B2036	0B2035	pint2
0B2037		
0B2038		
0B2039	0B2036	ppint1
0B203A		
0B203B		
0B203C	0B2036	ppint2
0B203D		
0B203E		
0B203F	0B2035	pint1
0B2040		
0B2041	10	b

C++ initializes pointers to store 0. This special address has a symbolic name *NULL*. C does not initialize pointers by default, hence it is good practice to explicitly set a pointer to *NULL*, if it does not point to anything. Pointers can be declared to point to any type of variable, object and even functions.

```
int *a;  // pointer to an integer variable
int **b; // pointer to a pointer which points to
         // an integer variable
int (*f1)(void);
         /* f1 is a pointer to a function, which takes
            no arguments and returns an integer */
void (*f2)(int, char*);
         /* f2 is a pointer to a function, which takes
            an integer and a pointer to a character
            arguments and returns no value */
```

Pointers to functions can be used to pass functions to a function via an argument.

C++ performs *type checking* on pointer variables; hence it is not possible to set (*at least without warning*) a pointer of type A to point to a variable of type B. However, there is a special pointer type which can point to all type of variables. A pointer variable declared as *void** can hold the address of any variable or function.

```
void* gp = NULL;
int *pi = NULL;
char *pc = NULL;
pi = pc; // error: pi and pc are of different types
gp = pi; // ok
gp = pc; // ok
pc = gp; // error: pc and gp have different types
```

3.3 Arrays

Arrays collect variables of the same type in one contiguous chunk of memory. In C and C++ , arrays are always indexed from *0* to *length - 1*. The following declaration defines `intarray` to be an array which can hold 10 integers.

```
int intarray[10];
```

In general, a statically allocated array is declared as:

type name-of-array $[d_1][d_2]...[d_n]$

where $d_1, ..., d_n$ are the dimension(s) of the array. `intarray[0]` is the first element of *intarray* and `intarray[9]` is the last element. It is *impossible* to set the dimensions of a statically allocated array runtime.

```
int matrix[10][5];
```

declares a two-dimensional array (*matrix*), where `matrix`[2][3] is the entry in the "third row" and "fourth column". It is also possible to initialize a statically allocated array at declaration:

```
  int vector[5] = { 1, 2, 3, 4, 5 };
  int matrix[3][3] = {
     {11,12,13},
     {21,22,23},
     {31,32,33}
  };
 char name[12] = {'S','a','n','t','a',' ','C','l','a','u','s',0};
```

A character array (*last declaration*) is treated as a string in C and C++ . A string is a *null* terminated character array and an alternative declaration which automatically inserts the *null* character is:

```
char name[12] = "Santa Claus";
```

When an array is initialized (*and hence defined*) at declaration, it is not necessary to provide all the dimensions.

```
int vector[] = { 1, 2, 3, 4, 5 };
int matrix[][3] = {
   {11,12,13},
   {21,22,23},
   {31,32,33}
};

char name[] = "Santa Claus";
```

However, in the case of a multidimensional array, only the first dimension may be omitted, because the compiler must know at compile time the exact size of statically allocated variables. In the above example, `matrix`[0][0] is the entry in the "first row" and "first column" and `matrix`[0] is the first "row". In fact, `matrix`[0] can be assigned to an `int*`. However, an `int`[] is not the same as an `int*`. `int`[] is the type that holds the address of a unique statically allocated array variable, while `int*` is a type that holds the address of any integer variable. For this reason, it is always possible to assign an `int`[] to an `int*`, but it is not possible to assign an `int*` (*or another* `int`[]) to an `int`[]. A type `int`[][] does not exist, hence `matrix` cannot be assigned to a `int**`. With the `sizeof` operator, the programmer can query the size of a variable. Using the above definition of variable `matrix`,

```
  int*  pr;
  pr = matrix[1];
       // ok, pr points to the first element of the second row
  pr = &matrix[1][1];
       // ok, pr points to the second element of the second row
  matrix[1] = pr;
       // error, matrix is of type int[], hence it cannot be assigned

  cout << "matrix: " << sizeof(matrix)/sizeof(int) << endl;
  cout << "matrix[0]: " << (sizeof(matrix[0])/sizeof(int)) << endl;
  cout << "sizeof(pr): " << sizeof(pr) << endl;
```

The output of the program is:

```
matrix:  9
matrix[0]:  3
sizeof(pr):  4[2]
```

This allows the programmer to check the size of a statically allocated array (*ie: of type* `type`$[d_1][d_2]...[d_n]$).

Character arrays can also be created at declaration with initialization by assigning to a pointer. The difference is that the "size" information is lost[3]:

```
  char * n1 = "Santa Claus";
  char  n2[] = "Santa Claus";
  cout << "n1: " << sizeof(n1) << endl;
  cout << "n2: " << (sizeof(n2)/sizeof(char)) << endl;
```

The output of the program segment:

```
n1:  4
n2:  12
```

The string "Santa Claus" has only 11 characters including the "space", however the compiler automatically inserts a *null* character.

Pointer Arithmetic

It is also possible to perform simple *arithmetic* on pointers. For example:

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

[2]the size of a pointer is operating system dependent, but it is most likely to be 4 bytes (or 32 bits). DOS used to have 16 bit pointers, while some operating systems already use 64 bit ones

[3]more accurately, the size is the size of a `char*`, or the size of a pointer variable

```
int *ptr;
ptr = array; // or ptr = &array[0]
cout << *ptr << ' ' << *(ptr+3) <<
           ' ' << ptr[3] << endl;
```

gives

```
1 4 4
```

This means that `ptr` was moved 3 × `sizeof(int)` many bytes. For a pointer `ptr` of type `A`, the expression `(ptr + k)` evaluates to the *address* `(int)ptr + k * sizeof(A)`. The notation `*(ptr+3)` and `ptr[3]` mean the same thing and usually the second one is used. In general, for any pointer variable `ptr` and integer n, the following are equivalences:

$$\texttt{ptr} + n \equiv \&\texttt{ptr}[n]$$

$$*(\texttt{ptr} + n) \equiv \texttt{ptr}[n]$$

3.4 Dynamic Memory Allocation

The main use of pointers, besides aliasing, is associated with dynamic memory. Dynamic memory is allocated at *run time* while static memory is allocated at *compile time*. In many applications, it is impossible to know ahead of time how much memory is needed. Using dynamic memory, however, is a potential source of hard-to-fix problems. As a dynamic memory chunk does not have a symbolic name, the address of its first byte must be saved in a *pointer*. If that pointer is lost, there is no way to *deallocate* the memory chunk. If this is done in a loop or in recursive calls, so much memory can be lost that it halts the system. This situation is referred to as having a *memory leak*. When dynamic memory is not needed anymore, it should be returned to the main pool, so successive allocations can reuse the space. C++ provides two operators for manipulating dynamic memory. Operator *new* allocates a new variable and returns its address, while operator *delete* returns the memory chunk pointed to by the pointer.

```
int *a; // pointer to integer
a = new int;
   // a now points to a valid address
*a = 5;
delete a;
   // memory associated with a is deallocated
*a = 3;
   // this may crash your computer!!!!
a = new int;
```

```
     // the address of a new memory chunk is in a
  a = new int;
     // again, but the previous chunk is forever lost!!!
  delete a;
```

To allocate and deallocate contiguous memory, operator *new[]* and operator *delete[]* can be used.

```
int *a; // pointer to integer
a = new int[10]; // a contains the address of an integer array
a[2] = 5; // third element is set to 5
delete [] a; // all memory associated with a is gone
```

It is important that `delete`[] is used to return contiguous memory, because `delete` and `delete[]` work differently. Also, the memory allocated by `new` and `new[]` are slightly bigger than if they were statically allocated because the size information must be kept together with the memory chunk. Later we will see that the meaning of *new* and *delete* can be overloaded and in fact can be customized by a skilled programmer.

Allocating multidimensional arrays is a bit more complicated. Ideally we would like to write `int **a = new int`[3][4]; to allocate a 4×3 matrix, however C++ does not support such mechanism (Java *does*). There is still a somewhat less elegant way to allocate the 3×4 matrix run time:

```
int **a; // pointer to the 2 dimensional array
a = new int*[3]; // allocate pointers to the 3 rows first
for(int i=0;i<3;i=i+1)
   a[i] = new int[4]; // each row has 4 columns
a[2][3] = 6;
   // setting the entry in the third row and fourth  column
for(int i=0;i<3;i=i+1)
   delete [] a[i]; // delete rows first
delete [] a;
```

Why the above actually works can be derived from *pointer arithmetic*. Recall that

$$*(\texttt{ptr} + n) \equiv \texttt{ptr}[n]$$

Extending the above formula recursively for 2 (*and possibly more*) dimensions for pointer `ptr`[4] and integers n and m:

$$*(*(\texttt{ptr} + n) + m) \equiv \texttt{ptr}[n][m]$$

[4]`ptr` has to be of the appropriate type, or one ”extra” * for each dimension

and in general

$$*(...(*(*(\texttt{ptr} + n) + m)...) + r) \equiv \texttt{ptr}[n][m]...[r]$$

Dynamic Memory Allocation C Style

C has a more explicit approach to dynamic memory allocation and although it is still available in C++ , these functions should only be used to actually *overload* or *implement new* and *delete*. The functions provided by standard C are:

```
void* malloc(size_t bytes)
void* calloc(size_t n, size_t bytes)
void* realloc(void* ptr,size_t bytes)
void free(void* ptr)
```

`malloc` can be used to allocate *byte* many bytes and it returns a pointer to the newly allocated chunk. `calloc` is essentially the same, except it allocates *bytes* $\times$ *n* many bytes. `free` returns memory associated with pointer *ptr*. `realloc` can be used to change the size of the memory location pointed to by *ptr* to be *bytes* many bytes large. `size_t` is a numeric type and its extent may vary from compiler to compiler. In most cases, it is the same as an *unsigned int*. `realloc` changes the size of the block referenced by *ptr* to *bytes* many bytes and returns a pointer to the possibly moved block. The contents will be unchanged up to the lesser of the new and old sizes.

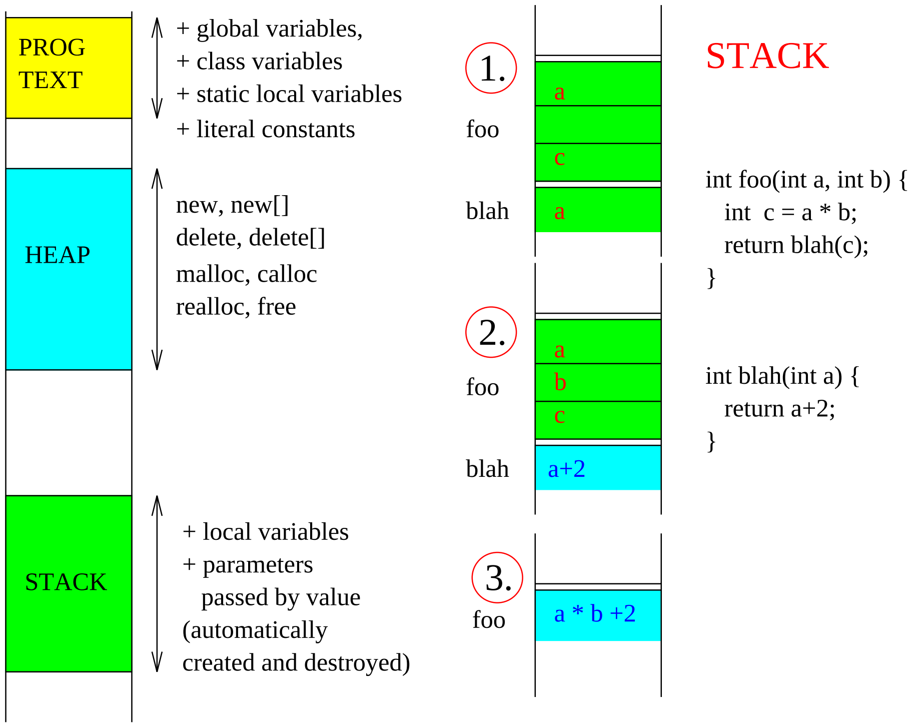
```
int *a;   // a is a pointer to an integer
a = (int*)malloc(sizeof(int));
          // new memory for one integer is  allocated
*a = 3;
free(a);  // memory returned
a = (int*)calloc(5,sizeof(int));
          // array of 5 elements created
a[3] = 4; // fourth element set to 4
a = (int*)realloc(a,sizeof(int)*10);
          // size of a changed from 5 to 10
free(a);
```

The C++ *new* and *delete* operators should never be mixed with the C functions shown above. It is a serious mistake to deallocate memory created with *new* using *free*[5]

[5]the reason is that `new` and `new[]` call the constructor while `malloc, calloc` and `realloc` don't. Similarly `delete` and `delete[]` call the destructor, while `free` doesn't.

3.5 Run-time, Compile-time and Automatic Memory Allocation

There are three fundamentally different ways of allocating memory. These are compile-time (or global or static), automatic (also local or "on the stack") and run-time (or dynamic or "on the heap"). Compile-time allocated variables are physically part of the program text (or compiled program). Global variables, class variables (*we discuss them later*), `static` local variables and literal constants (such as `"hello world"`) are compile-time allocated and are part of the executable image.



Automatic variables are created and destroyed as the program is running and are implicitly managed. They are created on the stack and they must be of fixed size. Whenever a function is called, sufficient space is allocated for parameters which are passed by value and for all non-static local variables. When the function is finished executing, these variables are automatically destroyed. If the function returns a value, that return value is left on the stack to be picked up by the function that initiated the call. In the figure above, the first stack layout shows two frames on the stack, `foo` and `blah`. Currently `blah` is executing. After it finished, its frame is popped and its return value stays there for `foo` to pick it up (second stack layout). Finally when `foo` has finished, its stack frame is popped and the return value is ready to be picked up by the caller[6].

Dynamic memory is allocated at run-time in a separate area of memory, often referred to as *heap.* This area of memory is explicitly managed by the programmer by the `new`, `new[]`, `delete`, `delete[]` operators in C++ and the `malloc`, `calloc`, `realloc` and `free` functions in C . The constructor and destructor mechanisms may implicitly be invoked and instantiate and/or destroy memory chunks on the heap. Some environments (Java , SmallTalk) implement a garbage collector mechanism which asynchronously manages the heap.

3.6 Parameter Passing (*simple types, pointers, arrays*)

C++ provides a rich variety of parameter passing and returning mechanisms. These are *passing by value*, *passing by reference* and *passing the address.* Similarly, a function can return a *value*, a *reference to a value* or an *address.*

Passing by value.

When a parameter is passed *by value*, a local copy is created on the call stack. Hence any modification made to the parameters passed by value, is made on a local copy. This can be illustrated on the famous *swapping* problem.

```
void swap(int a,int b) {
   int tmp;

   tmp = a;
   a = b;
   b = tmp;
}
```

Inside the body of the function, we only swap the local copies. This problem is solved by the other two methods.

Passing by reference.

When a parameter is passed *by reference*, there is **no** copy created. The parameters are *aliases* to the actual variables passed, hence any modifications made to them is really made on the real ones.

[6]usually stack frames are implemented such that the space of for the return value is at the beginning (*or low address*) but – in theory – the exact layout of a stack frame is irrelevant.

```
void swap(int& a,int& b) {
   int tmp;

   tmp = a;
   a = b;
   b = tmp;
}
```

Do not confuse the & symbol with the address operator. It is **not true** that the address is passed! There is no variable created on the stack. This is the fastest parameter passing method, because there is no parameter passing, the *reference* is just another name for the actual variable. This mechanism is a new feature to C++ and is not available in C .

Passing the address.

A pointer is a variable which contains an address. In essence, passing the address in a pointer is passing a pointer variable by value. A new pointer variable is created on the call stack and it points to the same address as the pointer variable passed.

```
void swap(int* a,int* b) {
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}
```

If we declared two integer variables, `int d,e;`, the first and second versions of *swap* are called like:

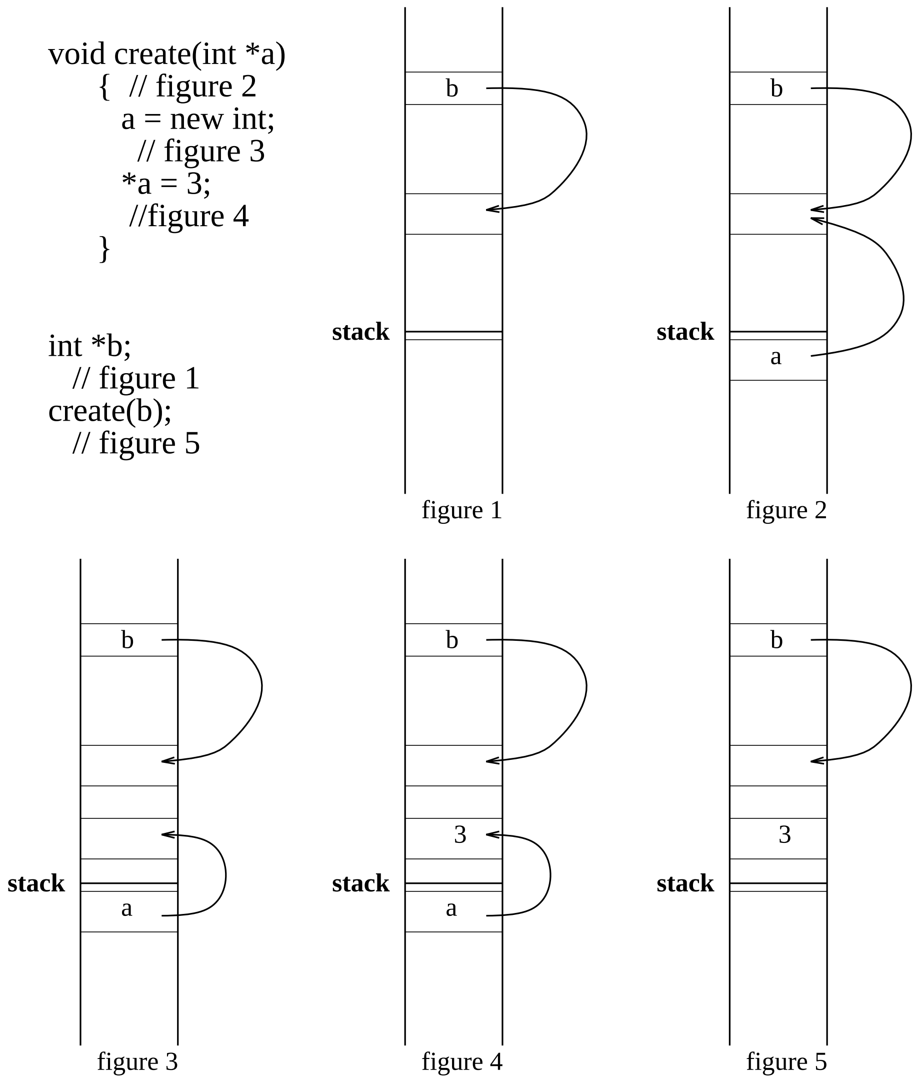
`swap(d,e);`

Hence just by the calling convention, *passing by value* and *passing by reference* are indistinguishable. Calling the third version, *passing the address* has the expected syntax:

`swap(&d,&e);`

Passing the address has the advantage of making it syntactically explicit, that the values referenced by the pointer variables may change. However one must be careful; the following does not work:

```
void create(int *a) {
   a = new int;
   *a = 3;
  }
...

int *b;
create(b);
cout << *b << endl; // the computer probably crashes
delete b; // if it did not crash yet, it may crash now
```



```
void create(int *a)
     {  // figure 2
        a = new int;
          // figure 3
        *a = 3;
         //figure 4
     }


int *b;
  // figure 1
create(b);
  // figure 5
```

Let us take a closer look why the above example does not work. When a function is called, a copy of the variables *passed by value* are created on the stack. Any modifications made to these variables are only made to the local copies. When the function

returns all local variables and local copies of parameters are destroyed. *Passing the address* of a variable actually involves passing a pointer *by value.*
There are two remedies, passing the address of the pointer as opposed to the address it holds, or passing the pointer by reference.

```
void create(int** a) {
   *a = new int;
   **a = 3;
}
```

In this case we would call `create(&b);`

```
void create(int*& a) {
   a = new int;
   *a = 3;
}
```

and the above version can be invoked as: `create(b);`

Arrays are always passed by address, regardless which notation we use. However the different notations force different type checking to be performed. For example:

```
void foo1(int z[2][2]) {
}

void foo2(int z[][2]) {
}

// variables of type int[k][2] can be passed, where
// k does not have to be known at compile time
// some compilers may complain if k != 2 for the former

void foo3(int z[3]) {
}

void foo4(int z[]) {
}

// variables of type int[k] can be passed, where
// k does not have to be known at compile time
// some compilers may complain if k != 3 for the former

void foo5(int *z) {
   // variables of type int[k] and even simple addresses
```

```
      // of integers can be passed, k arbitrary
   }

   int a1[2][2] = { { 11, 12}, { 21, 22} };
   int a2[3] = { 1, 2, 3};

   foo1(a1);
   foo2(a1);
   foo3(a2);
   foo4(a2);
   foo4(a1[1]);
   foo5(a2);
   foo5(a1[1]);
   foo5(&a1[0][1]);
```

When the `return` statement with a value is called, a *new* copy is returned unless it is returned by reference. The former return mechanism is called return *by value.* Hence returning the address of a local variable or returning a reference to a local variable is an error because as soon as the function terminates its space could be reused.

```
int& bug1() {
   int a = 7;
   return a; // a's space is lost!!!
}

int* bug2() {
   int a = 7;
   return &a; // a is gone!!!
}

int* ok() {
   int* a;
   a = new int;
   *a = 7;
   return a;
}
```

The last function, `ok()` works. `a` is a local variable, however the memory chunk returned by `new` is not on the stack, hence it will not be destroyed. When the `return` statement is called, a new copy of the variable `a` is made. As `a` is a pointer variable which holds the address to the dynamic memory piece the copy of `a` returned will also hold the same valid address.

Arrays can only be returned by returning the address in a pointer. For example, `int[] foo()` is not a valid prototype; it should be `int* foo()`.

We have seen how to declare pointers to functions. Functions can be passed *by address* to another function.

```
void map(int array[], int size, void (*f)(int&)) {
   for(int i=0; i<size; i++)
     f(array[i]);
}

void triple(int& a) {
   a = a * 3;
}

int X[] = {1 , 2 , 3, 4, 5};

map(X,5,triple);
```

Even the program itself can receive arguments form the command line. `main` has two forms:

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

The second form takes two parameters, *argc* (*argument count*) holds the number of arguments passed to the program. *argv* (*argument vector*) is an array of character strings which are themselves the arguments. `argv`[0] is the name of the program.

```
#include <iostream.h>

int main(int argc, char* argv[]) {
   cout << "you called " << argv[0] << " with "
        << "these arguments: " << endl;

   for(int i=1; i<argc; i++)
      cout << argv[i] << ' ';

   cout << endl;

   return 0;
}
```

If the program were compiled to `prog` and you called `prog 4 monkeys` then the output is:

```
you called prog with these arguments:
4 monkeys
```

In C++ , (*and not in* C), the programmer can provide defaults for parameters. If the parameters with the default values are not passed, then the default values are used.

```
  void foo(int a, int b=1,const char* c="blah") {
     cout << a << ' ' << b << c << endl;
  }

  foo(2,3,"hello"); // prints 2 3 hello
  foo(2,3);         // prints 2 3 blah
  foo(2);           // prints 2 1 blah
```

If an argument in the parameter list is provided with a default value then all parameters following that one must have a default value as well. Abusing this mechanism can lead to ambiguity; that is, the compiler may not be able to resolve a call if more than one functions match the call with the missing parameters.

3.7 Constants

A rudimentary way to declare constants is to use the preprocessor. The preprocessor is run on the code before it is passed to the compiler. To communicate with the preprocessor, *preprocessor directives* must be inserted into the code. The `#define` directive can be thought of as a global *search and replace* in a word processor. For example the file:

```
#define TOOL FOOL
#define e 2.71828
TOOL
2*e+1
```

after running the preprocessor, becomes

```
FOOL
2*2.71828+1
```

Hence, `#define pi 3.14159`, simulates the effect of having a constant for π. C++ and ANSI C , on the other hand provides a much more sophisticated and elegant way for declaring, passing and returning constants.

In C++ , the `const` keyword can be used to declare constants. By definition a constant cannot change value, so it must be initialized at declaration:

```
const int a = 5; // a is an integer constant
const char[] prompt="Please enter a value"; // string constant
const double i[] = { 1, 0, 0 };
const double j[] = { 0, 1, 0 };
const double k[] = { 0, 0, 1 };
   /* the three unity vectors in 3D */
```

Pointers can also be declared constants. The question is, whether the pointer is constant or the value it is pointing to. C++ provides all combinations.

```
const int a = 2;
c = 4;
const int *pc;
   // pointer to a constant value
int * const cp = &c;
   // a constant pointer, must be initialized
const int * const cpc = &a;
   // a constant pointer to a constant
   // must be initialized

pc = &a; // ok, pc can be assigned
*pc = a;
   // error: pc is pointing to a constant
*cp = a;
   // ok, the value cp is pointing to can be assigned
cpc = &a;
   // error: cpc is a constant pointer, hence cannot be assigned to
*cpc = a;
   // error: cpc is pointing to a constant
```

Only a pointer to a constant can hold the address of a constant. A constant pointer cannot be assigned to, hence it cannot change the address it holds. Modern operating systems provide a *memory mapping* mechanism for devices (*such as the serial port*). Because the address of the device is static, a constant pointer can guarantee that the address is not lost.

Constants also play an important role in parameter passing. It can be specified at the parameter list that the argument is a constant, and the compiler would signal an error if the argument is on the left hand side of an assignment statement or is passed to a function or method which does not specify the parameter to be `const`.

```
void foo(const int& a) {
   a = 3; // error: a is a constant
}
```

3.8 Operators and Precedence

C++ has a rich set of operators. Most can be grouped into the following subclasses: *relational*, *arithmetic*, *increment*, *assignment* and *bitwise logic* . Some of these we have already seen and some will be mentioned later as we encounter them. Most of these operators can be *overloaded*, meaning that the programmer can implement custom versions.

Arithmetic operators:

+	addition
+	unary "+"
–	subtraction
–	unary "-", *"negative"*
*	multiplication
/	division
%	modulus

Relational operators:

Relational operators return 0 for *false* and a non-zero value (*usually 1*) otherwise. In ANSI C ++ there is a true boolean type `bool` with values `true` and `false`. If available they are used instead. In either case, `false` and `true` are implicitly typecasted to 0 and 1 respectively in such a context.

==	equal
!=	not equal
!	not
<	less
<=	less or equal
>	bigger
>=	bigger or equal
&&	and
||	or

For example, using the famous *De Morgan* laws, `!(a && b) == !a || !b` and `!(a || b) == !a && !b`.

Increment operators:

C and C++ provide increment operators.

++	post- and pre-increment
––	post- and pre-decrement

```
int a,c;
c = a++; // equivalent to c = a; a = a + 1;
c = ++a; // equivalent to a = a + 1; c = a;
c = a--; // equivalent to c = a; a = a - 1;
c = --a; // equivalent to a = a - 1; c = a;
```

In C++ these operators can be overloaded to operate on any object.

Assignment operators:

=	assignment
+=	a += b is equivalent to a = a + b
–=	a –= b is equivalent to a = a – b
*=	a *= b is equivalent to a = a*b
/=	a /= b is equivalent to a = a/b

There are a few more, related to bitwise operations, scope resolution, etc ...

&	address operator
sizeof	number of bytes to store the variable
`[ ]`	array subscript
`new`, `new[]`, `delete`, `delete[]`	memory management

The precedence order of these operators:

postfix `++`, `--`, []
prefix `++`, `--`
`unary +`, `unary -`, `new`, `new[]`, `delete`, `delete[]`
`!,*,/,%`
`+, -`

Parentheses have higher precedence than these operators, hence they can be used to overwrite the default. For example, 3 + 2 * 2 by default is resolved as 3 + (2 * 2) as opposed to (3 + 2) * 2 (** has higher precedence than +*).

3.9 Defining Types with `typedef`

Both C and C++ allow the creation of user defined types. Using the `typedef` keyword the programmer can create new types. The syntax is identical to variable declarations, except for the preceding `typedef` keyword:

```
typedef int vector3d[3];
   // vector3d is a new type, not a variable
   // if it were declared: int vector3d[3]
   // then it would be a variable
vector3d i = { 1, 0, 0 }; // i is a variable of type vector3d

void foo(vector3d v) // parameter passing
...
```

`vector3d` is a *type* not a *variable*! `typedef char* string` – *for example* –, declares a new *type*, which in essence is a `char*`. User defined types can be arbitrarily complex.

3.10 Typecasting

Typecasting stands for changing the interpretation of a variable. This is sometimes done implicitly.

```
void foo(double d) {
   ...
}

int a = 2;
foo(a);
```

The compiler implicitly converts `a` into a `double`, as opposed to giving an error that the types do not agree. For numeric types this conversion is implicit and is applied when needed. However, sometimes it is useful to explicitly typecast one type for another. For example, a general memory allocation routine must return `void*`, however a `void*` cannot be assigned to, let's say, an `int*`. In this case, an explicit typecast is needed. The syntax of typecasting in C and C++ :

`( type ) expression`

Examples:

```
double pi = 3.14159;
int a = (int) pi; // a is 3
int * a = (int *) malloc(sizeof(int));
```

C++ also provides safer and more elegant ways of performing type casts by the `static_cast`, `dynamic_cast` and `reinterpret_cast` mechanisms. `const_cast` can be used to "cast" the `const` away or actually to modify the value of a constant without compiler warnings.

```
void foo(void* context) {
   int *a = static_cast<int*>(context);
   // equivalent to (int*) context
}

void foo(Student* s) {
   PartTimeStudent* st = dynamic_cast<PartTimeStudent*>(s);
   // NOT equivalent to (PartTimeStudent*)s !!
   // if the actual instance pointed to by s is not
   // also an instance of the subclass PartTimeStudent
   // st is set to NULL!
}

void foo(int address) {
   double* a = reinterpret_cast<double*>(address);
   // NASTY! put the actual value "address" into the pointer
   // system work sometimes demands such things
   // equivalent to (double*)address
}

void foo(const int* a) {
   *(const_cast<int*>(a)) = 3;
   // cast the const away and modify the value pointed to by a
}
```

3.11 Examples of Variable Declarations

This section demonstrates how nasty variable declarations can get. While rare in practice, complicated variable declarations and type definitions do occur in C and C++ programs and one must be aware of them.

```
int a[5][4][6];
```

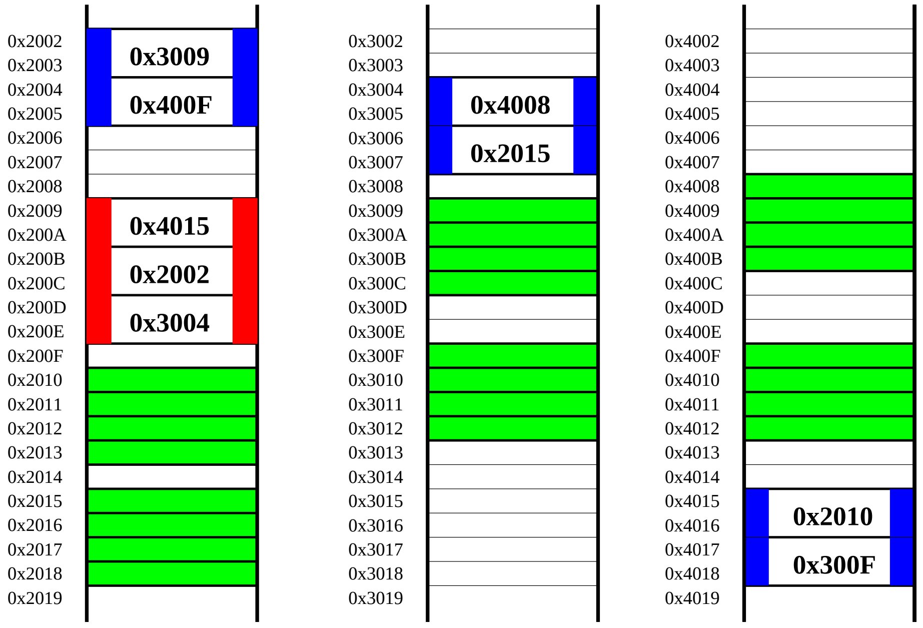
A $5 \times 4 \times 6$ array.

`int ***a;`

A pointer which points to a pointer which in turn points to an integer. Such a variable can point to a 3 dimensional structure.

```
short int ***a; // 3 x 2 x 4 array
a = new short int**[3]; // depth
for(int i=0; i<3; i++) {
   a[i] = new short int*[2]; // height
   for(int j=0; j<2; j++) {
      a[i][j] = new short int[4]; // width
   }
}
```

The layout of such a beast in memory looks something like this: (*Assuming that a pointer takes 2 bytes and a* `short int` *needs one byte. Also, there is some extra space required for the arrays to store size information and the actual address alignment should reflect the word size, which we ignored here. The point is to observe the layout and which pieces are contiguous and which are not!*)



```
int (*f) (int, char*, double = 6);
```

`f` is a pointer to a function which takes an integer, a character string and optionally a double with default value 6 and returns an integer.

```
int foo1(int i, char* name, double d = 6) {
    /* ... */
}

int foo2(int i, char* name, double d = 6) {
    /* ... */
}

f = foo1;
f(2,"hello");      // calls foo1(2,"hello",6)
f = foo2;
f(3,"bye",-2.2);   // calls foo2(3,"bye",-2.2)
```

```
bool (*f) (double*, int, double (*)(double&, void* = NULL), void* = NULL);
```

`f` is a pointer to a function which takes a `double*`, an integer, a function which takes a `double` by reference and an optional `void*` and returns a `double` and an optional `void*` and returns a boolean.

```
double triple(double& d, void* = NULL) {
   return d *= 3; // triple d
}

double func(double& d, void* context = NULL) {
   double b = context == NULL ? 1
                       : *(static_cast<double*>(context));

   return b * d;
}

bool map(double* array, int l,
   double (*foo)(double&, void* =NULL), void* context = NULL) {

   for(int i=0; i<l; i++) {
      foo(array[i],context);
   }

   return true;
}

bool foo(double* array, int l,
   double (*g)(double&, void* =NULL), void* context = NULL) {

   if (l == 0) return false;

   double s = 0;

   for(int i=1; i<l; i++) {
      s += g(array[i],context);
   }

   return s > 300;
}

...

double a[] = { 1, 4, -2, 5, 3 };
double b = 10;

f = map;
f(a,5,triple);  // all values of a are tripled
                // a = { 3, 12, -6, 15, 9 }

f = foo;
if (f(a,5,func,&b)) {
   cout << "YES" << endl;
} else {
   cout << "NO" << endl;
}

// (3*10 + 12*10 -6 * 10 + 15*10 + 9 * 10 == 330) > 300
// it writes YES
```

```
int (*A[3])(char);
```

A is an array of three pointers, where each pointer points to a function which takes a char and returns an int. It is an array of pointers rather than a pointer to an array because [] has higher precedence than *. Explicitly parenthesized:

```
int (*(A[3]))(char);
```

An example:

```
int f1(char c) {
   return c > 'a' ? -1 : 1;
}

int f2(char c) {
   return c < 'z' ? -2 : 2;
}


int (*A[3])(char) = { f1, f2, f1};

int main(void) {

    cout << A[1]('k') << ' ' << A[0]('k') << endl;
    return 0;
}
```

3.12 Notes on Syntax and Semantics

Pointers are declared by putting stars between the type specifier and the identifier. Many pointers and arrays can also be declared on the same line as a comma separated list. As far as the star(s) are in between the type and the identifier, the declaration is syntactically correct. For example

```
int *a, * b, **c,
    * * d, e[2], *f[3], (*g)[3];
```

a and b are pointers to integers, c and d are pointers to pointers to integers, e is an array of two integers, f is a pointer to an array of three integers and g is an array of three integer pointers. To distinguish between `f` and `g`, one must know that `[]` has higher precedence than `*`, hence `f` is implicitly `int *(f[3])` rather than `int (*f)[3]`. Function names represent the addresses where the functions are actually stored.

```
int f(int a) {
   return a*a;
}

int (*p)(int);

...
p = f;

int k = f();
```

`f` is a function stored at address `f` and `p` is a variable which can hold the address of a function which takes an `int` and returns an `int`. Hence the assignment `p = f` is valid because `f` is the address where the function is stored. `k = f()` on the other hand is the invocation of the function stored at address `f`. After the assignment of `f` to `p` it could have been `k = p()`.
One must note that there is some inconsistency here! If `p` is a pointer to a function, why didn't we write `p = &f` instead. Or calling `f` through `p`, why do we write `p()` rather than `(*p)()` – *as if* `p` *is the address of the function, then* `*p` *must be the function.* A function can only do two things: it can be called or its address can be taken. Hence compilers provide some leniency with respect to notation:

```
p = f; // OK
p = &f; // ALSO OK

int k = p(); // OK
int k = (*p)(); // ALSO OK
```

Functions are often passed as arguments – even more so in C than in C++ .

```
void execute(void (*callback)(void*), void* context) {
    // do something
    ...
    // when finished execute callback in context
    callback(context);
}
```

```
void cleanup(void* context) {
   Context * ptr = (Context*)context;
   // do some resource deallocation
   ...
   ptr->finished = true;
}


....
execute(cleanup,this_context);
```

`execute` is a function whose purpose is to *asynchronously*[7] perform some job, however it should communicate the result when it is finished. To do so, it calls the function passed as an argument in `callback` in the context passed in `context`. A variable of type `void*` can point to any kind of type or class. So we pass `cleanup` who typecasts `context` to a pointer to a `Context` object which possibly amongst others has a boolean variable which is set to true when the job is finished. This kind of callback mechanism is often used in distributed programming and in particular in Telecommunications. In the object oriented paradigm it is done better by passing an object which incorporates the context and it calls a method on itself. Since the introduction of declared interfaces in Java , such classes do not even have to belong to the same hierarchy.
It is vital to understand the difference between *passing by value*, *passing the address* and *passing a reference.* While this lecture has emphasized the importance, it becomes even more pronounced – as we shall see later – when objects are passed and constructors and destructors may be implicitly invoked. It is not true that when passing by reference the address is passed. In fact, what really happens is that the reference becomes an alias for the actually passed variable and when the program is compiled any reference to the alias is replaced by a reference to the originally passed identifier. This explains why local variables cannot be returned by reference: the variable on the stack is automatically cleaned up and hence the variable returned does not exist.
Pointers can also be passed by reference – and often they *should be.*

```
void f1(int *a) {
   a = new int;
}

void f2(int *&a) {
   a = new int;
}
```

[7]or in essence *parallel*

We have already seen this example and the memory trace of what goes wrong with `f1`. The important thing to understand is that `a` is just a variable which holds the address of an integer. As such it exists on the stack and is initialized to store the actual address passed to it. When `a = new int` is executed, the address stored in this local `a` has changed but now it is out of synch with the original variable passed to it which still points where it used to. On the other hand the `a` in `f2` is also a variable which holds an address, but this `a` is not on the stack, it is actually physically the same as the variable passed to `f2` with *another* name. It is very important to understand this concept, so take the time and type in the following program to convince you:

```
#include <iostream.h>

int *b;

int f1(int *a) {
   cout << "f1" << endl;
   a = new int;
   cout << " a = " << (void*)(a)  << "  b = "
        << (void*)(b) << endl;
   cout << "&a = " << (void*)(&a) << " &b = "
        << (void*)(&b) << endl;
}

int f2(int *& a) {
   cout << "f2" << endl;
   a = new int;
   cout << " a = " << (void*)(a)  << "  b = "
        << (void*)(b) << endl;
   cout << "&a = " << (void*)(&a) << " &b = "
        << (void*)(&b) << endl;
}

int main() {
    cout << "b = " << (void*)b
         << " &b = " << (void*)(&b) << endl;
    f1(b);
    f2(b);

    return 0;
}
```

When memory is allocated dynamically by the `new` or `new[]` operator it is very important that the appropriate version of `delete` is called (*ie* `delete` *or* `delete[]`). The reason is that there is type and size information kept that `delete` and `delete[]` read to deallocate the memory chunk. Since `new` and `new[]` return addresses which are usually saved in pointers, neither version of `delete` can tell from the argument pointer variable if it holds the address of an array or not. For some time in earlier versions of C++ there was only `delete` and no `delete[]`. In ANSI C++ there is a difference and not using the correct version of the operator can lead to memory leaks and crashes as well.

3.13 Exercises

3.1 Variables of type `void*` can point to any type of variable. Would a type `void`[] make sense? Explain!

3.2 Would the following swap procedure work?

```
void swap(int *a, int *b) {
   int *tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

Explain! (*draw a memory trace!*)

3.3 Write a program, which reads an n dimensional vector v of *doubles* and reads A an $n \times n$ matrix (*linear transformation*) and prints the resulting vector after applying the transformation A (*ie,* $A \times v$). n, the dimension, should also be read from the keyboard, and A and v should be dynamically allocated.

3.4 Write a procedure, which returns a pointer to the transpose of an $n \times m$ dynamically allocated matrix.

3.5 Write a program `sort`, which is passed an arbitrary number of reals at the prompt and it prints them in non-decreasing order. For example `sort 3 1 24 0 2.3` should print `0 1 2.3 3 24`. You can use any sorting algorithm you know.

Chapter 4

Classes I.

4.1 Aggregate Types

Most programming languages provide some form of *aggregate type* or *record type*. The C and C++ records fundamentally differ. In C , the struct keyword can be used to declare records, in C++ it is used to define a *class*.

```
struct foo {
   int a;
   char* name;
   foo* link;
} S;
```

The above definition, in C , declares a variable S with three fields: a of type integer, name of type character string, and a *link* field which as a pointer to the same kind of foo structure. The link field demonstrates that structures can be self referential. The *type* of S is "struct foo". The fields can be accessed by S.a, S.name and S.link respectively. C also has a *union*. A *union* in C is declared like a struct, but using the union keyword instead. The fields of the union share the same data space. In other words, memory referenced by the fields overlap. This mechanism can be used to save space when it is known that only one of the fields make sense or it can be used as an alternative to typecasting.

```
union {
   int i;
   double d;
} U;

U.i = 100;
cout << U.i << " : " << U.d << endl;
U.d = 100;
```

```
  cout << U.i << " : " << U.d << endl;
```

On my computer the output is:

```
100 :  2.122e-312
1079574528 :  100
```

4.2 The C++ Class

The *class* is more than a C record. It combines *data* and *methods*. In C++ , and unlike in C , the keywords `struct`, `union` and the new keyword `class` always define a class. An object is a particular instance of the class, which can be created statically at declaration and dynamically using the `new` operator. The C++ class provides a mechanism for information hiding and encapsulation so that objects can be accessible through a public interface and the internals could be private to the class and its derived classes. **Member functions can see all instance variables and methods.**

```
 class A {
    private:
       int a;
    public:
       void set_a(int);
       int get_a();
  };

  void A::set_a(int i) {
     a = i;
  }

  int A::get_a() {
     return a;
  }

  A a; // ``A'' is the class and ``a'' is the instance

  a.a = 2; // error: a.a is private!
  a.set_a(2); // ok
  cout << a.get_a() << endl;
```

`A` is a class with one instance variable `a`. `a` is *private* which means that it cannot be dereferenced directly. The two methods, `set_a` and `get_a` are *public* so they can be accessed anywhere and they also have access to all the private variables of the same class. `private` and `public` explicitly set the access right for the methods and variables following the keyword until another access modifier keyword is encountered in the same class definition. By default, every method and variable of a class is *private*[1]. The methods are declared by putting the *prototype* in the class definition. Later, when they are actually implemented, it must explicitly be stated which class they belong to. This is achieved by the `::` *scope resolution operator*. Methods can freely access the variables of the class, regardless of the access privilege specified. It is possible to make instance variables public, but *in general* it is considered to be a bad practice. There should be *set* and *get* methods to modify or inspect the value of an instance variable. Methods of the class can also be private; they can be used to implement public methods. A variable of a class which does not have a *set* method is *read only*, while a variable with no *get* method is *write only*.

When an object is dynamically allocated, its public methods and instance variables can be accessed via the usual `.` operator, or by the `->` operator.

```
A* ptr;
ptr = new A;
ptr->set_a(2); // equivalent to (*ptr).set_a(2)
cout << ptr->get_a(); // equivalent to (*ptr).get_a()
```

Information hiding is more than just hiding instance variables – it also facilitates protection and maintainability. Classes should be designed in terms of an interface: the programmer should first identify the methods that could be publicly available to be invoked on instances. It is quite a common misconception to believe that it is a useless effort to implement public *set* and *get* methods for an instance variable when in fact the variable could itself be made public. Once instance variables are public the structure of the class becomes open and unchangeable as the rest of the program may already rely on it. On the other hand if the instance variable is hidden behind an interface – like *set* and *get* methods – it can be changed. For example, consider the class `student` which most likely will include such properties as name, marks, *etc...* The first implementation would probably have instance variables representing these attributes. Suppose as the application grows students are stored in a database and information is no longer kept in run-time present instances but is looked up dynamically in the database. If the properties were properly hidden behind a public interface the instance variables can be changed without the rest of the application ever noticing. The *get* methods query the database while the *set* methods update it. To keep in mind that the structure of a class may change but how it is used is probably static is one of the driving forces of object oriented design and you should make an effort to always keep this in mind.

[1]in fact, the only difference between a class declared by the `class` keyword and a class declared by the `struct` keyword is that all members are *public* by default in the latter case and all members are *private* in the former case.

4.3 Creating, Passing, Copying and Destroying Objects

Objects are potentially complex structures with many data fields which themselves can be objects or even pointers either storing the address of dynamically allocated objects or used for aliasing. This raises the question how should new instances be initialized. Initialization gets even more complicated with inheritance. The derived class may have extra fields plus the ones it inherited from the base. How is the base going to be initialized? We have also seen that when a parameter is passed or returned by value a copy of it is created on the stack. Moreover, the stack is cleaned up automatically when local variables leave scope. In this section, we start by the constructor/destructor mechanism for instances of classes without inheritance and later we extend the picture with inheritance. Every object oriented language uses a very similar mechanism (*including* Java *and* SmallTalk) with the exception that destruction is achieved asynchronously by a garbage collector.

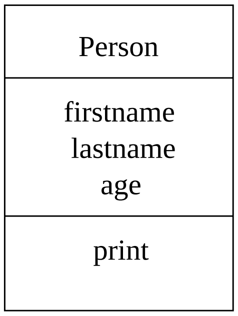
When a new instance of a class is created (*dynamically or statically*), a *constructor* is called to initialize the object. Constructors are supposed to initialize instance variables and establish associations. If there is no constructor, the compiler provides one. This, however, is seldom the one the application needs, specially when the class has dynamically allocated instance variables. There could be more than one constructors, each providing different parameters to initialize the object. The constructor always has the same name as the object and it returns no value. The constructor which takes no arguments is called the *default constructor*. If there is a constructor defined but it is not the default constructor, the compiler will not provide a default constructor! The constructor which takes the same class object is called the *copy constructor*. The copy constructor must take this instance by reference (*why?*)! The default constructor is implicitly called when an object is declared and the copy constructor is automatically executed when the object is passed by value to a function or when an object is returned by value. When an object leaves its scope, it is automatically deallocated and the *destructor* method is implicitly called. If the class does not have a destructor then the compiler provides a default one. Again, the one automatically supplied may not be the one the application really needs. The destructor has the same name as the class, but it is preceded by the character $\sim$. The assignment operator can (*and in most cases should*) also be overridden for every class. When on object is assigned to another, this method will be executed as opposed to the default one.

The following is a declaration of the class `A` with a default constructor, destructor, copy constructor and assignment operator:

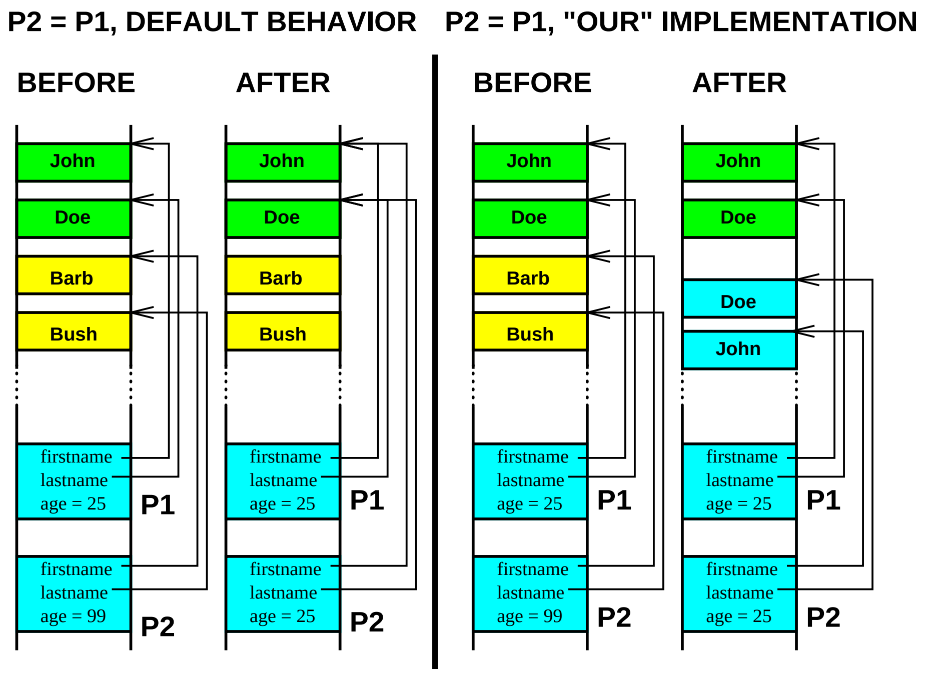
```
class A {
   public:
      A();          // Default Constructor
      ~A();         // Destructor
      A(const A&);  // Copy Constructor
      A& operator=(const A&);
                    // Assignment Operator
};
```

The constructors and the destructors are implicitly called when the objects are created and when they leave their scope or by the `delete` operator for dynamically created instances.

Why would someone implement a default constructor a copy constructor, an assignment operator and a destructor when the compiler provides one by default? The answer – as usual – is because of pointers. Pointers are used fundamentally in two ways: *aliasing* or sharing an object with other objects and storing the address of an object *dynamically allocated* on the heap. In the former case, if an instance shares an object with another instance, we would not want to have `delete` called on the pointer variable because it may be used by other instances. In the latter case, if it is known that the only pointer to the object is the one in question and it was dynamically allocated, we would want to have `delete` free its space. The compiler not knowing what the pointer really represents takes the conservative approach and does not call `delete` on pointer instance variables. Also, when one instance is assigned to another and it has pointer variables, the assignment operator by default makes the instance point to the same objects (*creating aliasing*) and would not call `delete` on the pointers before their values are overwritten (*potentially causing a memory leak*). Similarly, the copy constructor provided by default would copy the addresses into the pointer creating *aliasing*. Of course, it may be the case that the pointers are really only used to alias (*see iterators later*) but more often it is not the case and when a copy is created (*copy constructor and assignment operator*) you want to have `new` creating a replica of the object pointed by the pointer instance variables and when the object leaves its scope (*destructor*) `delete` should be called on the pointer(s). Very often it is also the case that some of the pointer instance variables are used as *aliases* and some aren't. In light of these scenarios, one should be convinced that it is very important to address how pointer instance variables are used and design the appropriate constructors, assignment operator and destructor. The following example illustrates the implicit mechanism for the class `person`. Every `person` object is associated with two pointer variables (*of type* `char*`) and an integer instance variables.



The figure below shows what the assignment operator provided by default would do and what our implementation does. The picture would be almost analogous for the copy constructor. If the pointer instance variable (*in our case* `lastname` *and* `firstname`) is used to store the address of a *new* dynamically created object, then the proper behavior of the copy constructor is to *create a new object* and initialize that object so it has the same settings. The assignment operator must also destroy the dynamically created objects first before it assigns the new ones.



In the implementation below the assignment operator and the copy constructors create carbon copy replicas and the destructor frees the memory allocated for `firstname` and `lastname`.

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

int lineno = 0;

class person  {
   private:
      char * firstname;
      char * lastname;
      int    age;

      void copy(char* = NULL,char* = NULL,int = -1);
                   // A PRIVATE METHOD, WHICH CAN TAKE
                   // 0, 1, 2 OR 3 ARGUMENTS.
      void destroy();
                   // A PRIVATE METHOD, WHICH DEALLOCATES
                   // DYNAMICALLY ALLOCATED MEMORY
   public:
      person();    // DEFAULT CONSTRUCTOR

      person(char*, char* =NULL,int=-1);
                   // A CONSTRUCTOR WHICH CAN TAKE 1,2 OR
                   // 3 ARGUMENTS, THE SECOND AND THIRD
                   // ARE AUTOMATICALLY SET TO THE DEFAULT
                   // IF NOT PROVIDED

      person(const person&); // THE COPY CONSTRUCTOR

      ~person();   // THE DESTRUCTOR

      person& operator=(const person&); // THE ASSIGNMENT OPERATOR

      void print();
};

person::person(): firstname(NULL), lastname(NULL), age(-1) {
   cout << ++lineno << " - DEFAULT CONSTRUCTOR" << endl;
}
```

```
person::person(char* lname,char* fname,int k) {
   copy(lname,fname,k);
   cout << ++lineno << " - SECOND CONSTRUCTOR" << endl;
}

person::person(const person& p) {
   copy(p.lastname,p.firstname,p.age);
   cout << ++lineno << " - COPY CONSTRUCTOR" << endl;
}

person::~person() {
   destroy();
   cout << ++lineno << " - DESTRUCTOR" << endl;
}

person& person::operator=(const person& p) {
   // DESTROY OLD AND THEN COPY NEW

   destroy();
   copy(p.lastname,p.firstname,p.age);
   cout << ++lineno << " - ASSIGNMENT OPERATOR" << endl;

   return *this;
}

void person::copy(char* lname,char* fname,int k) {

   age = k;

   if (lname!=NULL) {
      lastname = new char[strlen(lname)+1];
      strcpy(lastname,lname);
   } else lastname = NULL;

   if (fname!=NULL) {
      firstname = new char[strlen(fname)+1];
      strcpy(firstname,fname);
   } else firstname = NULL;
}
```

```
void person::destroy() {
   delete [] firstname;
   delete [] lastname;

   firstname = lastname = NULL;
}

void person::print() {
   cout << ++lineno << ' ' << (firstname==NULL?"unknown":firstname)
        << ' ' << (lastname==NULL?"unknown":lastname);
   if (age!=-1) cout << " (age: " << age << ')';
   cout << endl;
}

person foo(person p) {
   return p;
}



int main(void) {
   person people[2];
   people[0].print();
   people[1].print();
   cout << ++lineno << " array people declared" << endl;

   person john("DOE");
   cout << ++lineno << " john declared" << endl;

   person jane("SMITH","JANE",27);
   cout << ++lineno << " jane declared" << endl;

   people[0] = john;
   cout << ++lineno << " john assigned to p[0]" << endl;

   people[1] = foo(jane);
   cout << ++lineno << " jane assigned to p[1]" << endl;

   people[0].print();
   people[1].print();
   person *ptr;
   cout << ++lineno << " pointer to person is declared" << endl;

   ptr = new person;
   cout << ++lineno << " new person created dynamically" << endl;

   delete ptr;
   cout << ++lineno << " ptr destroyed" << endl;

   john = "KING";
   john.print();
   cout << ++lineno << " \"KING\" is assigned to john" << endl;

   cout << ++lineno << " end" << endl;

   return 0;
}
```

The output of the program is:

```
1 - DEFAULT CONSTRUCTOR
2 - DEFAULT CONSTRUCTOR
3 unknown unknown
4 unknown unknown
5 array people declared
6 - SECOND CONSTRUCTOR
7 john declared
8 - SECOND CONSTRUCTOR
9 jane declared
10 - ASSIGNMENT OPERATOR
11 john assigned to p[0]
12 - COPY CONSTRUCTOR
13 - COPY CONSTRUCTOR
14 - DESTRUCTOR
15 - ASSIGNMENT OPERATOR
16 - DESTRUCTOR
17 jane assigned to p[1]
18 unknown DOE
```

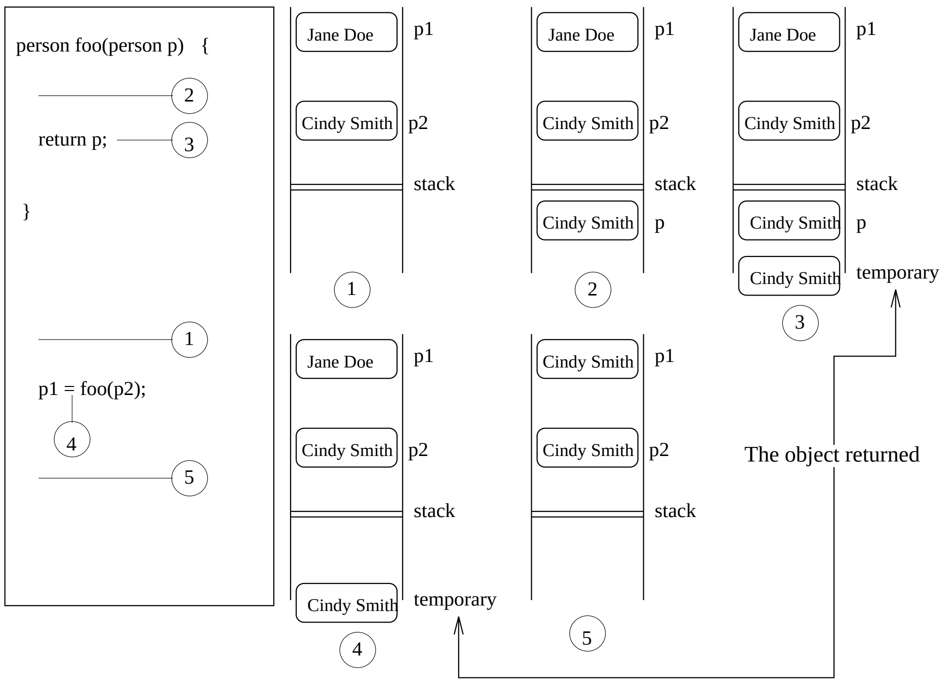
```
19 JANE SMITH (age: 27)
20 pointer to person is declared
21 - DEFAULT CONSTRUCTOR
22 new person created dynamically
23 - DESTRUCTOR
24 ptr destroyed
25 - SECOND CONSTRUCTOR
26 - ASSIGNMENT OPERATOR
27 - DESTRUCTOR
28 unknown KING
29 "KING" is assigned to john
30 end
31 - DESTRUCTOR
32 - DESTRUCTOR
33 - DESTRUCTOR
34 - DESTRUCTOR
```

Class `person` has three instance variables: `firstname`, `lastname` and `age`. These are private variables which can only be set by the constructors (*in our little program*). The *default constructor* assigns *NULL* to the character strings and *-1* to the integer variable. The instance variables are initialized using a special syntax only available for constructors: the chain of instance variables following the `:` (*colon*) are initialized. This chain is called the *initializer* and it must be used to initialize constant members, member classes and reference members. If a class is a subclass then the super class **must** be initialized using the initializer if a non-default constructor is to be called on the super class. In that case, it is called *explicit base initialization*. There is another constructor which can take 1, 2 or 3 arguments, by providing default values for the last 2 parameters. The *assignment operator* and the *copy constructor* only call the private method `copy`. This method creates a new dynamically allocated copy of first name and last name. The alternative is to simply set `firstname = p.firstname`. In this case, however, there would be two objects which share the same memory location for first names. This situation is called *aliasing*; two or more pointers point to the same memory location and the contents can be changed via either one of them. This may be undesirable in some cases. For this reason, `copy` creates a *deep copy*. A *deep copy* of an object is one which is logically identical to another one and they share no memory. A *shallow copy*, on the other hand, is not only identical to another object but they also share some physical memory. The latter case may be tricky because it must be handled which object is responsible for deallocating shared objects. The *default-* and *copy-constructors* and the *assignment operator* automatically provided by the compiler literally copy the contents of the variables. This results in *aliasing* if there are pointer instance variables. In general, there is no need to declare *copy-constructors*, the *assignment operator* and a *destructor* for a class which has *no* pointer instance variables. On the other hand, they must be declared if there are pointer variables, and there purpose is not specifically *aliasing*. The *assignment operator* is referred to by `operator=` and like the copy constructor, it takes a reference to an object of the same class. The value returned by the assignment operator is a reference to itself. The pointer `this` is defined for every class and it holds the address of the object itself. For example, the instance variable `firstname` inside the implementation of the methods of class person can be explicitly referred to as `this->firstname`. The destructor deallocates the dynamic memory chunks assigned to the pointers `firstname` and `lastname`, using private function `destroy`. If operator `delete` is applied to *NULL*, it does nothing. The function `foo` takes an object of class *person* by value and returns an object of class *person* by value. This function demonstrates how and when the *copy constructor*, the *assignment operator* and the *destructor* are involved in passing and returning objects by value.

If there is no default constructor declared for the class but there is a constructor which takes parameters then it is not possible to allocate an array of such instances. The reason is that when an array is declared, the default constructor is called on each and every element. If there is no public default constructor then no such array can be created. The compiler only provides a default constructor if there is no other constructor declared.

```
class A {
   public:
      A(int);
      ...
      // No default constructor A()
};

...

A array[10]; // NO! A needs a default constructor
             // to initialize each element!
```

Back to the example, first, an array for two person objects is declared. Because it is a statically declared array, each person object is implicitly initialized by the *default constructor*. This would also be the case if we had a dynamically allocated array. The `new`[] operator also causes the default constructor to be invoked on each slot. `joe` and `jane` are declared using the second constructor which takes parameters. Then we assign `joe` to the first slot of the array. The second slot is assigned via function `foo` (*lines 12 – 16*). First the *copy constructor* is called to create a copy of `p` on the stack (*line 12*). The *return* statement creates a temporary object of the local copy to be returned (*line 13*), this object will be picked up by the *assignment operator*. The local copies are then destroyed using the *destructor* (*line 14*). The *assignment operator* uses the temporary object, which is destroyed when the assignment is complete (*line 16*). The following memory trace shows the steps involved.



The temporary object (*second copy*) on the stack is created by the *return statement*. When the function terminates, this will be the only object left on the stack to be picked up. The scope of this anonymous temporary variable is the *statement* `foo` is embedded in.

The assignment of the string "KING" also creates a temporary copy. The *assignment statement* expects a `person` object as its argument. Because there is a constructor which can create a `person` object from a character string, it is invoked implicitly. Whenever a string (`char*`) is used in a context where a `person` instance is expected, the `person::person(char*,char*=NULL,int=-1)` constructor is called to create a *temporary* and *anonymous* instance.

Passing objects by reference increases performance and C++ allows to declare a reference parameter *constant*. When a reference parameter is constant, the compiler checks whether the parameter is not modified in any way in the function. The case when the parameter is on the left hand side of an assignment statement is trivial. However, it is more difficult (*in fact, in general impossible*) to detect at compile time whether the parameter passed to another function will modify it. A member function can (*and should be*) declared *constant* if it does not modify the instance variables. For example the *length* method of a *list* object should be *const*:

```
  class list {
     private:
        ...
        int size;
     public:
         ...
         int length() const;
  };

 ...
 int list::length() const {
    return size;
 }
```

There are cases when some parts of the objects cannot be *const* even when the object is *logically const*. For example, an object may keep a count of how many other objects are pointing to it for memory management purposes. Those instance variables which can never be *const* should be declared *mutable* using the `mutable` keyword[2].

To be fair, there is justification for such complicated mechanisms to initialize and destroy objects. It is true that in Java and SmallTalk there are no implicit calls to destructors and constructors. The constructors are only called by the `new` operator (*or class method in* SmallTalk) and by the constructors of subclasses to initialize the base class. However, Java and SmallTalk do not have statically or compile-time allocated instances. Also, in Java and SmallTalk , instances cannot be passed by value, because variables are really only the *descriptor*s[3]. Just like with primitive types, passing by value creates a copy. For objects, this copy is created by the constructor. Similarly, if an object is returned by value a copy must be returned, as local variables will be auto destroyed with the stack frame. This should warn programmers to pass and return by reference or pass the address when possible. However, sometimes it is not an option. Often you need to create a new object by a function and you cannot return the address or a reference to a local variable. Many times, specially with recursive methods, the method should create a new object, otherwise it would modify the original. In such a situation the object should be passed by value. The implicit mechanism which automatically creates an instance if the value passed matches the argument for a constructor was condemned by the object oriented community and it was excluded from Java . This, however is part of C++ , and programmers must be aware of it.

[2]`mutable` is only available in ANSI C++

[3]a variable descriptor is like a pointer, which besides the location of the actual object contains type information, possibly reference counts and pointers to virtual method lookup table(s)

```
class A {
   ...
   public:
     A();
     A(char*);
     A(double);

};

foo1(A) {
  ...
}

foo2(const A&) {
  ...
}



...
char * s = "Santa";
double d = 3.14;

foo1(s);          // A(char*) is called!
foo1(d);          // A(double) is called!

foo2(s);          // A(char*) is called!
foo2(d);          // A(double) is called!

A a1 = "Scrooge", a2 = -0.2; // Again!
```

4.4 Access Modifiers and Inheritance

Inheritance is a powerful mechanism to aid elegant software design and code reuse. Through inheritance instance and class variables and methods can be reused and implementation can be *extended* as opposed to *recoded*. This however assumes that the design observed and exploited areas that are applicable in more than one contexts. The issue of reusable object model design is addressed in detail in the second part of the lecture notes. This section is concerned with the syntax and semantics of inheritance in C++ .

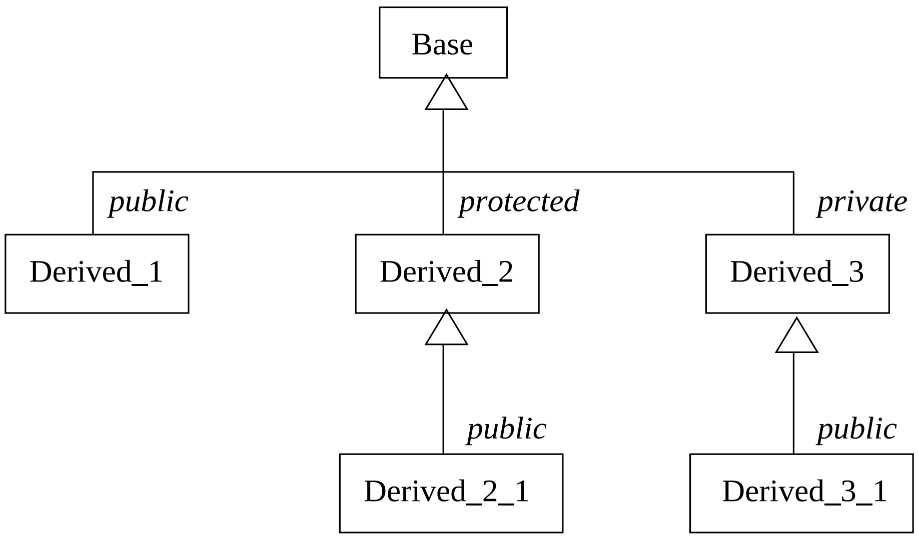
The class embodies a logically and structurally coherent part of a software application. Instance variables and methods can be hidden if they are declared with the keyword `private` and the access is granted by declaring the interface `public`. The `protected` keyword is yet another access modifier which gives the class components the same access privileges as `private`, however, they are visible in the *derived class*. All instance variables and methods inherit but the ones declared *private* in the base class are not visible in the derived class. In addition, C++ provides access modifiers for the inheritance itself, which may restrict access privileges in derived classes. If the *inheritance* is *private* then all otherwise visible members in the derived class are *private*. If the *inheritance* is *protected* then *public* members of the base class become *protected* and when the *inheritance* is *public* then visible members keep their access privileges, as declared in the base class. By default, inheritance is *private*.

Visibility			
member access	**class**	**derived class**	**outside class**
private	*YES*	*NO*	*NO*
protected	*YES*	*YES*	*NO*
public	*YES*	*YES*	*YES*

Private inheritance			
member access	**class**	**derived class**	**derived derived class**
private	*private*	*not visible*	*not visible*
protected	*protected*	*private*	*not visible*
public	*public*	*private*	*not visible*
Protected inheritance			
member access	**class**	**derived class**	**derived derived class**
private	*private*	*not visible*	*not visible*
protected	*protected*	*protected*	*protected or private**
public	*public*	*protected*	*protected or private**
Public inheritance			
member access	**class**	**derived class**	**derived derived class**
private	*private*	*not visible*	*not visible*
protected	*protected*	*protected*	*protected or private**
public	*public*	*public*	*public, protected or private**

* *it depends on the inheritance between derived and derived derived*

The following example illustrates the inheritance mechanism.



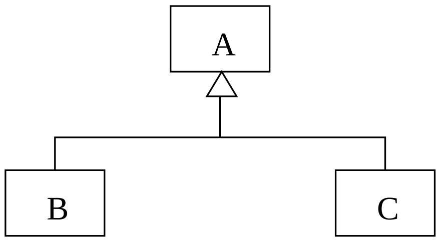
```
class Base {
  private:
    int a_priv;
  protected:
    int a_prot;
  public:
    int a_pub;

  protected:
    void set_a_priv(int i) { a_priv = i; }

  public:
    void set_a_prot(int i) { a_prot = i; }
    void set_a_pub(int i)  { a_pub  = i; }

    void foo() { }
};

class Derived_1 : public Base {
   void foo() {
      a_priv = 2;    // error, a_priv is not visible
      set_a_priv(2); // ok
      a_prot = 2;    // ok
      a_pub  = 2;    // ok
   }
};
```

```
class Derived_2 : protected Base {
   void foo() {
      a_priv = 2;    // error, a_priv is not visible
      set_a_priv(2); // ok
      a_prot = 2;    // ok
      a_pub  = 2;    // ok
   }
};

class Derived_3 : private Base {
   void foo() {
      a_priv = 2;    // error, a_priv is not visible
      set_a_priv(2); // ok
      a_prot = 2;    // ok
      a_pub  = 2;    // ok
   }
};

class Derived_2_1 : public Derived_2 {
   void foo() {
      a_priv = 2;    // error, a_priv is not visible
      set_a_priv(2); // ok
      a_prot = 2;    // ok
      a_pub  = 2;    // ok
   }
};

class Derived_3_1 : public Derived_3 {
   void foo() {
      a_priv = 2;    // a_priv is not visible
      set_a_priv(2); // set_a_priv is not visible
      a_prot = 2;    // set_a_prot is not visible
      a_pub  = 2;    // set_a_pub is not visible
   }
};

int main(void)
  {
   Derived_1 d1;
   Derived_2 d2;
  Derived_3 d3;

  d1.set_a_priv(2);  // error, set_a_priv is protected
  d1.set_a_prot(2);  // ok
  d1.set_a_pub(2);   // ok

  d2.set_a_priv(2);  // error, set_a_priv is protected
  d2.set_a_prot(2);  // error, set_a_prot became protected
  d2.set_a_pub(2);   // error, set_a_pub became protected

  d3.set_a_priv(2);  // error, set_a_priv became private
  d3.set_a_prot(2);  // error, set_a_prot became private
  d3.set_a_pub(2);   // error, set_a_pub became private

  return 0;
}
```

While all combinations of the *private*, *protected* and *public* access modifiers can be used for members and the inheritance itself, in general the following heuristic will suffice:

- Members (*variables and methods*) which are not part of the interface of the class should be *protected*.
- Instance variables, in most cases, should not be *public*.
- Members which specifically belong to *this* class and would create confusion for the programmer (*possibly another person*) who creates a derived class should be *private* provided with a *protected* interface. This set up should also be used to *hide* complicated mechanisms.
- The *inheritance* itself, in most cases, only makes sense to be *public*.
- The interface to the class must be *public*.

Data members, hence are always present in the derived classes but they may have changed access privileges. This raises the question, how do the constructors initialize the inherited variables and how are they released by the destructor? The program below defines three classes. `A` is the base class of class `B` and class `C`. While both `B` and `C` are equal in logic and functionality, class `B` illustrates common problems and class `C` demonstrates the proper implementation.



```
#include <iostream.h>

int lineno = 0;

class A {
  protected:
     int a;
     int b;
  public:
     A():a(1),b(2) {
       cout << ++lineno << ' ' << "A::A()" << endl;
     }

     A(int i, int j):a(i),b(j) {
       cout << ++lineno << ' ' << "A::A(int,int)" << endl;
     }

     A& operator=(const A& c) {
       a = c.a;
       b = c.b;
       cout << ++lineno << ' ' << "A::operator=(const A&)" << endl;
       return *this;
     }

     virtual ~A() {
       cout << ++lineno << ' ' << "A::~A()" << endl;
     }
};

class B : public A {
  protected:
     int c;
  public:
      B() {
        cout << ++lineno << ' ' << "B::B()" << endl;
      }

      B(int i, int j, int k) {
        a = i;  // this should not be done this way
        b = j;
        c = k;
        cout << ++lineno << ' ' << "B::B(int,int,int)" << endl;
      }

      B& operator=(const B& d) {
        a = d.a; // this can be done simpler
        b = d.b;
        c = d.c;
        cout << ++lineno << ' ' << "B::operator=" << endl;
        return *this;
      }

      ~B() {
        cout << ++lineno << ' ' << "B::~B()" << endl;
      }
};

class C : public A {
  protected:
     int c;
  public:
      C() {
        cout << ++lineno << ' ' << "C::C()" << endl;
      }

      C(int i, int j, int k):A(i,j),c(k) { // proper base initialization
        cout << ++lineno << ' ' << "C::C(int,int,int)" << endl;
      }

      C& operator=(const C& d) {
        A::operator=(d);
        c = d.c;
        cout << ++lineno << ' ' << "C::operator=" << endl;
        return *this;
      }

      ~C() {
        cout << ++lineno << ' ' << "C::~C()" << endl;
      }
};

int main(void) {

  A a;
  cout << ++lineno << ' ' << "a is declared" << endl;

  B b1;
  cout << ++lineno << ' ' << "b1 is declared" << endl;
  B b2(3,4,5);
  cout << ++lineno << ' ' << "b2 is declared" << endl;
  b1 = b2;
  cout << ++lineno << ' ' << "b1 = b2" << endl;

  C c1;
  cout << ++lineno << ' ' << "c1 is declared" << endl;
  C c2(4,5,6);
  cout << ++lineno << ' ' << "c2 is declared" << endl;
  c1 = c2;
  cout << ++lineno << ' ' << "c1 = c2" << endl;

  return 0;
}
```

The output of the above program is:

```
1 A::A()
2 a is declared
3 A::A()
4 B::B()
5 b1 is declared
6 A::A()
7 B::B(int,int,int)
8 b2 is declared
9 B::operator=
10 b1 = b2
```

```
11 A::A()
12 C::C()
13 c1 is declared
14 A::A(int,int)
15 C::C(int,int,int)
16 c2 is declared
17 A::operator=(const A&)
18 C::operator=
19 c1 = c2
20 C::~C()
21 A::~A()
22 C::~C()
23 A::~A()
24 B::~B()
25 A::~A()
26 B::~B()
27 A::~A()
28 A::~A()
```

When an instance of class `B` or class `C` is declared, the default constructor of the base class is automatically called unless another constructor of the base class was specified to initialize the base. This is demonstrated by the three integer argument constructor. When an instance of class `B` is declared with three arguments (*lines 6-7*), the default constructor is executed by default. However, both the default constructor of `A` and the three integer argument constructor of `B` initialize the instance variables `a` and `b`. If they were not of a primitive type, but instances of classes with complex structure, this double initialization would be unacceptable. Class `C` demonstrates the proper way of initialization using an existing constructor of a base class (*lines 14-15*). `C::C(int,int,int)` also shows the proper way of initializing the new instance variable `c` without creating a temporary copy of the argument `k` on the method's stack. Unlike the constructors, the assignment operators *do not* call the assignment operator of the base class. However, the assignment operator can be explicitly called using the `::` scope resolution operator. If the base class has private instance variables the only way to have them assigned to is by invoking the base's assignment operator this way! The destructor, like the constructor, implicitly calls the destructor of the base class. Hence if the base class has a pointer variable which points to a dynamically allocated memory chunk and its destructor deallocates it, then this instance variable *should not* be explicitly deallocated in the base class. If the pointer instance variable is set to *NULL* after the `delete` operator was called, then multiple explicit deallocations of this instance variable in the derived classes only result in a redundant call. If the pointer is not set to *NULL* and the `delete` operator is called on this pointer variable in the derived classes, then this may result in memory fault. The `delete` operator, when applied to an instance of a class, also (*implicitly*) executes the destructor method of the class.

4.5 The Initializer

The initializer is not just some special syntax which could be avoided and its behavior mimicked in other ways. The initializer is irreplaceable in the following two contexts:

- a non-default constructor is required to initialize the base
- an instance variable is to be initialized by a non-default constructor

As we know, constructors of the derived class first implicitly call the default constructor of the base class unless another constructor is explicitly specified instead of the default constructor. Similarly, the destructor of the derived class implicitly calls the destructor of the base class after it has executed its code. The only way to specify a non-default constructor is to use the initializer. If the base has private instance variables, then this is the only way to provide them with default values and specifying a non-default constructor is often necessary to avoid double (*or multiple*) initialization as demonstrated by the example below. Contrary to popular belief, it is not only redundant but could potentially be very dangerous to initialize certain types of variables twice. For example – as usual – pointer instance variables initialized twice by addresses returned by `new` create memory leaks. Similarly, resource objects – such as streams – often cannot be initialized more than once without very serious and unwanted side effects (*like opening a file twice!*). A memory leak is demonstrated in the next example.

```
class Base {
   protected:
      char* name;  // POINTER VARIABLE
   public:
      Base() { // Default constructor
         name = new char[strlen("hello")+1];
         strcpy(name,"hello");
      }

      Base(char* s) { // Non-Default constructor
         name = new char[strlen(s)+1];
         strcpy(name,s);
      }
      ...
};
```

```
class Derived {
   public:
      Derived() { // Default constructor, INCORRECT
         name = new char[strlen("bye")+1];
         strcpy(name,"bye");
      }

      Derived():Base("bye") { // Default constructor, CORRECT
      }
      ...
};
```

The incorrect default constructor of `Derived` innocently assigns `"bye"` to `name`, but as we know the default constructor of `Base` is called first by the constructors of `Derived` unless overridden explicitly. Hence first `"hello"` is assigned to `name` and then `"bye"` loosing the memory needed to store `"hello"` forever. If `name` were `private` as opposed to `protected` then assigning anything to it explicitly would not even be possible because private instance variables are not visible (*but still there!*) in derived classes.

Besides providing a mechanism to bypass the constructor of the base by invoking a different constructor, the initializer is also the only mechanism that can be used to invoke a non default constructor on instance variables which are not pointers.

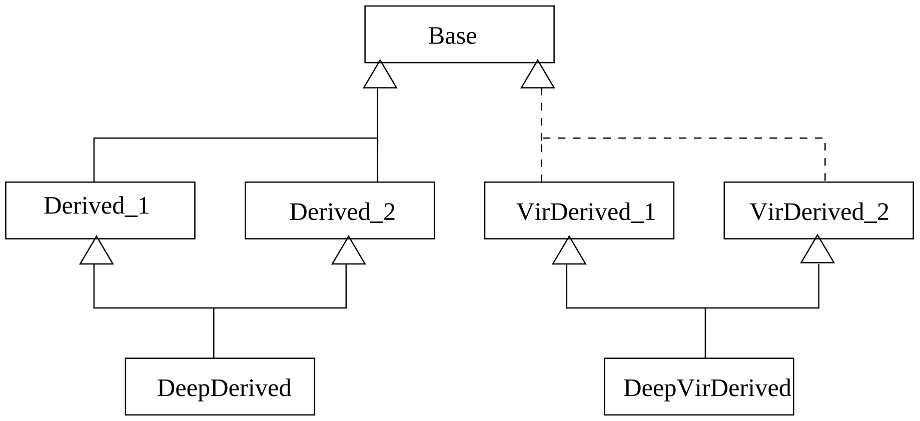
```
class A {
   public:
      A() { ... }
      A(int) { ... }
};

class B {
   private:
      A a;
   public:
      B(int i):a(i) { // initialize a with A(int) as opposed to A()
      }
};
```

4.6 Multiple Inheritance

Through multiple inheritance, a class can be derived from more than one base classes. The benefit of multiple inheritance is much debated in the object oriented community and because of the extra complexity it introduces, most object oriented programming languages (Java , SmallTalk) do not allow it. Often an object logically belongs to more than one hierarchy, because it plays a different *role* in each (*like rusty the cat's role as a feline in evolution and his as role of a pet).* In Java there is language support to implement *role hierarchies* without multiple inheritance using *interfaces*. Some versions of C++ , like gcc 2.7.2 *signatures*, provide non-standard mechanisms to implement multiple roles of an object without relying on multiple inheritance. We will address this issue in the second part of the lecture notes in more detail. Here we are only concerned with the syntax and semantics of the multiple inheritance mechanism of C++ . The problems that naturally arise are: if both base classes have members with the same name which one is present in the derived class and if deeper in the hierarchy two classes which are subclasses of the same class rejoined, are there more than one copies of the base class? The following example addresses both of these issues.

```
#include <iostream.h>

int lineno = 0;

class Base {
   protected:
     int a;
   public:
     Base():a(0) {
       cout << ++lineno << ' ' << "Base::Base()" << endl;
     }
     void set_a(int i) { a = i ; }
     int  get_a() { return a; }
  };

class Derived_1 : public Base {
   protected:
     int b;
   public:
     Derived_1():b(0) {
       cout << ++lineno << ' ' << "Derived_1::Derived_1()" << endl;
     }
     void set_b(int i) { b = i + 1; }
     int get_b() { return b; }
  };

class Derived_2 : public Base {
   protected:
     int b;
   public:
     Derived_2():b(0) {
       cout << ++lineno << ' ' << "Derived_2::Derived_2()" << endl;
     }
     void set_b(int i) { b = i - 1; }
     int get_b() { return b; }
  };

class VirDerived_1 : virtual public Base {
   protected:
     int b;
   public:
     VirDerived_1():b(0) {
       cout << ++lineno << ' ' << "Derived_1::Derived_1()" << endl;
     }
     void set_b(int i) { b = i + 1; }
     int get_b() { return b; }
  };

class VirDerived_2 : virtual public Base {
   protected:
     int b;
   public:
     VirDerived_2():b(0) {
       cout << ++lineno << ' ' << "Derived_2::Derived_2()" << endl;
     }
     void set_b(int i) { b = i - 1; }
     int get_b() { return b; }
  };

class DeepDerived : Derived_1, public Derived_2 {
  public:

     void set_a(int i) {
        Derived_1::set_a(i);
     }

     void set_b(int i) {
        Derived_2::set_b(i);
     }
     int get_a() {
        return Derived_2::a;
     }

     int get_b() {
        return Derived_1::get_b();
     }
  };


class DeepVirDerived : public VirDerived_1, public VirDerived_2 {
   public:

     void set_b(int i) {
        VirDerived_2::b = i;
     }


     int get_b() {
        return VirDerived_1::b;
     }
  };

int main(void) {

  DeepDerived dd;
  cout << ++lineno << ' ' << "dd is declared" << endl;

  DeepVirDerived dvd;
  cout << ++lineno << ' ' << "dvd is declared" << endl;

  dd.set_a(3);
  dd.set_b(3);
  cout << ++lineno << ' ' << "dd.a = " << dd.get_a()
       << " dd.b = " << dd.get_b() << endl;

  dvd.set_a(3);
  dvd.set_b(3);
  cout << ++lineno << ' ' << "dvd.a = " << dvd.get_a()
       << " dvd.b = " << dvd.get_b() << endl;
  return 0;
}
```

The following diagram is the inheritance hierarchy. The dashed lines indicate that the inheritance is *virtual*. The keyword `virtual` maybe added to a base class specifier in the definition of the derived class. The members of the base class will be shared, *as opposed to duplicated* by every class that specified the base class to be *virtual*. Hence `DeepVirDerived` has only one instance of the members of `Base`; in this case specifically, instances of `DeepVirDerived` have only one `a`. On the other hand, instances of `DeepDerived` have multiple instances of `a`. Both `DeepDerived` and `DeepVirDerived` have two instances of `b` (*and methods set_b and get_b*) and this cannot be prevented.



The output of the program is:

```
1 Base::Base()
2 Derived_1::Derived_1()
3 Base::Base()
4 Derived_2::Derived_2()
5 dd is declared
6 Base::Base()
7 Derived_1::Derived_1()
8 Derived_2::Derived_2()
9 dvd is declared
10 dd.a = 0 dd.b = 0
11 dvd.a = 3 dvd.b = 0
```

Because `dd` has two instances of `a`, the constructor on each instance is called (*lines 1-4*), while there is only one constructor `Base::Base()` needed for `dvd` because it has only one copy of the members from `Base`. Both, however have multiple copies of `b`, `set_b` and `get_b`. This ambiguity must be resolved, hence `get_b` and `set_b` must be reimplemented to specify which version to be used. Both examples show a faulty resolution and hence demonstrate the existence of the two copies of `b`. The ambiguity also exists for `a`, `set_a` and `get_a` in class `DeepDerived`, but this ambiguity is automatically resolved in class `DeepVirDerived` by *virtual inheritance*. This simple example demonstrates the complications arising from the use of multiple inheritance. In the second part of the lecture notes we look at techniques to avoid it and still giving objects multiple logical roles.

4.7 Notes on Syntax and Semantics

The `C` `struct` and `C++` `class` are very different. The fact is that many `C++` applications today use old `C` libraries with `C` `struct`s. `C` `struct`s have only data fields (*which may be function pointers*) but no member functions. To declare a variable of the `C` record type, one must use the `struct` keyword.

```
struct S {
   int a;
   double d;
};


struct S f(int i, double e) {
   struct S s;
   s.a = i;
   s.d = e;

   return s;
}
```

If `S` were a `C++` `struct` instead – *which declares a class with all its fields public* – then `f` would be defined as:

```
S f(int i,double e) {
  S s;
  s.a = i;
  s.d = e;

  return s;
}
```

The way `C` and `C++` structs are stored is different as well. As `C++` structs are classes as well the descriptors store extra information, such as pointers to the virtual function table, run-time type information, *etc.* Because the types are incompatible, such code compiled by a C compiler has to be linked in a different way. As type definitions are usually placed in header files, when you include C headers place them into an external linkage block. Suppose the C typedefs with the C structs and function prototypes are in a header file "defs.h" and the implementation was compiled by a C compiler. Then to include and use these functions and types in a C++ application place the definitions into the block:

```
extern "C" {
  #include "defs.h"
}
```

This will instruct the compiler that the prototypes and structs in "defs.h" are C style and they were built by a C compiler.
Often beginner programmers are confused by the use of the scope resolution operator – ::. This operator is used to designate which class the method or member belongs to. It is very often the case that more classes have a method with the same name. When the method is defined it must be known which class it belongs to:

```
class A {
   public:
      void foo();
      ...
};

class B {
   public:
      void foo();
      ...
};

void A::foo() {
  // A's foo
  ...
}

void B::foo() {
  // B's foo
  ...
}
```

The scope resolution operator is also used to access a method in a super class.

```
class A {
   public:
      void foo();
      ...
};

class B : public A {
   public:
      void foo();
      ...
};

void B::foo() {
   A::foo(); // call A's foo
}
```

If we did not explicitly specify `A::foo()` then a simple `foo()` would be a recursive call. It is indeed often the case that the method which reimplements the inherited one needs to use the original implementation. A classical example is the assignment operator. If the assignment operator is overloaded, it is almost certain that it should call the assignment operator of the super class – *which is not done by default.*
The `this` pointer is accessible in instance methods and it stores the address of the object which invoked the method. The assignment operator returns a reference to itself (*the left hand side of the assignment*) but how would it know what "itself" is?

```
A& A::operator=(const A&) {
   ...
   return *this;
}
```

```
  ...

  A a1,a2,a3;

  a1 = a2 = a3;
```

Here the order of evaluation is `a1 = (a2 = a3)`. In other words first `a2 = a3` is evaluated and its result sets `a1`. To be able to this chain of assignments the assignment itself must have a value. The value of course is the left hand side which is only accessible via the `this` pointer.

Constructors and destructors are also the source of a lot of confusion for beginners. The root of the problem is that in C++ not all objects are dynamically allocated. In Java and SmallTalk objects must explicitly be created by the `new` operator. C++ does not have a garbage collector which means that memory deallocation is the programmer's responsibility. The constructor is implicitly invoked when an instance is declared and by the `new` and `new[]` operators. The destructor is implicitly invoked when the instance leaves scope and by the `delete` and `delete[]` operators. When an object is passed by value, a copy of it must be created on the stack and it is achieved by the *copy constructor*. Similarly when a method or function returns an instance by value, a copy created by the *copy constructor* is left on the stack to be picked up by the caller. Not being familiar with this mechanism can lead to very hard to detect problems. First the programmer must understand what a "copy" of the object should look like. In particular, if it has pointer instance variables which hold the addresses of dynamically allocated objects then should a copy of the object share this dynamically allocated memory chunk or should it create its own copy? There is no definite answer and either way is just as common in practice. The former way – *sharing memory* – is much more complicated to handle. For example, if the destructor calls `delete` on this variable it may delete it for many other instances as well. Once it is understood how a copy should look like, an appropriate copy constructor and destructor must be designed. In case of memory sharing, it may be necessary to use instance counters to keep track of how many pointers are pointing to the shared piece of memory.

```
class A {
   private:
      int *a;
   public:
      A(const A&) {
       a = new int;
       *a = A.a;
      }
      ...
}

class B {
   private:
      int *b;
   public:
      ~B() {
        delete b;
      }
};
```

In the above examples instances of class `A` and class `B` have a dynamically allocated instance variable, but `A` has a copy constructor and no destructor and `B` has a destructor but no copy constructor. If an object of type `A` is passed by value, then a copy of it is created on the stack which will be destroyed by the destructor. As there is no destructor declared, the default action is "nothing", hence every time an instance of class `A` leaves its scope or passed by or returned by value there is a memory leak. When instances of class `B` are passed by or returned by value a temporary copy is created on the stack by the *copy constructor*. As it does not exist the copy created will share the instance variable `a`. When the function is popped from the stack, the destructor gets called which will delete `a` for the original object as well because of memory sharing. In general it is important to understand that if there are dynamically allocated instance variables or there is aliasing the constructors, the destructor and the assignment operator must be carefully designed. If there is a need for a destructor, the chances are you also need an appropriate copy constructor and assignment operator. Sometimes it is unacceptable to pass instances of a particular class by value because they are so complicated or big that the copy created would consume too much memory or it would take too much time to create it. In that case, it is advisable to declare a copy constructor and an assignment operator and make them *private*. This guarantees that the code will not compile if instances of the class are attempted to be passed or returned by value.

It is also very important to understand how the constructor-destructor mechanism works together with inheritance. It is the rule in every object oriented language that the constructor of the base class implicitly calls the constructor of its super class before its executes its own code. The constructor called is the *default constructor* unless it is explicitly specified otherwise. The reason for this implicit call is that instance variables declared in the super class must also be initialized. To avoid multiple initialization, non-default constructors of the base class should propagate parameters by invoking the appropriate constructor of the base class. This can only be done by the initializer. The initializer syntactically is a colon followed by a comma separated list after the parameters and before the {. The elements of the initializer are either instance variable settings or a call to a constructor of the base class (*which overrides the implicit call to the default constructor of the super class*). If there is an instance variable whose class does not have a default constructor then it can only be initialized by the initializer.

4.8 Exercises

4.1 Give an example where it is undesirable to create a logical *or deep* copy of an object by the *copy constructor* and/or the *assignment operator*! Give a small example (*as a* `C++` *code fragment*) of an appropriate destructor.

4.2 Give an example where you would use a *class* declared by the `union` keyword. *Hint: such a class would probably be an instance variable of a "wrapper" class!*

4.3 Give a concrete C++ code example with two classes – `Base` and `Derived` – and a resource `foo`, where not explicitly calling a non-default constructor of the base class causes a serious problem *such as memory leak, loss of a resource or irrecoverable inconsistency.*

```
class Base {
 ...
 public:
  Base() {
  ...
  }
  Base(foo f) { // non-default constructor
  ...
  }
};
class Derived : public Base{
 ...
 public:
  Derived(foo f) { ... } // causes a serious problem
  Derived(foo f):Base(f) { ... } // ok
};
```

Explain!

4.4 Implement a class `smart_object` which keeps a count how many other objects are pointing to it. Also implement a class `smart_array` which has an array of pointers to `smart_objects`. When the destructor deallocates an instance of `smart_array` it deallocates the object it holds if and only if it has a reference count of 1. In other words, the array slot has the last pointer pointing to the object. Design and implement a very tiny application which demonstrates `smart_array` and `smart_pointer`.

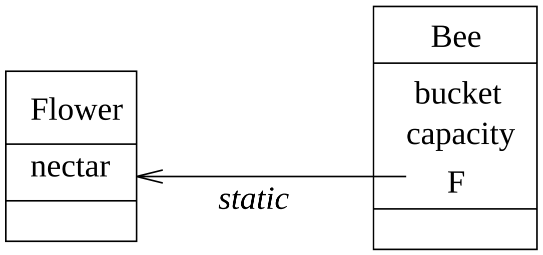
Chapter 5

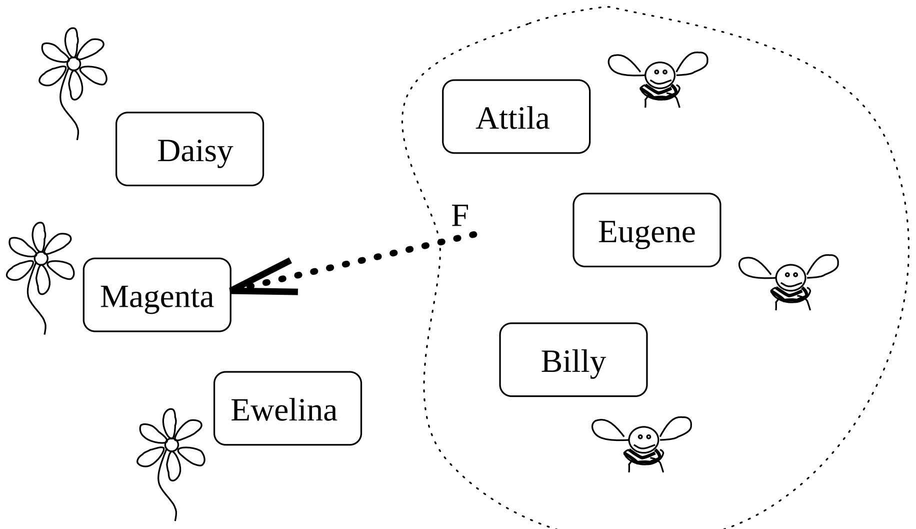
Classes II.

5.1 Class Variables and Methods

Instance variables belong to the specific instance, however it may make sense to share variables and methods across all instances of the class. For example a class variable may keep track of how many objects of the class are allocated at any point when the application is running. Class variables and class methods belong to the class, not to the instance, hence there is only one physical manifestation of a class variable and all instances share this common copy. The lifetime of a class variable is the entire program and its scope is all instances of the class. Class variables are physically allocated in the program text. Similarly, class methods belong to the class and they can be invoked – *if public* – even if no instances are currently allocated via the class name and the scope resolution operator. Class methods usually maintain and query the values of class variables and they perform tasks which logically belong to the class but there is no need for an actual instance to accomplish the job. The `static` keyword declares a variable or a method in the class definition to belong to the class and not to the instance. Class variables and methods are very useful to implement and maintain defaults[1]. The following is an example of sharing variables between all instances of the class using *static* variables and methods.

[1]memory allocation of new instances of the class in SmallTalk is always a class method, that is, the request `new` is sent to the class to ask for an instance. C++ , however, does not follow this elegant method of creating instances, because the `new` operator is global which may be overridden for a specific class





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

class Flower {
  protected:
    int nectar;

  public:
    Flower(int i=5):nectar(i) {
    }

    int give_nectar() {
       if (nectar>0)  {
         nectar--;
         return 1;
       }
       return 0;
    }
    int empty() {
       return nectar == 0;
    }

    void assess_capacity(char* name) {
      cout << name << " has " << nectar << " amount of "
           << "nectar left" << endl;
    }
};

class Bee {
   protected:
     static Flower* F;
     int bucket;
     int capacity;

   public:
     Bee(int i=5):bucket(0),capacity(i) {
     }

     void collect() {
        if ( bucket < capacity) {
          bucket += F->give_nectar();
        }
     }

     int full() {
       return bucket == capacity;
     }


     static void Send_to_flower(Flower *f) {
       F = f;
     }

     void assess_performance(const char * name) {
        cout << name << " collected " << bucket << " amount of nectar "
             << "and his bucket is " << (full()?"full":"not full") << endl;
     }
};
```

```
Flower* Bee::F = NULL;

int main(void) {

   Flower daisy(3);
   Flower ewelina(4);
   Flower magenta;

   Flower *field[] = { &daisy, &ewelina, &magenta };
   char* flower_name[] = { "daisy","ewelina","magenta" };

   Bee attila(4), joel(6), eugene;

   int flower_index = 0;

   Bee::Send_to_flower(field[flower_index]);  // setting the static variable

   while ( (!attila.full() || !joel.full() || !eugene.full() )
                         &&
           (!daisy.empty() || !ewelina.empty() || !magenta.empty())) {

     if (field[flower_index]->empty()  && flower_index < 2) {
        flower_index++;
        Bee::Send_to_flower(field[flower_index]);
     }

     switch(random()%3)  {
       case 0: attila.collect();
               cout << "attila <- ";
               break;
       case 1: joel.collect();
               cout << "joel <- ";
               break;
       case 2: eugene.collect();
               cout << "eugene <- ";
               break;
     }

    cout << flower_name[flower_index] << endl;
   }
   attila.assess_performance("attila");
   joel.assess_performance("joel");
   eugene.assess_performance("eugene");

   daisy.assess_capacity("daisy");
   ewelina.assess_capacity("ewelina");
   magenta.assess_capacity("magenta");

   return 0;
}
```

A typical run is:

```
attila <- daisy
attila <- daisy
joel <- daisy
eugene <- ewelina
joel <- ewelina
attila <- ewelina
attila <- ewelina
joel <- magenta
joel <- magenta
joel <- magenta
attila <- magenta
eugene <- magenta
eugene <- magenta
attila collected 4 amount of nectar and his bucket is full
joel collected 5 amount of nectar and his bucket is not full
eugene collected 3 amount of nectar and his bucket is not full
daisy has 0 amount of nectar left
ewelina has 0 amount of nectar left
magenta has 0 amount of nectar left
```

And now the story: *attila*, *joel* and *eugene* are three friends who regularly fly out to collect nectar from their flower friends *daisy*, *ewelina* and *magenta*. In fact, they are so much inseparable that they always collect from each flower together until it has no nectar left. An average bee is capable of carrying 5 amounts of nectar in his bucket, but *attila* is somewhat smaller, so he can only carry 4 amounts while *joel* can carry 6 amounts. The flowers produce 5 amounts of nectar a day, but this nice spring morning, *daisy* only produced 3 amounts and her sister *ewelina* produced 4 amounts[2]. The rest

[2]while *attila*, *joel* and *eugene* are males, in reality male bees are not involved in the work of collecting nectar or producing honey, their main function is reproduction

of the story is the program's output.

F is a static variable, which belongs to class Bee, as opposed to the instances. In fact there is only one F for the entire class. Send_to_flower is a static member function, which can be called without an instance. The program shows how a message through a static method can be used to change the behavior of all instances of the entire class, without explicitly notifying each instance. The static variables and methods are also available through the instances, so attila.Send_to_flower(...) is a valid call.

Class members and methods can also be used to designate a *namespace*. Before namespaces became standard in C++ , they could be simulated by public class methods. As an example:

```
  class Math {
     public:
        static double sin(double);
        static double log(double);
        ...
  };

  ...

  double d = Math::sin(3.14);

  ...
```

To perform these trigonometric functions, there is no need for an actual instance of Math, but they do logically belong to a common entity.

5.2 Friend Classes and Functions

C++ allows *free functions*, methods of another class or an entire other class to have access to private variables and methods of a class, if this class specifically grants this access privilege. A class can explicitly declare a function or a class *friend* with the friend keyword.

```
class C; // forward declaration, so Friendly sees C
class Friendly; // forward declaration, so B sees Friendly

class B {
   public:
     int see_Friendly(Friendly);
};
```

```
class Friendly {
   private:
      int a;
      char b;

   public:
      Friendly():a(1),b('a') {
      }

   friend class C;
   friend int B::see_Friendly(Friendly);
   friend void foo(Friendly);
};

class C {
   private:
     int c;
   public:
     void set_C(Friendly f) {
        c = f.a;
     }
};

int B::see_Friendly(Friendly f) {
   return f.a;
}

void foo(Friendly f) {
  int c = f.a;
};
```

The access granted by `friend` does not inherit: `class Mean: public Friendly` would not grant access to any of the functions privileged by `Friendly`.

5.3 `class` vs. `struct` vs. `union`

In C++ it is also possible to define a class with the `union` and `struct` keywords. If the `struct` keyword is used to define a class then by default all members are public, as opposed to a class defined by the keyword *class* whose members are private by default. If the access modifiers are explicitly specified for each member, the class declared by *struct* is the same as if it were declared by *class*. On the other hand, a class defined using the `union` keyword is different from classes defined by the `class` and `struct` keywords. Like the C *union*, an instance of such a class shares the address space for the data members!

```
#include <iostream.h>

union U {
  private:
    char string[sizeof(int)];
    int integer;

  public:
    U(int i=0):integer(i) {
    }

    int get_integer() {
      return integer;
    }

    char* get_string() {
      return string;
    }
};

int main(void) {
  U u(123456);

  cout << "u.integer = " << u.get_integer() << endl;
  cout << "u.string = ";

  for(int i=0;i<sizeof(int);i++)
     cout << '[' << (int)(unsigned char)u.get_string()[i] << ']';
  cout << endl;

  return 0;
}
```

The output of the program on my machine:

```
u.integer = 123456
u.string = [0][1][226][64]
```

5.4 Inline Methods and Embedded Classes

C++ provides the *inline* mechanism to replace function calls with a substitution of their implementation. This results in faster execution speed. For example, if the length method of a list is defined *inline*, whenever the method is called, there is no function invocation. The call is appropriately replaced by the body.

```
class List {
private:
   int size;
   ...
public:
   inline int length();
   ...
};

inline int List::length() {
   return size;
}
```

The compiler may ignore the inline specifier and implement the method with a real function. By default, if the body of a function is given within the class definition, it is treated inline.

It is also possible to define a class within a class. In such a case, instances of the *embedded class* can only be created by the class, which defined the *embedded class*.

```
class Outer {
  private:

    class Inner {
       public:
           int a;
    };

    Inner I;

  public:
    int foo() {
      return I.a;
    }
};
```

Of course embedded classes can be nested in embedded classes as well. When a method of an embedded class is implemented in another file it must be explicitly defined which class(es) it belongs to – using the scope resolution operator `::`.

```
class Outer {

   protected:

      class Inner {
         protected:
            Inner();             // constructor
            ~Inner();            // destructor
            void foo(int) const; // a method
            ...
      };

   ...

};


// constructor
Outer::Inner::Inner() {
   ...
}

// destructor
Outer::Inner::~Inner() {
   ...
}

// the foo method
void Outer::Inner::foo(int a) const {
   ...
}
```

5.5 The Role of the Constructors and the Destructor

In general constructors initialize objects and the destructor releases resources associated with the instance. When an instance of a derived class is initialized, the compiler must make sure that members of the base class are initialized as well. However, it is a common misconception that this is achieved using the inheritance mechanism. The following OO axioms should clarify why and how implicit mechanisms are used to properly create, initialize or destroy instances.

Constructors do not inherit!

Instead the constructor of the base class first implicitly calls the default constructor of the parent class. It is possible to explicitly override the default constructor by another constructor of the base class, using the initializer.

```
class Base {
  private:
    int i;

  public:
    Base():i(-2) { }
    Base(int _i):i(_i) { }
};

class Derived : public Base {
  private:

    int j;

  public:
    Derived():j(-3) { }
           // Base() is implicitly called!!, i = -2

    Derived(int _j):j(_j) { }
           // Base() is implicitly called!!, i = -2

    Derived(int _i,int _j):Base(_i):j(_j) { }
           // Instead of Base() Base(int) is called
};
```

It is clear why both `Base` and `Derived` could not be initialized if constructors inherited: if `Derived()` could override `Base()`, then `Base()` would not be called; if `Derived()` is not implemented and constructors inherited, then the members introduced in `Derived` would not be initialized.

The destructor does not inherit!

The reasoning is analogous. If the destructor inherited, either members introduced in the base class or members introduced in the derived class would not be deallocated. The destructor of the derived class *first* executes its own code and then the destructor of the base class gets called implicitly.

The assignment operator inherits!

Operators implemented as instance methods are just like any other instance method: *they inherit.* This however is seldom what the programmer wants, because either members introduced in the base class or members introduced in the derived class will not be assigned. In other words, the assignment operator's implementation in the derived class should make an explicit call to the assignment operator of the base class.

```
class Base {
    ....
  public:
    Base& operator=(const Base& b) {
      ...
      return *this;
    }
};

class Derived: public Base {
     ...
  public:
     Derived& operator=(const Derived& d) {
        Base::operator=(d); // explicit call
        ...
        return *this;
     }
};
```

Implicit calls to the constructors and the destructor

- Whenever an instance or an array of instances of a class is statically declared, the default constructor is called: `Foo f; Foo af[10];` In the case of the array, the **default constructor** is called on each and every slot once. The destructor on these slots will be called in the exact opposite order.

- Whenever an instance or an array of instances is created dynamically by the `new` or `new[]` operator the **default constructor** is called: `Foo *f; f = new Foo;`

- All constructors of the derived class implicitly call the **default constructor** of the base class first, unless another constructor of the base class is specified explicitly (*in the initializer*).
- Whenever an instance is passed by *value*, the *copy constructor* is used to create a copy of the instance on the stack.
- Whenever an instance is returned by *value*, the *copy constructor* is used to create a copy of the instance returned.
- **If** there is a constructor of `Foo` which takes an instance of type `T` (`Foo::Foo( T t)`), then whenever a `T` object is used in a context where an instance of `Foo` or a reference to a `Foo` is expected the constructor is implicitly called to create a temporary object. `void fun1(Foo f); void fun2(const Foo& f); Foo f; T t; fun1(t); fun2(t); // in both cases a temporary Foo is created from t`
- Whenever a statically allocated instance or an array of instances is leaving its scope, the **destructor** is called.
- Whenever a dynamically created instance or array of instances is deleted by the `delete` or `delete[]` operator, the **destructor** is called.
- The destructor of a derived class implicitly calls the **destructor** of the base class, after its code has been executed.

5.6 Exception Handling

Exception handling is very different from traditional error handling. Both error handling and exception handling deal with *run-time* errors. Often the programmer has the ability to detect a run-time error:

```
int a, b;

cin >> a >> b;

if (!cin) // Something is wrong,
          // possible non integer values ...
```

The situation is more complicated, when the programmer bundles the implementation into a library of functions, classes and namespaces and this library is used to implement applications. The author of the library is able to detect *run-time* errors but it is the programmer of the application who may know how to handle them. The obvious solution is to propagate the error to the application program from the library stopping the line of execution at the point of detection and gracefully cleaning up stack frames and temporary resources while returning from nested calls. This mechanism is called *exception handling* and it is "built into" ANSI C++ . Most pre-ANSI compilers do not implement this mechanism or it is not portable. Many compilers still require a special compiler flag to be "on" to handle exceptions. This mechanism is powerful and allows the programmer to implement sophisticated run-time error handling. However a large part of programs still rely on the "old" C libraries (*stdio, stdlib, math, /X11/Xlib, ...*) which were written long before exceptions were introduced into the language. As C++ did not always have exceptions, there are also many C++ libraries in use today which do not utilize the exception mechanism.

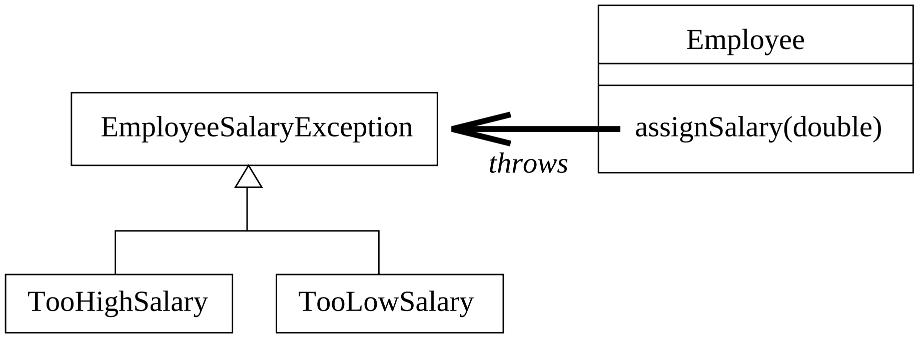
An exception can be *thrown* when a run-time error is detected. As soon as the exception has been thrown – no matter how many nested function calls deep it happened – the function stack frames are gracefully cleaned up and propagation of the exception continues until a piece of code is willing to *catch* it. An exception could actually be an instance of a class that the user can define to carry the "description" of the error. The piece of code where the programmer expects the error to occur must be placed into a *try* block:

```
...
try {
  // .. a run-time error may occur
  //    here
} catch (Exception1 e1) {
  // if error is of type Exception1
  // handle it here
}
...
  catch (ExceptionN en) {
  // if error is of type ExceptionN
  // handle it here
} catch (...) {
  // catch every type of exception
  // handle it here
}
```

`Exception1, ..., ExceptionN` are classes that the programmer can define. As soon as an exception is thrown in the try block, the logical thread of execution continues at the catch block with the same type as the exception. If the exception is not caught (*there is no catch block with a matching declaration*) the exception further propagates. If the exception is not caught in *main*, the program terminates with a run-time error. An exception can also be "partially" handled and "re-thrown" in the *catch* block, by inserting a simple `throw;` clause. The example below is somewhat unusual in the sense that it not only detects the error, but it also handles it. If a lower salary than the minimum is assigned, the salary is automatically raised to the minimum by the handler. The clause

```
   try {
     // whatever
   } catch (...) {
     // handle
   }
```

catches all exceptions. `...` (or *ellipsis*) is actually part of the syntax[3].



```
#include <iostream.h>


class Employee {
   protected:
     double _salary;
     char* _name;

     void copy(double d, char* n) {
        if (_name != NULL) delete [] _name;
        if (n!=NULL) {
          _name = new char[strlen(n)+1];
          strcpy(_name,n);
        }
        _salary =  d;
     }
   public:

     static double  MINIMUM_SALARY;
```

[3]as a related topic, see how can one pass a variable number of arguments to a function or method in the Appendix

```
     static double  MAXIMUM_SALARY;

     Employee(char* n=NULL):_name(NULL) {
        copy(-1,n);
     }

     Employee(const Employee& e):_name(NULL) {
        copy(e._salary,e._name);
     }

     Employee& operator=(const Employee& e) {
        copy(e._salary,e._name);
        return *this;
     }

     virtual ~Employee() {
        delete [] _name;
        _name = NULL;
     }

     void assignSalary(double);

     double salary() {
        return _salary;
     }

     char* name() {
        return _name;
     }
};

class EmployeeSalaryException {
};

class TooLowSalary : public EmployeeSalaryException {
};

class TooHighSalary : public EmployeeSalaryException {
};

void Employee::assignSalary(double d) {
    _salary = d;
    if (d<MINIMUM_SALARY) throw TooLowSalary();
    if (d>MAXIMUM_SALARY) throw TooHighSalary();
}

double Employee::MINIMUM_SALARY =  5000;
double Employee::MAXIMUM_SALARY = 50000;

int main(void) {

    Employee staff[3];
    Employee susan("suzan"),jerry("jerry"),vera("vera");

    staff[0] = susan;
    staff[1] = jerry;
    staff[2] = vera;

    for(int i = 0;i<3;i++) {
      try {
        staff[i].assignSalary(4000+(30000*i));
      } catch(TooHighSalary e) {
        cout << "Too high salary: " << staff[i].salary() << endl;
        staff[i].assignSalary(Employee::MAXIMUM_SALARY);
      } catch (TooLowSalary e) {
        cout << "Too low salary: " << staff[i].salary() << endl;
        staff[i].assignSalary(Employee::MINIMUM_SALARY);
      }
    }

    for(int i=0;i<3;i++) {
      cout << staff[i].name() << " earns: $" << staff[i].salary() << endl;
    }

    return 0;
}
```

The output of the program:

```
Too low salary: 4000
Too high salary: 64000
```

```
suzan earns: $5000
jerry earns: $34000
vera earns: $50000
```

5.7 Pointers to Member Functions

It may be necessary to invoke a member function on an object without actually referring to its name. Such a mechanism may implement a policy: a pointer to a member function is passed to a function or method together with an object. The member function is called on this object pointed to by the pointer. This can facilitate a run-time user or event driven decision on the actual method to be invoked on the object.

```
class A {
   public:
      // a static or class function
      static int foo0();

      // a pure virtual or abstract method
      virtual void foo1() const = 0;

      // a method
      int foo2(const char*);
};


class B: public A {
   public:
      // overloaded function
      virtual void foo1() const;
};

...


int (*p0)();
p0 = A::foo0;

p0();

A* a= new B;
```

```
B b;

void (A::* p1)() const = &(A::foo1);
void (B::* p2)() const = &(b.foo1);
int (A::* p3)(const char*) = &(a->foo2);

(a->*p1)();
(b.*p2)();
(b.*p3)("hello");

...
```

`p0` is a pointer to a function which takes no parameters and returns an `int`. `A::foo0` is a static or class function of class `A` which also takes no parameters and returns an `int`. Hence `A::foo0` can be assigned to `p0` and called via `p0`. In other words, a static or class function *works the same as regular functions* with respect to being passed, called or assigned to pointers.
This of course cannot be true for methods or member functions because they belong to the instance and they need access to the object's instance variables. Hence to call a member function via a pointer we will need a pointer to the member and an object as well! Syntactically a pointer to a member function is declared similarly to a pointer to a function, but the scope resolution operator is needed to identify what class the method belongs to.

```
void (A::* p1)() const;
void (B::* p2)() const;
int (A::* p3)(const char*);
```

`p1` is pointer to *a* member function of class `A` which takes no arguments and has no return value. The method must also be `const`. `p1` can hold the address of any such method of class `A`. `p2` is the same in prototype as `p1` however it must belong to class `B`. Let's pause here for a bit. Class `B` is a subclass of class `A` so can `p2` be assigned `A::foo1`? The answer is no and as we shall see in a bit, a pointer to a member function is not exactly like a function pointer. `p3` is a pointer to a method of class `A` which returns an `int` and takes a `const char*`.
No let's see the assignments.

```
p1 = &(A::foo1);
p1 = &(a->foo1);
```

```
   p2 = &(B::foo2);
   p2 = &(b.foo2);

   p3 = &(A::foo2);
   p3 = &(a->foo2);
```

It is irrelevant whether we obtain the address via an existing object – *i.e.* `a` *or* `b` – or actually via the class. Remember `p1`, `p2` and `p3` are pointers to member functions so they cannot be called unless there is an object is designated. Member functions belong to the instance and they need an instance. So let us take a look at the actual calls.

```
   (a->*p1)();
   (b.*p1)();
   (b.*p2)();
   int k = (b.*p3)("hello");
   (a->*p3)("world");
```

As we already indicated, we need an object and the pointer to the member function. The pointer to the member function cannot be an address in the same sense as a regular function pointer because it may not be resolved compile time. `foo1` is a pure virtual function which means that the actual `foo1` belongs to one of `A`'s subclasses. But this will need dynamic binding. So what does a pointer to member function represent if it may not be possible to have it resolved to a unique address unless the object (*or at least its concrete type*) is known? The pointer to the member function is actually like an index into the object's descriptor or virtual function table which is resolved to the unique address by selecting it from the actual object's descriptor run-time. To be able to do that some syntax must be introduced. Let `p` be a pointer to a method of class `A`, `a` be a pointer an actual instance of `A` or its subclasses and let `b` be an instance of `A` or its subclasses. Then the calls to the method pointed to by `p` on the instances referenced by `a` and `b` are

```
   (a->*p)(.../* parameters */)
   (b.*p)(.../* parameters */)
```

`->*` is the operator combination which is needed to invoke the method if the object is identified by a pointer and `.*` is the operator sequence needed to call the method on an actual instance. Note that in `(a->*p1)()` `p1` is actually `A::foo1` which is an abstract method and the type of `a` is `A` but the actual instance is of class `B`. Because of the run-time resolution (*run-time polymorphism*) the method `B::foo1` is called on the instance pointed to by `a`. To understand run-time polymorphism (*or dynamic binding*) and virtual functions you need to read the next chapter as well.

5.8 Notes on Syntax and Semantics

The role of the constructors and the destructor has been extensively reviewed in this lecture. Every C++ programmer must know it by heart to avoid endless frustration of debugging. Many coding problems – *often committed by "professional" programmers* – are due to not knowing how these mechanisms work. As an advice, write some code which demonstrates every point made in that section and also write code which would cause problems by not using the mechanism properly. This way you should be prepared for not only exams but to be able to identify memory leaks and segmentation faults popping up from "nowhere". If you think you may need C++ , this would be time well invested. Here is a list that you should be able answer and write code to demonstrate it.

- when does one *have to* implement a copy constructor?
- what can go wrong if a class has a destructor but no copy constructor?
- what can go wrong if a class has a copy constructor and no destructor?
- what is the initializer and what are the situations when it is absolutely necessary to use it?
- how do constructors initialize instance variables declared in the super class?
- how does the destructor clean up instance variables declared in the super class?
- how does (*or rather should*) the assignment operator assign values to instance variables declared in the super class?
- when does one not need to implement the copy constructor, the assignment operator and the destructor?
- how do instance variables with no default constructors get initialized?
- how do temporary objects get implicitly created and in what situations?

It is also an appropriate place to review the meaning of `static` in C++ .

- **static global variable and function**: a variable declared outside of functions and methods in a file makes this variable global to the file where it is declared. The purpose is that the identifier associated with the variable will not have a name clash with variables having the same name in other files. In essence such a variable is global but is only visible in the file it is declared. The same way, a static function is only visible in the file where it was declared. When a compiler encounters static global variables and functions, it does not generate symbolic information for the linker so it does not know that they exist.

- **static local variable**: a variable declared locally in a function or method is allocated at compile in the program text – *rather than on the stack* – but is only visible inside the method or the function. This creates the effect that the variable retains its last value from call to call.

- **static member function and variable**: they belong to the class rather than to the instance. Static instance variables are also allocated at compile time – *rather than by the constructor*. Such variables (*class variable*) are only directly visible between instances allocated and hence they share the same physical copy. A static member function (*class method*) also belongs to the class rather than to the instances. Class methods can be used to maintain (*set, get, update*) values of class variables. In some object oriented languages instance creation is achieved by class methods. They can also be used to implement functions which logically should be performed by the class but there is no need for an actual instance to do the computation.

The `inline` mechanism is also quite misunderstood by many programmers. A method or a function declared `inline` is replaced by its body to spare a function call, which of course makes it much faster. However one must be very careful how to use this facility. There is a mystery suggested by some authors that short functions and methods should be made inline. The problem with this heuristic is that the number of lines implementing the function is a false measure of the structure of the function and hence it is an inappropriate guide to use. The programmer should understand what it really means that a call is replaced by the body of the function at compile time and should be able to identify those situations where it is impossible. The following is a list where it is impossible or at least inadvisable to make the function or method inline.

- recursive functions cannot be `inline` – this would result in infinite replacement.

- virtual functions[4] cannot be `inline` because run-time polymorphism – *run-time invocation of the appropriate version of the operation* – cannot be decided compile time.

- `inline` functions cannot be passed to other functions and methods as parameters because they are not functions.

- dynamically linked functions and methods cannot be `inline` – again they are not functions.

- when debugging inline functions, the debugger does not know where to put the cursor. Some debuggers get around this by generating two code streams: one optimized and one without `inline`s and other optimizations turned off for debugging. This often results in the well known phenomenon that the "debug

[4]we will discuss virtual functions and polymorphism in detail later

version works" but the "optimized does not work". Again, in this case it cannot be blamed on the compiler!

- instances passed and returned by value passed to the `inline` function or method may not be replicated (*as they may not be needed*). However, not knowing this may result in virtually undetectable side effects.

- and finally an argument for long (*many line*) functions: excessive replacement with the function body can result in huge program text (*executable files*).

In short, do not use `inline` unless you know exactly what you are doing. And do not use `inline` until you have debugged your code. It is also true that methods implemented in the class definition (*or in essence in the header file*) are `inline` by default. However, remember that the compiler may choose to ignore the `inline` specifier. I believe explicitly specifying what should and what should not be `inline` is the most appropriate – *at least it gives the impression that some thought was put into the decision.* Besides, the compiler may not be smarter than you.

Friend classes and functions should also be used with care and understanding is essential. I think they should be avoided as much as possible. It is of course often beneficial that one class can access private members of another class without invoking a public method. The benefit in this case – *and in most cases with* `friend` – is speed. However this speed can often be achieved by creating a public method which sets or gets the variable of the other class and making it `inline`. The real problem with friend classes is that the privilege granted in the class *does not inherit* to the derived classes and it violates *information hiding* in the sense that one class must be aware of the structure of another instead of going through an interface. There are some examples where `friend` is the easiest way to go but they very rarely occur.

Embedded classes are a new edition to ANSI C++ and they are also present in Java 1.1 (*and higher*). Embedded classes further improve the encapsulation and information hiding capabilities of object oriented programming. Often one class necessarily needs another. For example, linked data structures need a building block class (*often called the "node"*). It is irrelevant for the user of the data structure what an actual node really is but how can it be totally hidden when it has to be defined. One way to hide it – *without embedded classes* – is to declare a class `node` and make all of its members (*including the constructor*) private and give `friend` access privileges to the data structure class. To be able to define `node` inside the data structure class is even better. For one reason it would be visible in derived classes while the friend privilege would not inherit. Also this way the internals can be entirely hidden behind an interface.

Exception handling is the best attempt so far to write programs that can easily detect and react to unusual situations. Exceptions are not necessarily errors and are often recoverable. The biggest contribution of this mechanism is the automatic propagation of the exception to the level where it can finally be handled while resource cleanup is in the process. As exceptions are instances of classes themselves, the programmer can also propagate context information which can help in diagnostics and potential recovery. An exception can also be partially handled by throwing the exception from the catch block. The following small example demonstrates how exceptions can propagate context information and provide partial handling and recovery:

```
class login_exception {
};

class unknown_host_exception : public login_exception {
   protected:
      char host[128];        // computer on network

   public:
      unknown_host_exception(const char* host_name) {
         strcpy(host,host_name);
      }
      const char* get_host() const {
         return host;
      }
};

class unkown_user_exception : public login_exception {
   protected:
      char user[128];        // user on host

   public:
      unknown_passwd_exception(const char* u) {
         strcpy(user,u);
      }
      const char* get_user() const {
         return user;
      }
};



...
```

```
void network::login(const char* host, const char* user) {
   Host H;

   try {
     H = connect(host);
   } catch (unknown_host_exception& e) {
     cerr << "host " << e.get_host() " does not exist!" << endl;
     char new_host[128];
     cout << "Enter a new host name: ";
     cout.flush();
     cin >> new_host;
     login(new_host,user);
   } catch (login_exception& e) {
     cerr << "connection error" << endl;
     throw e;
   }

   char passwd[128];
   cout << "Please enter your password: ";
   cout.flush();
   cin >> passwd;

   try {
      H.login_user(user,passwd);
   } catch(unknown_passwd_exception& e) {
      cerr << "no user" << e.get_user() << endl;
      char new_user[128];
      cout << "Enter user name: ";
      cout.flush();
      cin >> new_user;
      cout << "Enter your password: ";
      cout.flush();
      cin >> passwd;
      H.login_user(new_user,passwd);
   } catch(login_exception& e) {
      cerr << "invalid login" << endl;
      throw e;
   }

}
```

The login method tries to be forgiving. The `connect` method attempts to make a socket connection to the host computer. If the host name is incorrect it allows for one more attempt. For the second try, the exceptions are not caught and therefore a second failed attempt terminates the login process. If the exception is something other than unknown host name, then the message `"connection error"` is printed and the exception is thrown again (*partial handling*). The other potentially troublesome spot is the `login_user` method. Again if the user is unknown on the host another try attempted but the exceptions are not caught for the second try. If the exception is something other than unknown user then the message `"invalid login"` is printed and the exception is thrown again to be potentially caught by the caller.

5.9 Exercises

5.1 In SmallTalk `new` (*instance creation*) is a class method. In C++ , instances are created by the `new operator` which calls the constructor. However C++ cannot provide a universal operator which creates an instance from a logical *serialized form* (*input stream*). Consider the following declaration of the `create` method, which instantiates an object of class A (*or A's subclass*) from a stream.

```
class A {
 public:
   ...
   static A* create(istream&);
    /* return a new A (B or C)
     read from the stream */
   ...
};
...
class B : public A {
  ...
};
...
class C : public A {
  ...
};
```

Explain (*in words*), why it is a good (*or bad*) idea, to implement instance creation from a stream as a class method! Are there other options?

5.2 Given the following class definitions:

```
class A {
 public:
   A() { }
   A(const A&) { }
   ~A() { }
   A& operator=(const A&) { }
};
class B {
 private:
   A a[2];
 public:
   B() { };
   B(const B&) { }
   ~B() { }
   B& operator=(const B& b) {
     A::operator=(b);
   }
};
```

How many times and in what order the constructors and destructors of `A` and `B` are executed if the following function is called:

```
B foo(B b) {
  return b;
}
...
B b1,b2;
b1 = foo(b2);
```

This exercise should clarify the construction and destruction mechanism. Feel free to use the debugger to trace through the code. Add print lines to each of the constructors, destructors and assignment operators.

5.3 We mentioned that the C++ compiler may choose to ignore an `inline` specifier. Give concrete examples when it *must* ignore the `inline` specifier.

Chapter 6

Polymorphism

6.1 Polymorphism

Polymorphism is a mechanism which allows different implementations of the same function to exist in an application, with run-time resolution if needed. Run-time polymorphism is often referred to as *run-time binding*, *dynamic binding* or *overloading*. There is still a need of uniqueness, so the compiler or the run-time environment can resolve the function reference without ambiguity. Because C++ does not have a run-time environment (*it is a compiler*), run-time resolution only means that the address of the function in question is obtained by an extra level of indirection[1]. These functions must be declared *virtual*. On the other hand, often this resolution can take place at compile time. We address these situations first.

A function, *foo*, may have different implementations, because it takes different type of parameters or different number of arguments. This is called *parametric polymorphism*.

```
void foo(int);
void foo(char);
void foo(int,int= 5);
void foo(float);
void foo();
void foo(int *);
void foo(void *);
int  foo(); // ambiguous: void foo()
..........
    int a;
```

[1]Object Oriented interpreters, like SmallTalk , with a run-time environment can do more sophisticated resolution, and there are situations which a SmallTalk interpreter with the context knowledge can resolve, but the C++ generated code cannot. The extra level of indirection can be implemented quite efficiently but it involves using a compile time generated *virtual function table*

```
    char c;
    void *v;
    float f;

    foo(a);  // ambiguous: foo(int), foo(float), foo(char), foo(int, int=5)
    foo(&a);
    foo(v);
    foo(f);
```

The above function prototypes are examples of polymorphic versions of `foo`, however some of them are ambiguous. It is not possible to achieve polymorphism by *just on the return type*. The call to `foo(a)` is ambiguous, because an *int* can be implicitly type casted to a *char* or to a *float*.

We have already seen this mechanism for the constructor methods of objects, in fact, the *copy constructor* and the *default* constructor of a class are polymorphic versions of each other. Any method of a class can have an arbitrary number of (*unambiguous*) polymorphic versions. There is one case, where run time resolution is needed. The most powerful and elegant usage of polymorphism requires inheritance and method overloading. Instances of the derived class are instances of the base class. Hence a pointer variable to the instance of the base class can point to an instance of any of the derived classes. If a method is overloaded in the derived classes then method invocation through the pointer to the instance can only be decided *run-time*.

```
#include <iostream.h>

class A {
 public:
   void foo() { cout << "A::foo()" << endl; } // SHOULD BE VIRTUAL!
};

class B : public A {
 public:
   void foo() { cout << "B::foo()" << endl; }
};

int main(void) {
  A a; B b;
  A* p = &b; A& r = b;


  p->foo();
  r.foo();
  a=b;
  a.foo();
  return 0;
};
```

The output of the program is:

```
A::foo()
A::foo()
A::foo()
```

That is, in all three cases the *wrong* functions were called! The next section explains, how to instruct the compiler to force run-time lookup of the appropriate overloaded method implementation.

6.2 Virtual Methods

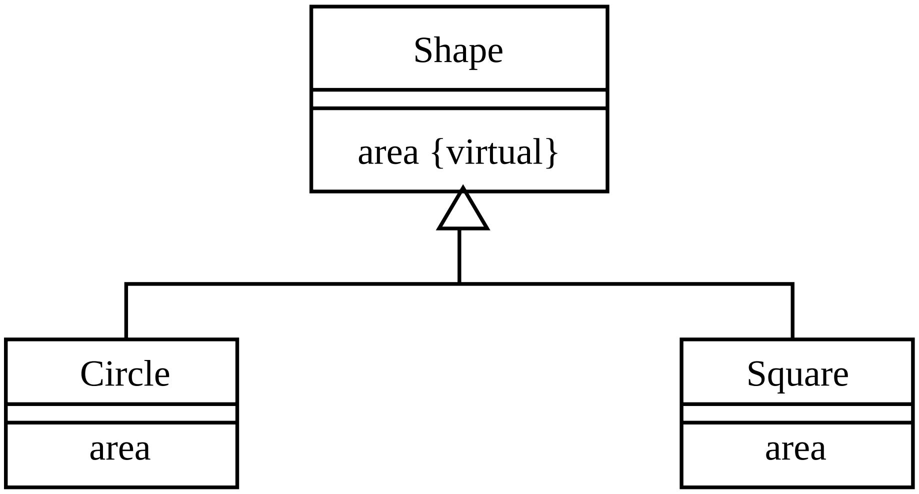
In the example above, the compiler made a static reference to `A::foo` in all three cases. In the case of `p->foo()`, the compiler knows that `p` is of type `A*`, hence the compiler inserts a static reference to the method `A::foo` into the object code. The same thing happens in the case of `r.foo()` and `a.foo()`. `r` is known to be a reference to an object of class `A` and `a` was declared to be an instance of class `A`. C++ , however provides the `virtual` keyword to instruct the compiler *not* to insert a static reference into the machine code for `A::foo`, but to resolve the reference at run-time (*dynamic binding*), depending on the actual object stored in the variable. The modifications to the code:

```
class A {
 public:
   virtual void foo() {
     cout << "A::foo()" << endl;
   }
};

class B : public A {
 public:
   void foo() {
     cout << "B::foo()" << endl;
   }
};
```

Now the output of the program is:

```
B::foo()
B::foo()
A::foo()
```

All but one calls were of the proper function. The one that did not work is `a.foo()`, and this cannot be fixed. This is because `a=b` does not change the descriptor of `a`, or in other words, run-time polymorphism does not apply to statically declared objects[2]. **Run-time polymorphism only works for pointers and references!** The following program demonstrates an application of the use of run-time polymorphism.



```
#include <iostream.h>

const double pi = 3.14159;

class Shape {
  public:
      virtual double area() = 0;
};

class Square : public Shape {
  protected:
      double sidelength;
  public:
      Square(double l=5):sidelength(l) {
      }
```

[2]Expecting run-time polymorphism to work on statically allocated objects is a common mistake. To be fair, this problem has no fix, because statically allocated objects are really allocated at compile time. Java and SmallTalk solve the problem because all objects must be created dynamically with the `new` keyword.

```
      double area() {
        return sidelength*sidelength;
      }
};

class Circle : public Shape {
  protected:
      double radius;
  public:
      Circle(double r=3):radius(r) {
      }

      double area() {
        return radius*radius*pi;
      }
};

int main(void) {
  const int length = 5;

  Shape *shapes[length];

  shapes[0] =  new Square(2);
  shapes[1] =  new Circle(3);
  shapes[2] =  new Circle(5.2);
  shapes[3] =  new Square(4);
  shapes[4] =  new Circle(1);

  double total = 0;

  for(int i=0;i<5;i++) total += shapes[i]->area();

  cout << "The total area of the " << length << " shapes: "
       << total << endl;

  for(int i=0;i<5;i++) delete shapes[i];

  return 0;
}
```

The output is:

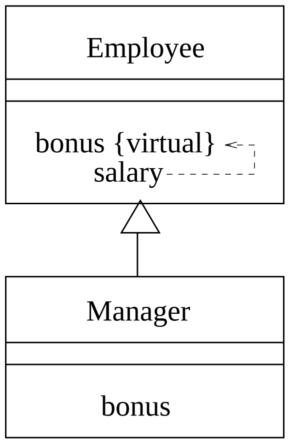
```
The total area of the 5 shapes: 136.364
```

The method `area` of class `Shape` has no implementation; instead we set it to zero. This means that `area` has no implementation in this class, but subclasses must implement the method. A method declared this way is called *pure virtual*. Because the method is not implemented, instances of the base class cannot be declared statically or cannot be allocated by the `new` operator. Pointers and references of the class can point to or alias instances of the non abstract derived classes. This makes `Shape` to be an *abstract class*. Most hierarchies should have abstract classes on the very top. If a *pure virtual* method is not overridden in the derived class then the derived class stays *abstract*. Abstract classes cannot have instances *directly* in the system.

The use of run-time polymorphism is a bit more expensive, so it does not make sense to declare all methods virtual. For example, `area` in the derived classes is not virtual. The following heuristics serve as a *rule of thumb*:

- If a method *may have* a different implementation in a subclass, then the method should be *virtual*
- If a method cannot be overridden or the class will not be subclassed, then the method should not be *virtual*
- All classes which may be subclassed *must* have their destructor declared to be *virtual*

Another use of run-time polymorphism is to insert *hooks* into an application. A *hook* is an explicitly specified entry point which is not implemented but functionality can be plugged in. This mechanism can also be used to extend a particular part of an algorithm without actually touching the initial version.



```
#include <iostream.h>

class Employee {
  protected:
     double pay;
     char* s;

     virtual double bonus() {
         return 0;
     }

  public:
     Employee(char* n,double d=10000):
                  pay(d),s(new char[strlen(n)+1]) {
      strcpy(s,n);
     }

     double salary() {
         return pay+bonus();
     }

     char* name() {
         return s;
     }

     virtual ~Employee() {
         delete[] s;
         s = NULL;
     }
};

class Manager : public Employee {
   protected:
      double extra;

      double bonus() {
         return extra;
      }

   public:
      Manager(char* n,double d=15000, double b=1200):
```

```
                  Employee(n,d),extra(b) {
      }
};


int main(void) {
   const int num_workers = 4;
   Employee *department[num_workers];

   department[0] = new Employee("john",12000);
   department[1] = new Employee("sally",21000);
   department[2] = new Employee("jack",15000);
   department[3] = new Manager("rita",28000);

   for(int i=0;i<num_workers;i++) {
     cout << department[i]->name() << " earns "
          <<  department[i]->salary() << endl;
     delete department[i];
   }

   return 0;
}
```

The output of the program is:

```
john earns 12000
sally earns 21000
jack earns 15000
rita earns 29200
```

Even though `salary` is implemented in class `Employee`, the proper *overloaded* version of `bonus` is called.

`Employee`'s destructor is `virtual`. In general, the destructor should always be virtual if the class is (*or may be*) subclassed. The reason is to ensure the invocation of the appropriate version in the subclasses. While we have noted that destructors do not inherit, we still want to have a run-time decision for the actual type of the instance. In this case, if a pointer of class `Employee` is pointing to an actual instance of class `Manager`, we would want the destructor ∼`Manager` to be called.

6.3 Virtual Destructor

Virtual methods are *virtual* because the compiler may not be able to decide what version of the method to call compile-time and hence it has to be decided with a lookup run-time. Destructors are not regular methods because they do not inherit – instead the destructor of the derived class implicitly calls the destructor of the base class after it has finished executing its own code. On the other hand, it may be necessary to delay the decision of invoking the appropriate destructor for an object until run-time. Consider the following simple example:

```
class A {
   public:
     ...
     ~A() { cout << "A::~A()" << endl; }  // ~A() should be virtual
};

class B: public A {
   public:
     ...
     ~B() { cout << "B::~B()" << endl; }
};


...
A * a = new B();

delete a; // OOPS A::~A is called instead of B::~B
```

Unless ∼A is *virtual* the compiler would assume that `a` is pointing to an instance of `A` which is *not* an instance of `B`. Not invoking the appropriate destructor can cause a memory leak if there are dynamically allocated instance variables in class `B`. This is demonstrated by the example below:

```
class A {
  public:
    A() { cout << "A::A" << endl;}
    ~A() { cout << "A::~A" << endl; } // should be virtual
};

class B : public A{
  private:
    int* huge_array;
  public:
    B() {
      huge_array = new int[10000];
      cout << "B::B" << endl;
    }
    ~B() {
      delete [] huge_array;
      cout << "B::~B" << endl;
    }
};

int main(void) {
   A* a;
   for(int i=0; i<1000; i++) {
     a = new B;
     delete a; // does not call the right destructor!!
   } // it eventually runs out of memory and crashes

}
```

Constructors on the other hand *cannot* be virtual in this sense, because it must be explicitly specified either at declaration or for the `new` operator which subclass the instance belongs to.

6.4 Operator Overloading

Operators are the most likely candidates to be overloaded because they abstract an operation which can logically be applied on many types. For example, C++ knows how to read and write native data types, but it does not provide the basic I/O operators ($\ll$ *and* $\gg$) for objects. Basic input and output makes sense for most objects. Arithmetic operators are only defined for numeric types, but the `+` operator could be used to concatenate strings. Most arithmetic operators make sense to be overloaded for more complex numeric objects, such as *complex numbers*, *matrices* and *fractions*.

Overloading Arithmetic Operators

The arithmetic operators are: `+,-,*,/` and `%`. Each of these operators have an associated assignment operator: `+=, -=, *=, /=` and `%=`. Arithmetic operators only make sense to be overloaded, if the object in question is a numeric type. C++ does not restrict the use of this mechanism, but an operator for a class should not be overloaded, if the operator does not have a clear and well defined meaning (*for example, the multiplication operator can be overloaded for class Employee, but does it make sense?*).

```
class Integer {
  protected:
     int i;
  public:
     Integer(int j=0):i(j) {
     }

     Integer operator+(const Integer& b) const {
        return Integer(b.i+i);
     }
};

   ...
   b2 = b1 + b3; // ok
   b2 = b2 + 4;  // ok
   b2 = b3 + 2;  // ok, because operator+ is const

   b2 = 4 + b2;  // error 4 is not on object of type Integer
   ...
```

The assignment `b2 = b2 + 4` works but `b2 = 2 + b2` does not. The reason is, that *4* is a literal constant of type `int`. `Integer Integer::operator+(const Integer&) const` only works if the variable on the left *is* an instance of class *Integer*. This can be circumvented, if we overload `Integer operator+(const Integer&, const Integer&)`. The former is a member function of class `Integer`, while the latter is a *free* function, which takes two `Integer` arguments. Operators can be overloaded as *member* and as *free* functions. The former is the more *object oriented*, however the latter is more appropriate if the operator is to be applied on "literal" constants. If the operator is overloaded as a free function for class `C`, then either operand can be an instance of a class `B` for which a constructor of type `C::C(const B&)` is defined.

The following program demonstrates arithmetic operator overloading for class `complex`.

```
class complex {
   private:
     double re;
     double im;
   public:
     complex(double =0,double =0);
     friend complex  operator +(const complex&,const complex&);
     friend complex  operator *(const complex&,const complex&);
     friend complex  operator /(const complex&,const complex&);
     friend complex  operator -(const complex&,const complex&);
```

```
    friend complex  operator ~(const complex&); // conjugate
};

complex::complex(double r,double i):re(r),im(i) {
}

complex operator +(const complex& c1, const complex& c2) {
   return complex(c1.re+c2.re,c1.im+c2.im);
}

complex operator *(const complex& c1, const complex& c2) {
   return complex(c1.re*c2.re-c1.im*c2.im,c1.re*c2.im+c1.im*c2.re);
}

complex operator -(const complex& c1,const complex& c2) {
   return complex(c1.re-c2.re,c1.im-c2.im);
}

complex operator ~(const complex& c) {
   return complex(c.re,-c.im);
}

complex operator /(const complex& c1, const complex& c2) {
   return complex((c1*~c2).re/(c2*~c2).re,(c1*~c2).im/(c2*~c2).re);
}
  ...
  complex c1,c2;
  const complex c3(3,-4);

  c2 = c1 + c3;  // ok
  c2 = c3 + c1;  // ok
  c2 = c2 * 5;   // ok
  c2 = 4 / c2;   // ok
  c2 = c3 - 1;   // ok
  ...
```

Overloading Assignment Operators

C++ has many assignment operators. The ones of interest for this course are: `=`, `+=`, `-=`, `*=`, `/=`[3]. If there is an overloaded assignment (`=`) operator and `+` operator for a class, the compiler *does not* provide the `+=` operator. The assignment operators *should*

[3]Some others are: `&&=`, `||=` for logical and `&=`, `|=` for bitwise operators

not be implemented as *friend* functions, because constants cannot be assigned to. The following modifications are needed to the `complex` class to include the assignment operators:

```
class complex {
   public:
     ....
     complex& operator=(const complex&);
     complex& operator+=(const complex&);
     complex& operator-=(const complex&);
     complex& operator*=(const complex&);
     complex& operator/=(const complex&);
     ....
};

complex& complex::operator=(const complex& c) {
   // for this particular class, the default assignment
   // operator is identical to this one, so this implementation
   // is redundant
   re = c.re;
   im = c.im;
   return *this;
  }

complex& complex::operator+=(const complex& c) {
   re += c.re;
   im += c.im;
   return *this; // or simply return (*this = *this + c)
  }

complex& complex::operator-=(const complex& c) {
   re -= c.re;
   im -= c.im;
   return *this; // or simply return (*this = *this - c)
  }

complex& complex::operator*=(const complex& c) {
   re = re*c.re-im*c.im;
   im = re*c.im+im*c.re;
   return *this; // or simply return (*this = *this * c)
  }
```

```
complex& complex::operator/=(const complex& c) {
   complex tmp = *this;  // why do we need a temporary ??
   re = (tmp*~c).re / (c*c).re;
   im = (tmp*~c).im / (c*c).re;
   return *this; // or simply return (*this = *this / c)
  }
```

Overloading Comparison Operators

Comparison operators return *true* or *false*. In C , true is the same as a non-zero value and false is always zero. ANSI C++ defines the type `bool` which is a true boolean type. The modifications needed to the `complex` class to implement the comparison operators [4]:

```
class complex {
   private:
     .....
     double abs_sqr() const; // absolute value squared
   public:
     .....
     friend bool operator==(const complex&,const complex&);
     friend bool operator<=(const complex&,const complex&);
     friend bool operator>=(const complex&,const complex&);
     friend bool operator>(const complex&,const complex&);
     friend bool operator<(const complex&,const complex&);
     friend bool operator!=(const complex&,const complex&);
};

double complex::abs_sqr() const{
   return (re*re) + (im*im);
}

bool operator==(const complex& c1,const complex& c2) {
   return (c1.re == c2.re) && (c1.im == c2.im);
}

bool operator<=(const complex& c1,const complex& c2) {
   return c1.abs_sqr() <= c2.abs_sqr();
}
```

[4]every logical comparison operator can be implemented using only one of `operator<`, `operator>`, `operator<=` or `operator>=`! In a later section, we discuss why that would be more advantageous! *See the example on page 202*

```
bool operator>=(const complex& c1,const complex& c2) {
   return c1.abs_sqr() >= c2.abs_sqr();
}

bool operator>(const complex& c1,const complex& c2) {
   return c1.abs_sqr() > c2.abs_sqr();
}

bool operator<(const complex& c1,const complex& c2) {
   return c1.abs_sqr() < c2.abs_sqr();
}

bool operator!=(const complex& c1,const complex& c2) {
   return  (c1.re != c2.re) || (c1.im != c2.im);
}
```

Overloading $\ll$ and $\gg$

Operator $\ll$ and $\gg$ can be overloaded for any class, however, for a class `C`, overloading `ostream& operator≪(ostream&,const C&)` and `istream& operator≪(istream&,C&)` empower instances of the class to be written to and read from streams. These not only include the standard output (`cout`) and the standard input (`cin`), but all subclasses of `ostream` and `istream`. For example, once these operators are implemented, an instance can be instantiated from a file or can be saved in a file (*made persistent*). The following additions to class `complex` are needed to overload the $\ll$ and $\gg$ operators:

```
class complex {
   public:
     friend ostream& operator<<(ostream&,const complex&);
     friend istream& operator>>(istream&,complex&);
};

ostream& operator<<(ostream& os,const complex& c) {
   return os << c.re << '+' << c.im << 'i';
}

class ParseError {
};

istream& operator>>(istream& is,complex& c) {
   // format assumed <real>[+|-]<imaginary>i
   // examples:   4+3i, 6-7i
   char sign,i;
```

```
  is >> c.re >> sign >> c.im >> i;

  if (!is) throw ParseError;
  if (sign == '-') c.im = -c.im;
  else if (sign != '+') throw ParseError;
  if (i!='i') throw ParseError;

  return is;
 }
```

Overloading `new`, `new[]`, `delete` and `delete[]`

C++ actually allows the programmer to overload the free store operators and take memory management under tight control. These operators have global versions but they can also be overridden for a specific class. Overwriting the global free store operators requires substantial knowledge and some very good reason. In this case, your implementation would be called instead of the one which works with the operating system the compiler was built for. It is much more common that the free store operators are overloaded for a particular class.

```
class C {
   ...
   public:
      ...
      void* operator new(size_t);
      void* operator new[](size_t);

      void operator delete(void*, size_t);
      void operator delete[](void*, size_t);
};
```

The `size_t` parameter is the actual size of the type or what operator `sizeof` would return. `size_t` is usually just a typedef for a `long`. One must be very careful when overloading these free store operators! A call to `new` inside the implementation of `new` is a recursive call! So one should probably use `malloc` or even better the `new` operator of another class. It is also important to keep size information and alignment to avoid memory leaks and for efficiency.

There are other operators, which can be overloaded. Later in the course, we overload the [], `++`, `--` operators for container classes.

6.5 Notes on Syntax and Semantics

It is sometimes a source of confusion for beginner programmers which methods of a class should be virtual. A method must be virtual if it is or if it will likely be overridden in a subclass. This facilitates the appropriate version of the method to be invoked through a reference or a pointer to an instance of the super class – in essence it allows for the run-time decision of method invocation. Actually at the machine level it is quite trivial to implement and to clarify things the next example shows how the compiler actual achieves this.

```
class A {
   public:
      virtual void foo1() {
         cout << "A" << endl;
      }

      void foo2() {
         cout << "A" << endl;
      }
};

class B : public A {
   public:
      void foo1() {
         cout << "B" << endl;
      }

      void foo2() {
         cout << "B" << endl;
      }
};

....

A * ptr = new B;

ptr->foo1();  // outputs "B"
ptr->foo2();  // outputs "A" ????
```

Once a method is defined `virtual` in a class this method is given an entry in the *virtual function table*. The virtual function table is just an array of pointers to the actual implementations. In this case, all instances of class `A`. Hence the descriptor of an `A` instance looks like this:

A	
method	**address**
`foo1`	→ `vtbl`
`foo2`	→ `A::foo2`

B::vtbl	
method	**address**
`foo1`	→ `B::foo1`

The virtual function table (*vtbl*) stores the locations of the methods for each class and each instance of `A` has a pointer to a *vtbl*, which could belong to `A` or its subclasses. Run-time polymorphism is probably the biggest contribution of the object oriented paradigm. Using inheritance together with polymorphism makes carefully designed applications easily extendable. There is practical evidence that the structural make up of an application is more static than the functionality it must perform. It is also often very difficult to integrate new functionality into existing applications. The object oriented paradigm focuses on the structure – *the building blocks and their associations*. The classes usually represent concrete entities and concepts but they very often can be abstracted. For example, a `bank_account` is an abstraction of `savings_account`. The functionality associated with `bank_account` such as `withdraw` apply to all kind of accounts but more specialized accounts may require a different way to perform the task. A model designed and implemented in terms of the abstract entities can easily incorporate such additions. If all kind of accounts are referred to through a pointer to the abstract class `bank_account`, then the invocation of method `withdraw` can be resolved run-time to the actual implementation (*which could be* `savings_account`, `checking_account`, `corporate_account`, *etc.*). Adding a new account, like `student_account` only requires to create the subclass and to override a few methods. As the calls to the methods are already in place in the application – *via a pointer to the super class* – `student_account` is automatically integrated. Of course it is not always possible to get away with such little work but a well designed model can handle a lot of additions and extensions with manageable effort. When the model can no longer incorporate changes without significant rework it is said to be saturated and redesign maybe necessary. An implementation without a carefully designed model is saturated as soon as the first line of code is written. A good object model is likely to be able to handle a lot before it saturates. This is the very reason why the object oriented paradigm is pushed in industry. Most of the effort and resources of development are spent on software maintenance.

6.6 Exercises

6.1 Let `B` be a class with a *non* virtual destructor. Let class `C` be derived from `B`. Give a small and concrete example (*as* C++ *code!*), where the sole problem is the fact that `B`'s destructor is not virtual!

6.2 Design a hierarchy of animals of at least two levels of inheritance and 10 classes of which there should be at least 3 abstract. Each class should implement the method `talk`, which prints the sound the animal makes. For example `cow abel; abel.talk()` would write `moo`. Read instances of animals from the keyboard into an array of pointers to instances of abstract class `animal`, and then print out the sounds they make in a loop.

6.3 Implement a `matrix` class, and overload the '*', '+' and '-' operators, as defined for matrices.

6.4 In addition to the operators defined for `matrix` in the previous exercise, overload the operator '/' to divide matrices. *Hint: $A/B = A \times B^{-1}$, use Gaussian elimination to implement the inverse*

6.5 Implement a `fraction` class, and overload the '*', '+', '-' and '/' operators. The fraction should always be in reduced form. *Hint: Use Euclide's algorithm to calculate the greatest common divisor. Calculate $\frac{n_1}{d_2} \cdot \frac{n_2}{d_2}$ as $\frac{\frac{n_1}{gcd(n_1,d_2)}}{\frac{d_1}{gcd(n_2,d_1)}} \cdot \frac{\frac{n_2}{gcd(n_2,d_1)}}{\frac{d_2}{gcd(n_1,d_2)}}$ and $\frac{n_1}{d_2} + \frac{n_2}{d_2}$ as $\frac{\frac{lcm(d_1,d_2)}{d_1} \cdot n_1}{lcm(d_1,s_2)} + \frac{\frac{lcm(d_1,d_2)}{d_2} \cdot n_2}{lcm(d_1,s_2)}$! Why??*

Chapter 7

Generic Programming

Generic programming is the ability to supply type information as a parameter to class definitions and functions. ANSI C++ is one of those rare languages which supports such a mechanism. This chapter mainly focuses on *container classes* which are probably the prime example why *templates* or parametric types are so important.

7.1 Container Classes

Container classes hold multiple objects which logically or semantically belong together. A container class is the data it holds together with the operations it provides to access, remove and add elements. The most common and well known container is the *array* which is directly supported by most programming languages. There are many different kind of container classes and they can be grouped according to several criteria; every container class belongs to one or more to the following categories:

Size:

- *Fixed size*: size of the container is determined at compile time or run time, but it cannot change afterwards. (*array, some implementations of vector, ...*)
- *Bounded size*: size of the container cannot exceed or cannot be smaller than a compile time or run-time determined measure. (*some implementations of stacks, vectors, queues,...*)
- *Variable size*: size of the container during run time may freely change and is only bounded by available memory. (*most "linked" data structures*)

Access:

- *Random access*: every element stored in the container can be accessed at any time and order, regardless of the order it was put in or what its positional relationship is with respect to the other elements. (*array, list, vector, hashtable, ...*)

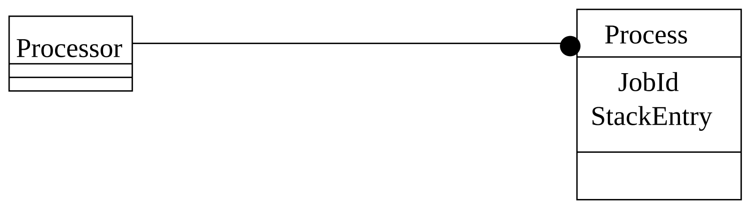
- *Restricted access*: only elements with a certain property can be accessed. (*stack's top, queue's first, priority queue's highest priority, heap's largest, ...*)

Special properties:

- *Ordering property*: elements of the container can be compared, and an element's magnitude is reflected in its position in the container. (*binary search tree: the left child's magnitude is less or equal to the parent node's, ...*)

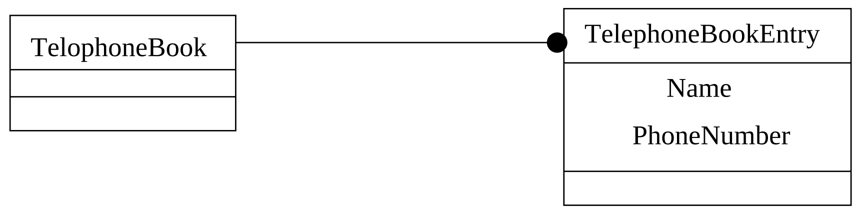
- *Structural property*: the structural organization of the elements in the container at the implementation level satisfy some criteria. (*an AVL tree is height balanced, ...*)

Special properties of container classes are often present to optimize the most frequently used operations. The object oriented paradigm is very suitable to implement container classes because it provides encapsulation and information hiding. The role of container classes is also more significant in object oriented programming because they can be used to implement associations and to utilize polymorphism. When one object is associated with many objects, it is usually a container class instance variable which holds the references to the objects. Iteratively traversing a container class and applying operations on the elements together with run time polymorphism can significantly simplify the logic and complexity of a program.



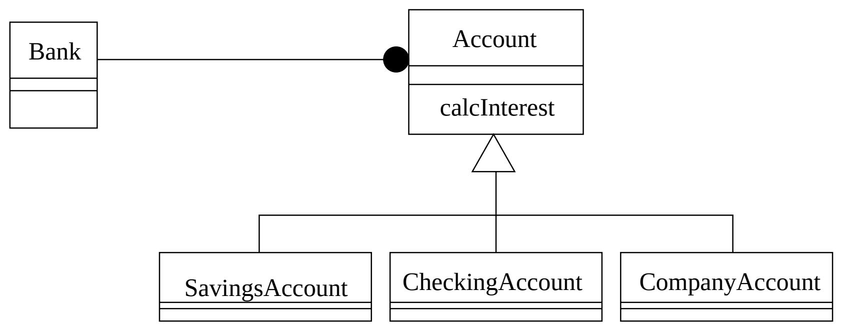
```
class Processor {
  private:
      PriorityQueue<Process>   processes;
  ....
}
```

```
class Proces {
  private:
     int JobId;
     void* StackEntry;
  ....
}
```



```
class TelephoneBook {
  private:
   AVLTree<TelephoneBookEntry> entries;
   ...
}
```

```
class TelephoneBookEntry {
  private:
     String Name;
     Phone PhoneNumber;
  ...
}
```



```
class Bank {
  private:
      list<Account*> accounts;
      ....
}
```

```
double Bank::TotalInterest() {
     double total = 0;
     list_iterator<Account*> currentAccount = accounts.first();
     while(currentAccount.hasMoreElements()) {
            total += currentAccount->calcInterest();
            currentAccount++;
     }
     return total;
}
```

List

Lists are usually random access, variable size containers optimized for random insertion and deletion. Unlike arrays, they can perform insertion and deletion in constant time, while accessing elements may require performing a number of operations proportional to the size of the list[1].

Operations	**Complexity**
insert_(before,after)	$O_{(1)}$
delete_this	$O_{(1)}$
append_copy	$O_{(n)}$
find	$O_{(n)}$
element_at	$O_{(n)}$ or $O_{(\log(n))}$

Most often lists are implemented with links to the previous and next elements. (*doubly linked list*).

Vector

A vector is a random access, variable or bounded size container optimized for random access.

Operations	**Complexity**
insert_at	$O_{(1)}$
element_at	$O_{(1)}$

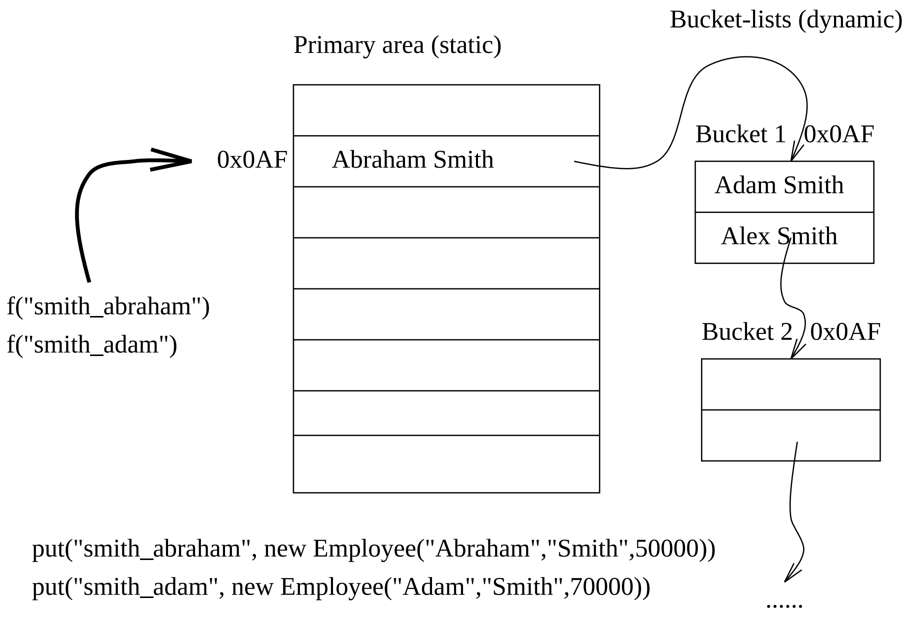
For truly variable size arrays, constant time random access cannot be guaranteed at all times because it may require new memory allocation; however if the expected size of the vector is known, most implementations can custom optimize.

Hashtable

A hashtable is a random access, variable size data structure optimized for retrieval. Each element of the hashtable is associated with an object called the *key* which is used as an index. In fact, a hashtable stores *(key, value)* pairs. The user of the table is also required to provide a hash function f which given the key can determine the location of the value associated with

[1]when random access is of major concern, lists are also associated with a dedicated data structure for indexing; usually a **B+** tree

it. When more than one keys *hash* to the same location, values are stored outside of the hashtable in "buckets". A hashtable is "full" when new values must be stored in the "buckets" and performance quickly degrades. It is often difficult to find an adequate hash function or to decide on the size of the primary area or the size of the buckets. Good hash functions exist for integers, real numbers, strings and calendar dates.



Operations	Complexity
put	expected: $O_{(1)}$
	worst case: $O_{(n)}$ or $O_{(\log(n))}$
get	expected: $O_{(1)}$
	worst case: $O_{(n)}$ or $O_{(\log(n))}$

Stack

A stack is a restricted access, variable size data structure. The last element inserted (*push*) is the first to be removed (*pop*).

Operations	Complexity
push	$O_{(1)}$
pop	$O_{(1)}$

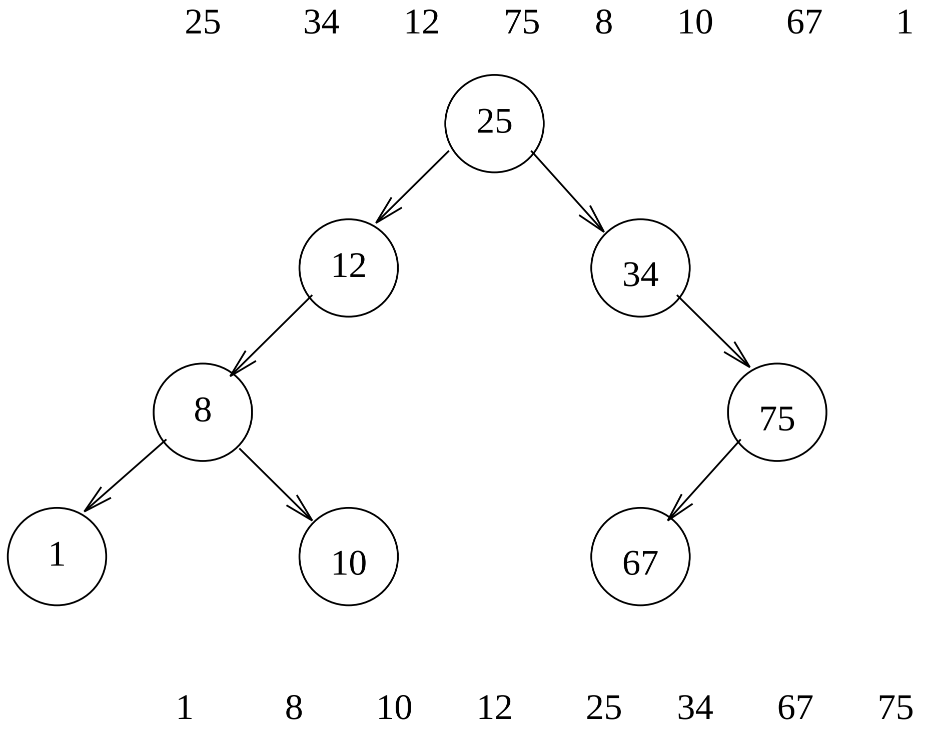
Queue and **Priority Queue**

Both queues and priority queues are restricted access, variable size data structures. The element with the highest priority is removed first. For priority queues, the user is expected to provide a comparison function to establish the priorities. Regular queues implement temporal ordering. Priority queues are usually implemented with *binary heaps*.

Operations	**Complexity**
enqueue	temporal: $O_{(1)}$
	priority: $O_{(\log(n))}$
dequeue	temporal: $O_{(1)}$
	priority: $O_{(\log(n))}$

Binary Search Tree

Binary search trees usually store elements which must be sorted at all times. There are variations which specifically used for indexing. They are usually random access and variable size.



Operations	**Complexity**
insert	$O_{(\log(n))}$
delete	$O_{(\log(n))}$
find	$O_{(\log(n))}$
sort	$O_{(n\times\log(n))}$

7.2 Iterators

Iterators provide a uniform mechanism to traverse container classes. Since container classes store more than one elements, it often makes sense to iterate through the data structure visiting and/or applying some operation to all of them. In some sense an iterator is a generalization of a pointer with the same syntax and semantics of pointer arithmetic. An iterator at any time points to one element and depending on the container class the usual pointer arithmetic syntax can be used to get to successive elements. Iterators can also be classified as one of *bidirectional iterator*, *forward iterator* and *random access iterator*[2] Let `I` be an iterator and `n` an integer.

Forward Iterators

Syntax	Value	Side Effects
`*I`	the value pointed to by `I`	
`I++`	the value pointed to by `I`	`I` moves forwards
`++I`	the value pointed to by `I+1`	`I` moves forwards

Bidirectional Iterators

Syntax	Value	Side Effects
`*I`	the value pointed to by `I`	
`I++`	the value pointed to by `I`	`I` moves forwards
`++I`	the value pointed to by `I+1`	`I` moves forwards
`I--`	the value pointed to by `I`	`I` moves backwards
`--I`	the value pointed to by `I-1`	`I` moves backwards

Random Access Iterators

Syntax	Value	Side Effects
`*I`	the value pointed to by `I`	
`I++`	the value pointed to by `I`	`I` moves forwards
`++I`	the value pointed to by `I+1`	`I` moves forwards
`I--`	the value pointed to by `I`	`I` moves backwards
`--I`	the value pointed to by `I-1`	`I` moves backwards
`I[n]`	the value pointed to by `I+n`	
`*(I+n)`	the value pointed to by `I+n`	

[2]The Standard Template Library (*STL*) also has *input* and *output* iterators , but we follow a simplified model. The Standard Template Library provides a number of iterators and container classes, and they should be used. The `list_iterator` and `list` class presented here are simplifications of the ones provided by the Standard Template Library. The full implementation is outside of the scope of these notes. For documentation on container and iterator API refer to *Alexander Stepanov, Meng Lee: The Standard Template Library, 1994 Hewlett-Packard Co.*

```
list_iterator<int> I = L.first();

int fifth_element = I[4];

while( I.hasMoreElements() ) {
  // do something with or to *I
  I++;
}
```

Structural changes to the container may invalidate an iterator which is pointing into an element. This may cause hard to fix run-time errors. In general, iterators should be short lived and special care must be taken to avoid inconsistency in distributed or multi-threaded applications.

7.3 Templates

Templates facilitate *parametric programming* or *generic programming*. It is wasteful and hardly maintainable to have different implementations of a list because they store different type of values (*integers, character strings, employees, ...*). Templates allow the programmer to declare classes where the type information is provided by a parameter. For example:

```
list<int> L1;    // an integer list
list<char*> L2;  // a list of strings
list<person> L3; // a list of persons
```

It is not any more complicated to define parametric classes than standard classes. For example, the following few lines define class "triple" which holds three elements of the type provided as a parameter:

```
template<class T>
class triple {
  private:
     T _first;
     T _second;
     T _third;
  public:
     triple(T,T,T);
     T second() const;
     void apply(void (*)(T&));
};

template<class T>
triple<T>::triple(T t1,T t2,T t3):_first(t1),_second(t2),_third(t3) {
};


template<class T>
T triple<T>::second() const {
  return _second;
};

template<class T>
void triple<T>::apply(void (*f)(T&)) {
  f(_first);
  f(_second);
  f(_third);
};

void add2(int& i) { i += 2; }

void printInt(int& i) { cout << i << ' '; }

void printReverseString(char*& s) {
    int l = strlen(s);
    for(int i=l-1;i>=0;i--) {
       cout << s[i];
    }
    cout << ' ';
}


....

triple<int> T1(3,4,5);

T1.apply(add2);
T1.apply(printInt);
cout << endl;

triple<char*> T2("abel","eugene","sarah");
```

```
  T2.apply(printReverseString);
  cout << endl;
```

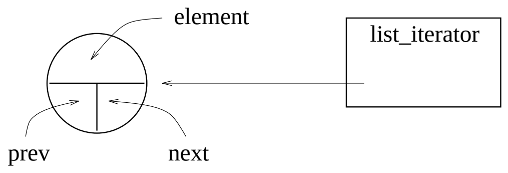
One must be careful when using container classes. Run-time polymorphism only works for pointers and references. Hence container classes are usually holding pointers to dynamically created instances. If that is the case, the destructor of the container class will not deallocate the instances held so the programmer must make sure to deallocate them before the last references go out of scope. The following example shows a common way of handling this situation.

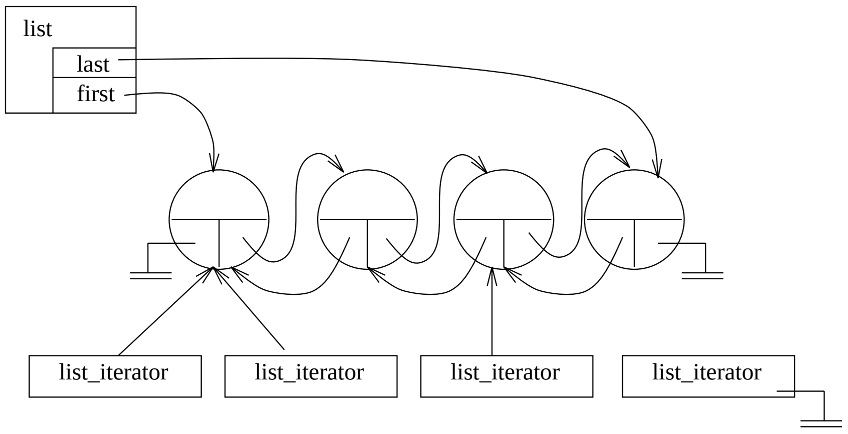
```
class shape {
  ...
  virtual double area() = 0;

};


class circle: public shape {
  ...
  double area();
};


class square: public shape {
  ...
  double area();
};



class shapeList {
  protected:
     list<shape*> elements;
  public:
     void insert(shape*);
     shape* get(list_iterator<shape*>);
     ...
     ~shapeList();
};

shapeList::~shapeList() {

  list_iterator<shape*> I = elements.first();

  while(I.hasMoreElements()) {
    delete *I;
    I++;
  }
}
```

One must be careful. Sometimes a data structure is meant to hold only aliases and is not supposed to deallocate them. Very often a more sophisticated deallocation scheme is needed.

7.4 Implementing a List

The following figure shows a run-time snapshot of a list. The `list_node` is the building block of the list and in essence class `list` is just a wrapper. This list is *doubly linked* so it can be traversed in both directions. A `list_iterator`, as shown on the figure is associated with one particular element of the list. There could also be iterators that are not associated with elements (*null*).





list_node

```
template<class T>
class list_node {
  protected:
   list_node<T>* next;
   list_node<T>* prev;
   T element;
   list_node();
   list_node(const T&);

  friend class list_iterator<T>;
  friend class list<T>;
};
```

list

The following definition of class `list` defines the operations. Random access operations (`insert_before`, `insert_after` and `remove`) take iterator parameters. This definition does not include those `private` and `protected` instance variables and methods which are possibly needed to implement the public interface (*see exercises!*).

```
template<class T>
class list {
  protected:

     list_node<T>* _first;
     list_node<T>* _last;
     int size;

  public:

     // list_iterator<T> i
     // if (!i.hasMoreElements) insert as first
     // otherwise insert before i
     void insert_before(list_iterator<T>,const T&);
     void insert_before(list_iterator<T>,const list<T>&);
     // if (!i.hasMoreElements) insert as last
     // otherwise insert after i
     void insert_after(list_iterator<T>,const T&);
     void insert_after(list_iterator<T>,const list<T>&);
     // remove element at i
     void remove(list_iterator<T>&);
     list_iterator<T> first() const;
     list_iterator<T> last() const;
     int length() const;


     list();
     list(const list<T>&);
     list<T>& operator=(const list<T>&);

     virtual ~list();
};
```

When implementing the methods, the programmer must specify which instantiation of `list` the method belongs to. In other words, it is not enough to write `int list::length() { return size;}`

```
template<class T>
int list<T>::length() const {
  return size;
}

template<class T>
list_iterator<T> list<T>::first() const {
  list_iterator<T> I(_first);
  return I;
};
```

list_iterator

The following definition of `list_iterator` defines the usual bidirectional access methods:

> `operator*, operator->`: dereferencing
>
> `operator++, operator--`: increment
>
> `operator[]`: random access

The method `hasMoreElements()` is "borrowed" from the Java `Enumeration` class, and is somewhat simpler than the equivalent operations of the Standard Template Library .

```
template<class T>
class list_iterator {
  protected:
    list_node<T>* node;

  public:
    // by default, point nowhere
    list_iterator(list_node<T>* = NULL);
    bool hasMoreElements();
    T& operator*();     // *node
    T* operator->();    // *node
    T& operator++();    // node->=next, *node
    T& operator--();    // node->=prev, *node
    T& operator++(int); // *node, node->=next
    T& operator--(int); // *node, node->=prev
    bool operator==(const list_iterator<T>&);
    bool operator!=(const list_iterator<T>&);
    T& operator[](int); // *(node + n)
    list_iterator<T>  operator+(int);
    list_iterator<T>  operator-(int);

    list_node<T>* getNode();
    void invalidate();  // node=NULL
};

template<class T>
T& list_iterator<T>::operator*() {
  return node->element;
};

template<class T>
T& list_iterator<T>::operator++() {
  // prefix
  node = node->next;
  return node->element;
};

template<class T>
T& list_iterator<T>::operator++(int) {
  // postfix
  T& remember = node->element;
  node = node->next;
  return remember;
};

template<class T>
list_iterator<T>  list_iterator<T>::operator+(int n) {
  // like pointer arithmetic
  list_iterator<T> I;
  I.node = this->node;
  for(int i=0;i<n;i++) I++;

  return I;
};

template<class T>
T& list_iterator<T>::operator[](int n) {
  return (*this + n).node->element;
};
```

Now list is a parametric container class which can be used to hold any type or class of values and it also has an associated iterator.

```
list<int> L;
list_iterator<int> I;

// append an element to the end of the list
L.insert_after(L.last(),5);

// add an element at the front of the list
L.insert_before(L.first(),4);

// traverse the list
while(I.hasMoreElements()) {
  cout << *I << ' ';
  I++;
}

// the fifth element of the list
if (L.length()>=5) {
   I = L.first();
   cout << I[4]; // or *(I+4)
}

// the element before the last one
if (L.length()>=2) {
   I = L.last();
   cout << I[-1]; // or *(I-1)
}
```

These definitions do not exactly show how the Standard Template Library implements iterators and containers nor they offer a better alternative. For simple containers which are part of STL, the programmer should refer to an STL documentation and use the library. It is very possible that the application demands a new kind of container not available in STL. In that case, one should still make an attempt to subclass a container or if that is not possible, keep the standard interface so the new data structure can be traversed by iterators. The exercises that follow could serve as a first introduction to learn the containers and iterators of the Standard Template Library.

7.5 Notes on Syntax and Semantics

Templates support generic programming or the ability to use types as parameters. The power of this mechanism is well demonstrated with container classes where the type information the data structure holds indeed *should* be a parameter. This idea can easily be extended beyond container classes. Some algorithms can also be parameterized – for example sorting performs the same steps regardless of the type of the objects.
The problem with templates is instantiation. The type descriptors cannot be built until a template class is instantiated with a type. In other words a `Vector<int>` is not the same type as a `Vector<Student>`. Hence compiling the file `Vector.cc` which contains the parametric definitions for `template <class T> Vector<T>` does not make too much sense unless `T` is known. However when a `Vector<int>` is defined in another file, the compiler must know where the template definitions are to be able to build the type descriptors.
A common way to avoid this problem is to include the implementation file as well via an `#include` directive. This however has its caveats. For one, every file which has a `Vector<int>` will have identical but separate definition in each object file. This should not cause problems if the *Makefile*[3] appropriately reflects dependencies but this may pose a challenge and has maintenance drawbacks. Some compilers – like *Borland* – try to come around this and implement *smart* template instantiations, where the compiler

[3]Makefiles are discussed in the Appendix

tries to collect information on template definitions and declarations on a separate pass and uses it to decide where and how to build the descriptors.
Another way to avoid this is to specify *external template generation.* In this case all template descriptors for a class instantiated with different types are in the same file – *as they should be.* In this case however the implementation file must be somehow aware what classes will be used to instantiate the template classes. I have been using the following schema for a long time, it solves the above problems.

```
// Header file "A.h"
#if !defined(_A_H_)
#define _A_H_
#pragma interface

template<class T>
class A {
   ... // class A has member(s) of type T
};

template<class T>void foo(const T&);
// function foo is parametric

#endif
```

```
// File A.cc
// Implementation of class A<T> and
// foo(const T&)
#pragma implementation "A.h"
#include "A.h"
...
#if !defined(A_INST)
// A_INST is the file which defines instantiations
// for A<T> and foo<T>
#error "A_INST must be defined"
#else
#include A_INST
#endif
```

```
// File my_inst.h
// instantiations for class A<T> and foo
#if !defined(_MY_INST_H_)
#define _MY_INST_H_
template class A<int>;     // A<int>
template class A<char*>;   // A<char*>

// foo<int>(const int&)
template void foo<int>(const int&);
// foo<char*>(char * const&)
template void foo<char*>(char * const&);
#endif
```

When the code is compiled the A_INST macro has to be set to "my_inst.h" and external template generation must be specified. With gcc this can be achieved by

```
$g++ -c A.cc -fexternal-templates -DA_INST=\"my_inst.h\"
```

If you are using a different compiler, check what flags you need for external template generation and how you can set macros at compile time.
"my_inst.h" provides the instantiations needed to generate descriptors for A<int>, A<char*>, foo<int> and foo<char*>. This file is being included in A.cc at compile time so the resulting object file A.o will have the typedescriptors for these types and other files can use these types as well with no multiple definitions.
The #pragma directive is for compiler specific flags. gcc requires interface defined for the class and function prototypes and implementation specified for their definitions. Compilers usually ignore the #pragma directive if it is not recognized. Check your compiler manual how to compile programs with templates and what methods it suggests.
Observe that if T is a pointer to a type A then const T after instantiation becomes A * const (*a constant pointer*) rather than const A * (*a pointer to a constant*). The reason is that T is the object in question, which in this case is the pointer rather than what it is actually pointing to.
Finally, let's see a function with a template – or a *parametric algorithm*. The well known quick_sort algorithm orders an array of comparable elements[4].
Our implementation has the prototype void quick_sort<T>(T* A, int l, int h, bool (*f)(const T&, const T&)). A is an array of elements of type T. l is the index

[4]extensively described in T. H. Cormen, C. E. Leiserson and R. L. Rivest: *Introduction to Algorithms*, MIT Press, 1989

of the first element where sorting should start and `h` is the index of the last element where sorting should end. `f` is a function which returns true if and only if the first parameter is *less* then the second parameter. `f` must be *strict* – if it implements $\leq$ instead of $<$, the algorithm may not terminate.

```
// prototype
template<class T>
void quick_sort(T*,int,int,bool (*)(const T&, const T&));


template<class T>
void quick_sort(T* array, int l, int h, bool (*f)(const T&, const T&)) {
   int i = l-1, j = h+1;

   if (l >= h) return;

   T cutoff = array[l];

   while(true) {
      do {
         --j;
      } while(f(cutoff,array[j]));

      do {
         ++i;
      } while(f(array[i],cutoff));

      if (i<j) {
         T tmp = array[i];
         array[i] = array[j];
         array[j] = tmp;
      } else break;
   }

   quick_sort<T>(array,l,j,f);
   quick_sort<T>(array,j+1,h,f);
}
```

Now to actually see `quick_sort<T>` in action, we write a program which takes parameters from the command line and first it sorts them by alphabetical order and then by order of string length.

The only thing we have to do is to implement different versions of `f` for different types and different semantic comparisons.

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

// return true iff length(s1) < length(s2)
bool less_length(char* const& s1, char * const& s2) {
   return strlen(s1) < strlen(s2);
}

// return true iff s1 comes before s2 in a dictionary
bool less_alphabetic(char * const& s1, char * const& s2) {
   return strcmp(s1,s2) < 0;
}

// function prints argv array
void print_args(int argc, char* argv[]) {
   for(int i=1; i< argc; i++) {
      cout << argv[i] << ' ';
   }
   cout << endl;
}

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

   // sort by alphabetical order
   quick_sort<char*>(argv,1,argc-1,less_alphabetic);
   cout << "alphabetic ordering: ";
   print_args(argc,argv);

   // sort by string length
   quick_sort<char*>(argv,1,argc-1,less_length);
   cout << "ordering by length: ";
   print_args(argc,argv);

   return 0;
}
```

Running the program at the command line:

```
$sort abrakadabra hello zebra c++ love

  alphabetic ordering: abrakadabra c++ hello love zebra
  ordering by length: c++ love hello zebra abrakadabra
```

Finally a little bit of syntax to demonstrate how to use pointers to member functions to parametric (*or template*) classes.

```
template<class T>
class C {
  ...
  public:
      ...
      int foo() const;
};

...

C<int> c1;
C<char*>* c2 = new C<char*>;

int (C<int>::* p1)() const = &(C<int>::foo);
int (C<char*>::* p2)() const = &(C<char*>::foo);

(c1.*p1)();
(c2->*p2)();
```

`C` is a parametric class. `c1` is an instance of `C<int>`, `c2` is a pointer to an instance of `C<char*>`. `p1` is a pointer to a method of `C<int>` which returns an `int` and takes no parameters and is set to `C<int>::foo`. `p2` is a pointer of the same kind of method but it belongs to `C<char*>` and is assigned `C<char*>::foo`. The last two lines – as expected – are the method invocations of the methods designated by the pointers on the instances. to the methods

7.6 Exercises

7.1 Read up on the Standard Template Library (*see references on page 234*) and explain in your own words how iterators abstract data structures. Explain why a programmer would use only iterator operations rather than native operations of the data structure (*like push or pop for stacks*). Describe an example!

7.2 Write an application which uses at least two of the container classes of the Standard Template Library and its iterators! Consult the references on page 234 and your compiler's manual!

7.3 Implement our definition of `list`. Feel free to add `private` and `protected` methods and instance variables to the definition.

7.4 Implement `list` (*like the previous exercise*) but make `list_node` an internal class of `list` and derive `list_iterator` from a generic iterator.

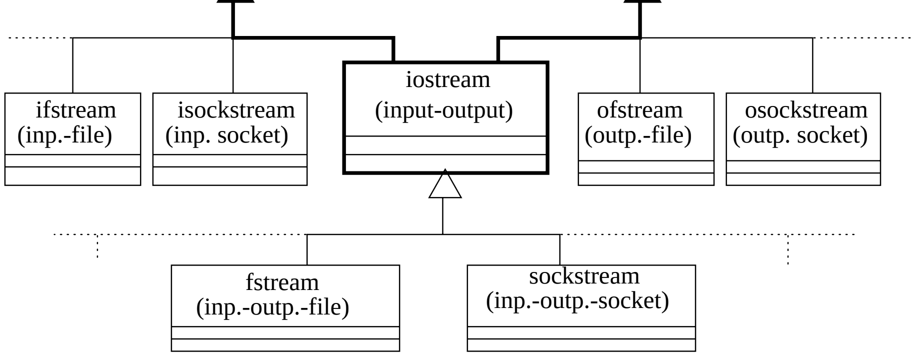
7.5 Subclass `list` (*see previous exercise*) and implement `sortable_list` which has an additional operation `sort`. You will need to supply a comparison function (*preferably passed to the constructor as an argument*) that compares two values of the parametric type `T`.

Chapter 8

Streams

8.1 Streams

Streams abstract low level input-output (*I/O*). Output can be thought of as a text or binary representation of an *object* which can be used to recreate the object or it can also be a raw stream of data. The process of creating a representation of an object which can be used to recreate it is also called *serialization*. Input is recreating objects from their text or binary representations or it can also be accepting a stream of raw data. The stream abstraction of an I/O channel or I/O port allows the programmer to treat the underlying device (*file, keyboard, screen, network socket, ...*) uniformly, use the same syntax and even the same code.



Every stream is a subclass of either `istream` or `ostream` or both (`iostream`). All `istream`s understand `operator`$\gg$ and all `ostream`s understand `operator`$\ll$. The programmer may freely implement an overloaded version of these operators which takes an argument of the specific class. `C++` provides implementations of `operator`$\gg$ and `operator`$\ll$ for primitive types (`char, char*, int, long, short, double, ...`) but the programmer is responsible to overload the operator for user defined types and classes. Supposing that the user has implemented `operator`$\ll$ and `operator`$\gg$ which can serialize and recreate an instance of class `C`, the following code demonstrates the advantages of stream I/O:

```
C c1,c2; // instances of c
ofstream ofs("/home/usr/santaclaus/output.txt");
         // ofs is an "output file"
ifstream ifs("/home/usr/cinderella/input.txt");
         // ifs is an "input file"

cin >> c1;  // using the implementation of operator>>
            // for objects of class C for initializing
            // c1 from the keyboard

ifs >> c2;  // using the VERY SAME implementation of operator>>
            // for objects of class C for initializing
            // c2 from a file

cout << c2; // using the implementation of operator<<
            // for objects of class C for serializing
            // c2 to the screen

ofs << c1;  // using the VERY SAME implementation of operator<<
            // for objects of class C for serializing
            // c1 to a file
```

The power of streams stems from the fact that serialization and initialization from a serialized form of an instance are independent of the type (*file, pipe, socket, ...*) and persistence (*temporary or permanent*) of the particular medium. Serialization is an operation which has many implementations (`operator`$\ll$) and so is initialization of an instance from its serialized form provided by multiple implementations of `operator`$\gg$.

From the programmer's point of view, there is a stream on the left-hand side and there is an object on the right-hand side.

$$\textbf{ostream} \ll \textbf{object}$$
$$\textbf{istream} \gg \textbf{object}$$

Because of inheritance of streams and polymorphism on the input and output operators the I/O operation (*i.e. the operator*) and the actual physical device represented by the stream are separate and hence the implementation of streams **and** the I/O operators are reusable because of this context independence. This elegant mechanism is much more powerful than the one used in the C language (*we cover that later in this chapter*). Unfortunately programmers often "prefer" the old way. It however must be understood that the C++ I/O mechanism is capable performing *everything* that the C mechanism can **plus** it provides reusability and context independence. The separation of the stream and the I/O device is also often referred to as I/O abstraction and *abstract I/O* means that the I/O operator is implemented for an abstract device (`istream` and `ostream`) whose identity and particulars do not have to be known.

8.2 File and String-Streams

File Streams

Instances of `ifstream` can read data from a file while instances of `ofstream` can write data to a file. Instances of `fstream` can do both reading and writing. All file streams have `open` and `close` methods.

Constructors & Basic Methods

`ifstream()`
`ifstream(`*int fd*`)`
`ifstream(`*const char* fname, int mode = ios::in, int prot = 644*`)`

`ofstream()`
`ofstream(`*int fd*`)`
`ofstream(`*const char* fname, int mode = ios::out, int prot = 644*`)`

`open(`*const char* fname, int mode = { ios::in or ios::out }, int prot = 644*`)`
`close()`

```
ifstream ifs;
ifs.open("/home/usr/santa/christmas.txt");
ofstream ofs;
ofs.open("/home/usr/santa/presents.txt");
```

`ifs` created by the constructor is an input file stream associated with no actual input file. Streams created this way must call the `open` method before reading can take place. Similarly `ofs` is an output file stream, initially not associated.

```
int ifd = open("/home/usr/santa/christmas.txt",O_RDONLY);
```

```
if (ifd == -1) { .. // something is wrong}
ifstream ifs(ifd);
int ofd = open("/home/usr/santa/presents.txt",O_WRONLY);
if (ofd == -1) { .. // something is wrong}
ofstream ofs(ofd);
```

> `ifs` is an input file stream associated with file opened by the `open` system call and file descriptor `ifd`. `ofs` is an output file stream associated with file with descriptor `ofd`.

```
ifstream ifs("/home/usr/santa/christmas.txt");
ofstrean ofs("/home/usr/santa/presents.txt");
```

> `ifs` is an input file stream associated with the file `christmas.txt` in directory `/home/usr/santa` and `ofs` is an output file stream associated with the file `presents.txt` in directory `/home/usr/santa`.

C++ 's `ios` class provides the following *static* constants for `mode`:

`ios::in` Open for input.

`ios::out` Open for output.

`ios::ate` Set the initial input (or output) position to the end of file.

`ios::app` Seek to the end of file before *each* write.

`ios::trunc` Always create a new file.

`ios::nocreate` If the file does not exist fail.

`ios::noreplace` Create a new file, but fail if the file already exists.

`ios::bin` Open in binary mode.

These values can be combined using the bitwise | operator: `ios::out | ios::create`. File streams opened with the third form of the constructor (*const char* fname,int mode, int prot*) are automatically closed when the object leaves scope (*by the destructor*). In general however, the programmer should call the `close` method explicitly before the file stream instance leaves its scope. Stream instances should always be passed or returned by reference (*why??*).

String Streams

String streams can be used to serialize objects to strings (`char*`) or to initialize objects from strings. These are specially handy, because unlike files, strings exist in memory. Hence applications can *serialize* to and *initialize* from memory directly without the overhead of file I/O. Input string streams are instances of `istrstream`, output streams are instances of `ostrstream` and instances of *strstream* can do both input and output.

Constructors

`istrstream()`
`istrstream(const char* str)`
`istrstream(const char* str, int size)`
`ostrstream()`
`ostrstream(const char* str)`
`ostrstream(const char* str, int size)`

If the size of the associated buffer is not explicitly specified, it is assumed to terminate with a null byte (`(char)0`).

```
const int buf_size = 100;
char* buffer = new char[buf_size];
buffer[buf_size-1] = (char)0; // null byte
istrstream is(buffer,buf_size);
ostrstream os(buffer,buf_size);

os << "hello world 2 3.14159" ;

char string[10];
int i;
double d;

is >> string; // string contains "hello"
is >> string; // string contains "world"
is >> i;      // i = 2
is >> d;      // d = 3.14159
```

The compiler has no way to determine the type (*int, char*, ...*) of the data to be read next from a stream, it is the programmers responsibility to make the appropriate assumptions. Fail safe programs always read streams a byte at a time and do their own parsing.

8.3 Using Streams and Manipulators

C++ by default provides the following instances, opened implicitly:

`istream cin;`

> The standard input stream; usually the keyboard.

`ostream cout;`

> The standard output stream; usually the display screen.

`ostream cerr;`

> The standard error stream; usually the display screen.

All these instances are variables (*as opposed to constants*), so they can be re-assigned.

Checking the state of a stream

The state of a stream can be checked by evaluating the pointer itself. For example:

```
if (cin) {
  cin >> ...
}
...
```

For more detailed diagnostics, the following instance methods of `ios` provide more information:

> `f` is an instance of `ios` (could be either `istream` or `ostream`)
>
> `f.good()` is true if no error indicators are set for `f`.
>
> `f.bad()` is true if the stream is not usable.
>
> `f.eof()` is true if the stream has reached the *end of file* (`EOF`).
>
> `f.fail()` is true if any of the error indicators is set.
>
> The method `clear` (`f.clear()`) can be used to *reset* the state of the stream.

Reading a stream

All instances of `istream` can read a single character, using any of the `get` methods:

> `ifs` is an instance of `istream`, `c` is a `char` and `i` is an `int`.
>
> `i = ifs.get();` reads one character from `ifs` using the `int istream::get()` method . If the end of file is reached `i == -1`. `c = (char)i;` is the character read otherwise.

`ifs.get(c);` reads one character from `ifs` using the `istream& istream::get( char&)` method.

`i = ifs.peek();` returns the next available input character (*or EOF*) without changing the state of `ifs` using the `int istream::peek()` method.

An entire line can be read using the `istream& istream:get( char* buffer, int len, char delim='\n')` method:

```
char buffer[100];

f.get(buffer,100);
```

reads at most 100-1 = 99 characters or up to the first '\n' character. The trailing null character is automatically inserted.

The `istream& istream::get(char*,int,char='\n')` method skips *white spaces* and is intended for ASCII character input. Raw bytes can also be read into a buffer without interpretation using the `istream& istream::read( void*, int len)` method. This method attempts to read `len` many bytes from the input stream.

```
char* buffer = new char[100];

if (f.read(buffer,100)) {
    // then we have 100 bytes in the buffer
    ...
}
```

The last character can safely be put back onto the stream using the `istream& istream::putback(char c)` method. This method is very handy for implementing parsers, and in fact, the only method that is guaranteed to backtrack on an input stream without changing its state.

```
...
// read characters up to but not including the first ';'
char buffer[100],c = f.get();
int i=0;
while(c != -1 && c != ';' && i < 100) {
  c = f.get();
  i++;
}
if (c==';') f.putback(c);
...
```

Writing to a stream

All instances of `ostream` can be written to. Object serialization is done using `operator`≪. For binary output, the following two methods can be used.

The `ostream& ostream::put(char)` method can be used to write a single byte to an output stream.

```
...
// ofs is an ostream
ofs.put((char)0x3); // EOT (or ^C) character
...
```

The `ostream& ostream::write(char* buffer,int len)` method can be used to write `len` many characters from `buffer` uninterpreted.

```
...
buffer char[] = {'A','T','D','T',' ','9','1','1'};
ofs.write(buffer,strlen(buffer)); // call 911
...
```

Output is most likely to be buffered by its nature. This means that characters may not appear on the output device (*screen, pipe, ...*) at the same time when the program executed the output operation. When this synchronization is necessary the `ostream& ostream::flush()` method can be used.

```
...
cout << "Please enter .... > "; // no '\n' or endl
cout.flush();
cin >> ...
```

Changing Stream Properties

Streams (*both input and output*) possess properties and attributes that define its actual state. The ones we consider here are formatting properties.

The `int ios::precision() const` method reports how many significant digits are used for output. By default, this number is 6. The `int ios::precision(int sigdig)` method returns the previous settings of significant digits and sets `sigdig` as the number of significant digits in use. The `int ios::width() const` method reports the current field width setting. This value is 0 by default, meaning that use exactly as many characters as needed. The `int ios::width(int num)` method returns the previous setting of the field with property and sets the field width to `num` many characters. The `char ios::fill() const` method returns the current setting of the padding character. By default it is ' '. The `char ios::fill(char padding)` method returns the previous setting and sets the padding character to `padding`. The `fmtflags ios::flags() const` method reports the current value of the complete collection of flags of the stream state. The method `fmtflags ios::setf(fmtflags flag)` method sets one particular flag, while the method `fmtflags ios::unsetf(fmtflags flag)` turns off flag `flag`. Both methods return the entire set of flag settings. The following chart describes the `ios` flags:

flag	**meaning**
`ios::dec`	decimal numeric base
`ios::hex`	hexadecimal numeric base
`ios::oct`	octal numeric base
`ios::fixed`	do not use scientific notation
`ios::left`	left justify
`ios::right`	right justify
`ios::internal`	middle justify
`ios::scientific`	use scientific notation
`ios::showbase`	prefix octal with `0` and hex with `0x`
`ios::showpoint`	trail even numbers with `.000...`
`ios::showpos`	display positive sign
`ios::skipws`	skip white space (default)

Most often, these properties are to be set for each output operation. Manipulators provide a convenient way of changing the state of a stream in the middle of expressions chained by the $\ll$ (or $\gg$) operators. The scope of a manipulator where it affects the state of the stream is only the "next" I/O operation. The following operators are provided for instances of `ios`:

manipulator	**meaning**
`ws`	skip white space
`flush`	write pending output *now*
`endl`	write a '\n' character and flush
`ends`	write a `(char)0` character
`setprecision(int n)`	set number of significant digits to n
`setw(int n)`	set the width parameter to `n` characters
`setbase(int b)`	change numeric base to `b`
`dec`	same as `setbase(10)`
`oct`	same as `setbase(8)`
`hex`	same as `setbase(16)`
`setfill(char c)`	set the padding character to `c`

Example:

```
   cout << setw(5) << setfill('$') << hex << 255 << endl;

   // $$$ff
```

8.4 Overloading `operator≪` & `operator≫`

`operator≪` is a method of `ostream` and `operator≫` is a method of `istream`, hence they have to be overridden as *free functions*. In that form for a class `C` the prototypes are as follows:

```
   ostream& operator<<(ostream&,const C&); // or
   ostream& operator<<(ostream&,C);

   istream& operator>>(istream&,C&);
```

Often these two operators are declared as friends in `C`'s class definition so instance variables can be accessed without method invocation. `operator≪` is used to *serialize* an instance of class `C` and `operator≫` should be able to recreate the same logical instance from this form. For example:

```
class student {

  private:
    char* lname, *fname;

    long snumber;
  public:

    .....
  friend istream& operator>>(istream&,student&);
  friend ostream& operator<<(istream&,const student&);
};

ostream& operator<<(ostream& os,const student& s) {
   os << s.fname << ' ' << s.lname << ' ' << s.snumber ;
   return os;
}
```

```
istream& operator>>(istream& is,student& s) {
   static char buffer[256];

   is >> buffer;
   s.fname = new char[strlen(buffer)+1];
   strcpy(s.fname,buffer);

   is >> buffer;
   s.lname = new char[strlen(buffer)+1];
   strcpy(s.lname,buffer);

   is >> s.snumber;

   return is;
}
```

8.5 C style I/0

The C language does not support streams or I/O abstraction but it does provide uniform and consistent I/O via the `printf` and `scanf` family of functions. These functions are declared in <`stdio.h`>.

Formatted output

The printf functions can take multiple arguments and format them as specified in the format string. The prototype of this function:

```
  int printf(char* format,...);
```

On success the function returns the number of characters successfully transmitted (*except the null byte*) and on failure it returns `EOF`. The three dots (*... or ellipsis*) are actually part of C and C++ syntax and specify that zero or more arguments follow. This facility of passing an arbitrary number of arguments should be used with care because there is no mechanism to do compile time type checking[1]. The format string can contain arbitrary text but the % character up to and including a conversion character is used to interpret an argument passed to `printf` for printing (*the % character can be "escaped" to prevent interpretation by* `printf`*: %%*).

[1]a variable argument list can be interpreted using a `va_list` variable, which can be stepped using `va_arg`. The programmer is responsible to determine or to *know* the type of the argument. *see the appendices on page 224*

Conversion characters

character	**interpretation**
d,i	decimal integer
o	unsigned octal number
x,X	unsigned hexadecimal number
u	unsigned decimal number
c	single character
s	character string
f	`double` or `float`
e,E	`double` or `float` with scientific notation
p	pointer (*print as address*)

Between the % sign and a conversion character, the following characters – in this very order – may appear. Omitting any of these forces the default to be used.

- A `-` sign to force left adjustment. (*no* `-` *is the default*)
- A decimal number that specifies the minimum field with. (*the default is to use as many as needed.)*
- A period (.), which separates the field width from the precision.
- A decimal number indicating the precision (*or the number of digits printed after the decimal point*) for floating point numbers. For character strings, it is the maximum number of characters to be
- An `h` character for `short` integers or an `l` character for `long`.

If an asterisk (*) is specified instead of the field width or the precision then these values can be passed as arguments.

Examples:

```
int i = 255;
float pi = 3.14159;
double  e = 2.718281828;
char *s = "hello world";

printf("|%s|\n",s);
printf("|%20s|\n",s);
printf("|%.5s|\n",s);
printf("|%-20.7s|\n",s);

printf("i = %d = %x \n",i,i);
```

```
 printf("e = %.2f = %e and pi = %.0f = %g\n",e,e,pi,pi);
```

The output of the program is:

```
|hello world|
|         hello world|
|hello|
|hello w             |
i = 255 = ff
e = 2.72 = 2.718282e+00 and pi = 3 = 3.14159
```

Formatted input

The `scanf` functions can be used to read the values of variables. The prototype of this function is:

```
   int scanf(char* format,...);
```

The character conversion table is analogous to the of `printf`'s[2]. To read `double`s one must specify `%lf` as opposed to `%f`. If non format characters appear in the format string they must exactly match that of the input. The arguments are the *addresses* of the variables. It is a very common and often fatal mistake to use the variable instead of the address! `scanf` returns the number of arguments properly converted and initialized or `EOF` if the end of file has been reached.

Examples:

```
int i;
double d;
int year,month,day;
char buffer[100];

scanf("%d",&i); /* read one integer */
scanf("%f",&d); /* read one double */
if (scanf("%d/%d/%d",&year,&month,&day) != 3) {
  printf("year/month/day expected\n");
}
```

[2]except for minor deviations.

```
scanf("%s",buffer); /* read a character string */
```

Reading and writing files

Files in C can be read and written using `fscanf` and `fprintf`. They are analogous to `scanf` and `printf` with the exception that they take a file pointer as their first argument. A file pointer in C represents a file and can be obtained by the `fopen` library call. The same file can be closed using the `fclose` function and flushed using the `fflush` function.

```
   FILE* fopen(char* filename,char* mode);

   int fscanf(FILE* fp,char* format,...);
   int fprintf(FILE* fp,char* format,...);

   int flcose(FILE* fp);
   int fflush(FILE* fp);
```

`mode` is one of the following:

mode	**meaning**
`"r"`	open for reading
`"w"`	open for writing
`"a"`	open for “appending”

To read or write a binary file the character `b` should be appended to `mode` (*like: "rb" for read binary*). `fflush` and `fclose` return 0 on success and `EOF` on error. `fopen` returns `NULL` if the file could not be opened.

```
FILE *fin = fopen("/usr/home/santa/christmas.txt","r"); /* for reading */
FILE *fout = fopen("/usr/home/santa/donelist.txt","w"); /* for writing */

if (fin == NULL or fout == NULL)  {
  /* something is wrong */
  ...
}

char kid[100],present[256];
fscanf(fin,"%s %s",kid,present);
/* santa buys present for kid */
fprintf(fout,"%s %s\n",kid,present);
```

Reading and writing character strings

The functions `sprintf` and `sscanf` can be used to write to and read from character strings.

```
   int sprintf(char* buffer,char* format,...);
   int sscanf(char* buffer,char* format,...);
```

They are identical to `printf` and `scanf` in all respects, except they take a character string as the first a argument.

Example:

```
char buffer[1024];
int i;

sprintf(buffer,"hello %s you are %d years old!\n","monica",20);

printf("%s",buffer); /* hello monica you are 20 years old! */

sprintf(buffer,"%d",200);
sscanf(buffer,"%d",&i);  /* i = 200 now */
```

Predefined file pointers

`stdin`	standard input, analogous to `cin`
`stdout`	standard output, analogous to `cout`
`stderr`	standard error, analogous to `cerr`

8.6 Notes on Syntax and Semantics

Stream abstraction – while not necessarily an object oriented idea – is a big contribution to programming in general. The most important thing to remember is that there are two parts to the abstraction. The stream on the left hand side and the object on the right hand side which understands the I/O operation. As far as the left hand side is a stream the object can be read from and written to network sockets, pipes, files, databases and virtually everything which can be abstracted by a stream. As far as only the stream changes the implementation of the I/O operation *does not need to change* for the object. If a new stream – *like a serial port* – is needed then only this stream has to be implemented. Similarly, if there is a new object to be serialized to a stream, only one version of the I/O operation must be implemented in terms of the *abstract stream* and of course, all subclasses will use their particular overloaded versions resolved run-time.

Unfortunately serialization also poses one of the biggest challenges in the object oriented paradigm. Consider the problem of instantiating objects from a stream. The problem is that all this power gained by polymorphism to resolve types run-time is lost. Suppose class `B` and class `C` are derived classes of class `A`. We would like to write a function or method which would return a new instance of type `A`. This new instance could of course be an instance of class `B` or class `C` because of inheritance. I cannot just implement an overloaded `read(istream&)` method, because until there is no instance there can be no resolution. The classical *chicken and egg* problem. One cannot be done without the other. Feel free to pause here and make sure you understand. There have been suggestions and even good solutions to the problem but non of them are really "object oriented". In fact while the object oriented paradigm is useful to implement parsers the underlying model is not really object oriented but rather procedural (*or even more often modeled by state machines*). Again, we have subclasses of a class and we would like to be able to implement a nice and automatic mechanism which can instantiate objects from a stream – even if they are actually instances of the subclasses.

The model I propose is almost fully automatic and should give some ideas. However before going any further you should understand what the issue is so try to think about it for a bit.

In our demo, we have a class `A` with two derived classes `B` and `C`. Instances of `A` which are actually either instances of `B` or instances of `C` are coming on a stream and we would like to write a method `A* A::read(istream&)` which returns a pointer to the instance read. Just to reiterate, the seemingly obvious solution to overload the method to implement `A* B::read(istream&)` and `A* C::read(istream&)` will not work, because I would need an actual instance to have `read` resolve to `B::read` or `C::read` but that instance is the very instance I am trying to read!

We will need a way to discriminate actual instances of `B` from instances of `C` on the stream. This will be achieved by a unique *magic number*. A *magic number* is a unique signature which identifies how a file should be interpreted. For example certain type of image files such as GIF, JPEG, MPEG, PPM and many more ... all start with a unique *magic number* which tells the image viewer how it should interpret the file. We use this very technique to identify instances of `B` by the signature `B_MAGIC` and instances of `C` by `C_MAGIC`. Then an obvious algorithm would be to read the magic number and if it is the former then return a new instance of `B` and if it is the latter then it would return an instance of class `C`. The problem with this method is that all the maintainability gained by polymorphism is lost. Suppose I would like to add a new subclass `D` derived from `A`. Well, I would add a new magic number `D_MAGIC` but I would have to modify an already existing method `A* A::read(istream&)` to return a new instance of `D` when its magic number is read. I am not supposed to modify existing code in the object oriented paradigm when I derive a new class! So what we do instead is to have an array of *readers* associated with magic numbers belonging to class `A`. And when a new subclass of `A` is derived, we just add a handler and a magic number by calling one class method of `A` and do not modify `A` at all.

```
class A_exception {
};

class A {
   protected:
      // A_reader is a function which
      // takes an istream by reference
      // and returns a new A object
      typedef A* (* A_reader)(istream&);

      static char** magic_numbers;
      static A_reader* readers;
      static int n;

   public:
      static A* read(istream&);
      static void add_reader(const char*,A_reader);
      static void destroy_readers();
      virtual void print(ostream&) const = 0;
};

class B:  public A {
   protected:
      int i;

   public:
      B(int);
      virtual void print(ostream&) const;
};

class C: public A {
   protected:
      char * s;

   public:
      C(const char*);
```

```
        virtual ~C();
        virtual void print(ostream&) const;
   };
```

`A_reader` is type which matches the prototypes of the handlers. Upon recognizing a magic number the corresponding handler takes over to instantiate the object. `magic_numbers` is an array of magic numbers and `readers` is an array of corresponding handlers. `n` is the number of handlers registered. The method `A* A::read(istream&)` is the one responsible to instantiate the objects. `add_reader(const char* magic_number, A_reader reader)` registers `reader` to `magic_number`. `destroy_readers` deallocates resources associated with the handlers.

```
   void A::add_reader(const char* magic_number, A_reader reader) {

      for(int i=0; i<n; i++) {
         if (strcmp(magic_number,magic_numbers[i]) == 0) {
            throw A_exception();
            // magic_number already exists, programmers error
         }
      }

      A_reader* new_readers = new A_reader[n+1];
      char** new_magic_numbers = new char*[n+1];
      new_readers[n] = reader;
      new_magic_numbers[n] = new char[strlen(magic_number)+1];
      strcpy(new_magic_numbers[n],magic_number);

      for(int i=0; i<n; i++) {
         new_readers[i] = readers[i];
         new_magic_numbers[i] = magic_numbers[i];
      }

      delete [] readers;
      delete [] magic_numbers;

      readers = new_readers;
      magic_numbers = new_magic_numbers;

      n++;
   }

   void A::destroy_readers() {
      for(int i=0; i<n; i++) {
         delete magic_numbers[i];
      }

      delete [] magic_numbers;
      delete [] readers;
      magic_numbers = 0;
      readers = 0;
      n = 0;
   }
```

add_reader throws an exception if magic_number is already registered, otherwise it extends the arrays by one new magic number and a corresponding handler reader. destroy_readers releases the arrays that store the magic numbers and the handlers.

```
   A::A_reader* A::readers = 0;
   int A::n = 0;
   char** A::magic_numbers = 0;

   A* A::read(istream& is) {
      char magic_number[128];
      is >> magic_number;

      for(int i=0; i<n; i++) {
         if (strcmp(magic_number,magic_numbers[i]) == 0) {
            return (readers[i])(is);
         }
      }

      throw A_exception();
      // No such magic_number
   }
```

The method read reads the magic number into a buffer and goes through the array of registered magic numbers and if found it invokes the handler to instantiate the object.

```
   A* B_reader(istream& is) {
      int a;
      is >> a;
      return new B(a);
   }
```

```
A* C_reader(istream& is) {
   char buffer[256];
   is >> buffer;
   return new C(buffer);
}
```

B_reader and C_reader are the two handlers that instantiate B and C objects respectively.
The rest of the B and C methods:

```
B::B(int a):i(a) {
}

void B::print(ostream& os) const {
   os << "B_MAGIC" << ' ' << i;
}


C::C(const char* c):s(new char[strlen(c)+1]) {
   strcpy(s,c);
}

C::~C() {
   delete [] s;
}

void C::print(ostream& os) const {
   os << "C_MAGIC" << ' ' << s;
}
```

We would like to use operator<< to print objects of type A. But operator<< belongs to ostream rather than to A, so how can we use polymorphism to run-time infer the dynamic type? This is why we had the virtual method print.

```
ostream& operator<<(ostream& os, const A& a) {
   a.print(os);

   return os;
}
```

The question is if we can employ the same trick to use operator>> to read instances of A. The answer is *of course not* since A is an abstract class and we would encounter the same problem that we were trying to avoid by using registered handlers. But we can use a *another* trick.

```
class A_WRAPPER {
   protected:
      A* ptr;
   public:
      A_WRAPPER();
      A* get_A();

   friend istream& operator>>(istream&,A_WRAPPER&);
};

A_WRAPPER::A_WRAPPER():ptr(0) {
}

A* A_WRAPPER::get_A() {
   return ptr;
}

istream& operator>>(istream& is, A_WRAPPER& w) {
   w.ptr = A::read(is);

   return is;
}
```

A_WRAPPER is a class which holds a pointer to an instance of A. Its only method is A* A_WRAPPER::get_A() which returns the address of the object held. Note that there is no destructor! Now we can read an A_WRAPPER using operator>> and get the actual A instance by calling the method.
And finally, the program:

```
int main(int argc, char* argv[]) {
   A::add_reader("B_MAGIC",B_reader);
   A::add_reader("C_MAGIC",C_reader);

   for(int i=1; i<argc; i++) {
      try {
         istrstream is(argv[i],strlen(argv[i]));
         A_WRAPPER w;
         is >> w;
         cout << *(w.get_A()) << endl;
      } catch (...) {
         cout << "something is wrong with: " << argv[i] << endl;
      }
```

```
    }

    A::destroy_readers();

    return 0;
  }
```

First we register the two handlers.
Then we just keep reading `A`'s by `operator>>` and writing them by `operator<<`. The most important to remember is that we can add a new derived class of `A` with a new handler and magic number and we would not have to modify `A`'s implementation what so ever! It is probably worth going through this example a few more times.

8.7 Exercises

8.1 In `Java` , there is an abstract `InputStream` class which implements a number of `read` methods. The `read` methods that read burst of characters into a buffer use the `read` implementation which reads a single character iteratively. In `C++` , it would be similar to the definitions below:

```
class InputStream {
 public:
  virtual int read() = 0; // abstract
  virtual void read(char*& b, int& n) {
   int c,i = 0;
   while((c = read()) != EOF) {
    b[i++] = (char)c;
    if (i==n) return;
   }
   n = i;
  }
};
class FileInputStream :  public InputStream {
 ...
};
class BufferedInputStream:  public InputStream {
 ...
};
```

Explain the advantages of using this method! (*Hint: how much work does one have to do to implement a new subclass of* `InputStream`*?*)

8.2 In Java , there is a subclass of `InputStream` (*see previous exercise*) `FilterInputStream` defined (C++ *style*) as follows:

```
class FilterInputStream:  public InputStream {
 protected:
  InputStream* in;
 public:
  FilterInputStream(InputStream *is):in(is) {
  }

  virtual int read() {
   return in->read();
  }

  virtual void read(char*& b, int& n) {
   in->read(b,n);
  }
};
```

Explain the use of such a class. Give examples!!!

8.3 Describe situations, when the programmer is required to

- ◦ Subclass `istream` or `ostream`
- ◦ Overload `operator`≫ or `operator`≪
- ◦ Subclass `istream` or `ostream` **and** overload `operator`≫ or `operator`≪

8.4 Implement `operator`≫ and `operator`≪ for `class matrix`, which is a class that manipulates $n \times m$ arrays of `doubles`. Also implement the same C style (`scanf`, `printf`). Is the C or C++ style of I/O is more appropriate?

@

Chapter 9

Object Oriented Design

9.1 Classes and Objects

What is a class?

A class is a set of *logically* similar entities. These entities may be similar in structure and have similar properties (*instance variables*) and they may be similar with respect to the *kind* of messages they respond to or with respect to the *kind* of operations they can perform (*instance methods*).

What is an object?

An object is *one particular* instance of a *class* whose properties and relationships are uniquely established (*instantiated*).

What is a class hierarchy?

Classes are arranged in a hierarchy (*tree*) according to the *is a* pre-order. A *derived class is a base class* (*the derived class is also called the subclass and the base class is also called the superclass*). It is similar to the *set-superset* relationship in the sense that instances of the derived class are instances of the base class. In other words, the *is a* relationship is transitive.

What is an instance variable?

An instance variable usually represents a property of the class. For example, `student_number` is a property of class `student`. An instance variable can also represent an *association*. For example, the instance variable `registered_in` may represent an association between a `student` and a `course`.

What is an instance method?

Instance methods provide the interface to the object. It may be used to set or report a property or a request to perform an operation. Instance methods belong to the instance and they can only be called with an instance. The actual invocation of a particular method with an instance is also referred to as a *message* sent to the object. For example, if `mary` is an instance of `student`, then the instance method `mary.getStudentNumber()` reports a property, while the instance method `mary.registerIn(csi2172A)` performs the operation *registerIn*.

What is a class variable?

A class variable represents a *common* property of all instances of the class. All instances share this variable (*there is one physical location*), and the variable can be changed with or without an instance. For example, a class variable may be used to count the number of instances allocated, or represent a shared resource for all instances.

What is a class method?

A class method may be used to report or change a property represented by a class variable, or it may also be used to perform an operation which is related to the class but requires no actual instance. For example, the `math` class may have a class method `sin` to perform the trigonometric function without having an instance of `math` allocated (`math::sin(x)`). In some object oriented languages, – like SmallTalk –, instance creation (*the* `new` *operator*) is a class method.

How does the object oriented paradigm differ from the procedural paradigm?

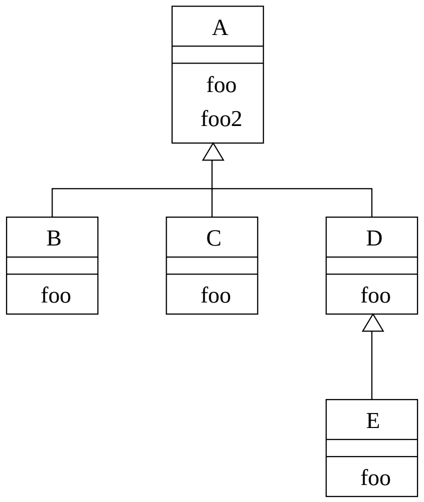
The main difference is the basis of problem decomposition. In the procedural paradigm, problem decomposition consists of identifying the *tasks* that the application must perform. These task are in turn refined to *subtasks*. Good decomposition minimizes *coupling* (*the inter-dependence of tasks*) and maximizes *coherence* (*each module has well defined and clear purpose independent of the context it is used*). However, it has been observed, that the structural make up of an application is more persistent than the functionality it must perform. In the object oriented paradigm, decomposition consists of identifying the building blocks (*or entities*) of the enterprise and their relationships (*associations*). Good decomposition, – as in the procedural paradigm –, minimizes coupling (*the topology of associations*) and maximizes coherence (*each object is well defined, reusable and context independent*).

9.2 Polymorphism: Operation vs. Methods

An *operation* is a function or transformation that has well understood semantics and is more-less context transparent. For example, *addition* of numeric values or drawing a shape on a window are operations. A *method* is a particular implementation of an *operation* for a class. For example, addition of *fractions* or *complex numbers* are methods that implement the abstract operation *addition* and the draw methods of *circle* and *triangle* are implementations of the operation *draw*.

The rationale behind using polymorphism is the mechanism provided by object oriented languages which ensures invocation of the appropriate implementation (*method*) of the operation, even if this decision has to be made run-time. Using and exploiting this facility, program logic can be greatly simplified and the application is more maintainable and extendable.

Consider the following object model. `A, B, C, D` and `E` are classes which all provide a method implementation of the `foo` operation. Furthermore, class `A`'s `foo2` operation calls `A`'s `foo` method.



How does polymorphism simplify program logic?

> Every instance of a derived class is also an instance of the base class. Hence instances of classes `B, C, D` and `E` are also instances of class `A`. A variable declared to hold an instance of class `A` can actually hold instances of the subclasses of `A`. `C++` has no way of determining if a statically allocated instance of `A` is also an instance of a subclass of `A`[1]. On the other hand,

[1]Most object oriented languages, – like Java and SmallTalk –, do not allow the programmer to allocate instances statically. Instances in these languages must be allocated by the `new` operator (*or class method*)

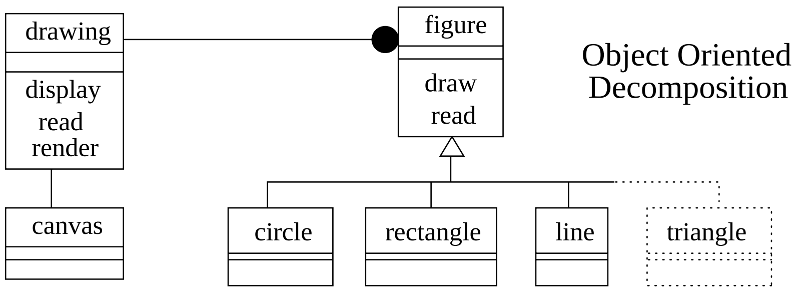
C++ pointers and references are dynamically (*run time*) de-referenced, so the programmer can rely on the appropriate version of the operation to be invoked.

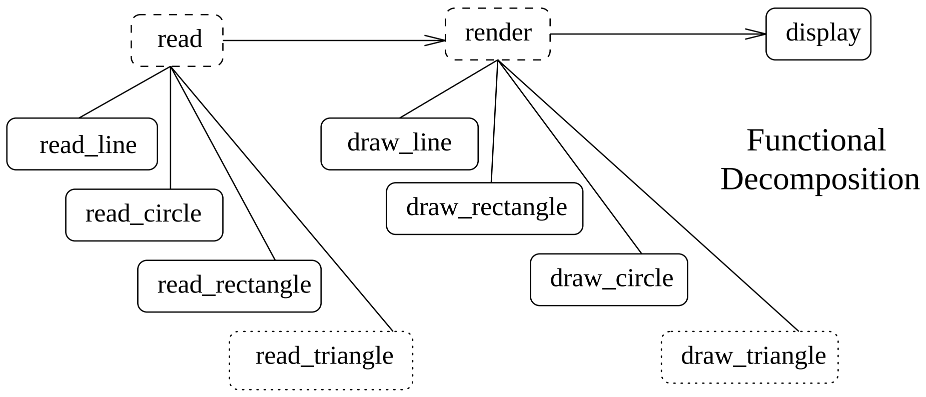
```
A *p1,*p2;

p1 = new B;
p2 = new E;

p1->foo(); // B::foo
p2->foo(); // E::foo
```

Because this mechanism is part of the language, the programmer does not have to implement selection of the appropriate method. Any class which is associated to an instance of class A or a container that holds instances of class A can exploit polymorphism to perform abstract *operations* on a variety of instances.

How does polymorphism make a program more extendable?

Consider an application, where the objective is to draw figures read from a file on the screen. Suppose the application supports drawing lines, rectangles and circles.





The figure above shows straight forward object oriented and functional decompositions. `read` reads the figures from a file, `render` draws each figure on the `canvas` and `display` displays the canvas in a window.

Suppose we would like to add the ability to draw triangles as well. Following our object model, the programmer only needs to create one new object and implement its `draw` and `read`[2] methods. Following our functional model, the programmer not only has to implement functions to read and draw a triangle, but he also must change functions `render` and `read`. Let us take a closer look why:

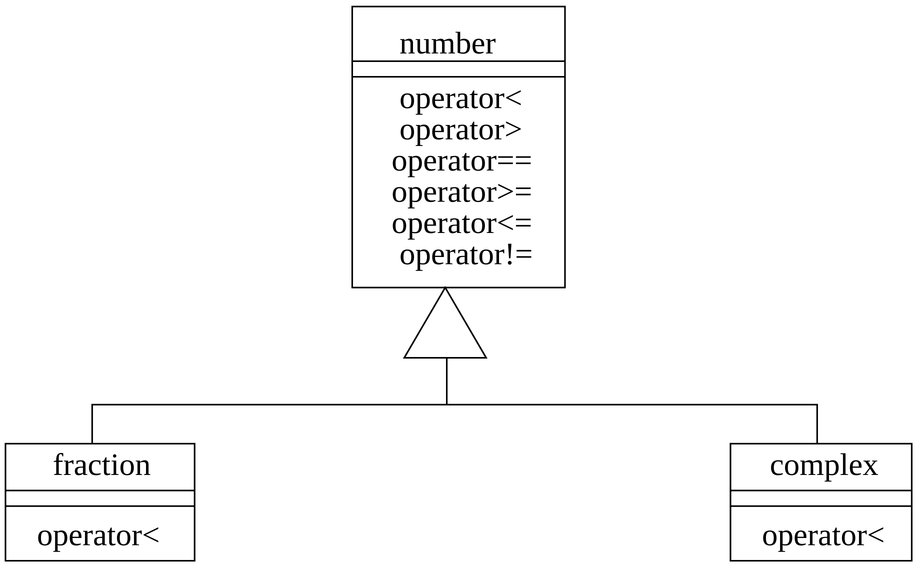
```
void render (canvas* c) {

   for(int i=0;i<nof_figures;i++) {
      switch(figures[i].type) {
        case 1: draw_line(c,figures[i]);
                break;
        case 2: draw_rectangle(c,figures[i]);
                break;
        case 3: draw_circle(c,figures[i]);
                break;
        case 4: draw_triangle(c,figures[i]);  // must be
                break;                        // added
        default:
                panic("unknown figure");
                break;
      }
   }
}

void drawing::render(canvas& c) {
   for(int i=0;i<figures.length();i++) figures[i]->draw(c);
   // polymorphism decides which version of draw!
}
```

To extend a program incrementally without even partially rewriting already existing code requires a design in terms of abstract classes and operations. In this case the class `figure` is abstract because it can only be instantiated as one of its subclasses. The operation `draw` is also abstract, because it must be implemented in the subclasses.

Another way to exploit polymorphism is to implement an operation in terms of some

[2]reading instances of subclasses is bit tricky, see the example in *Notes on Syntax and Semantics* for the *Streams* chapter

other well defined operations. In the first example (*page 199*) , we said, that `A::foo2` is implemented using `A::foo`. As all methods inherit, classes `B, C, D` and `E` also have a `foo2` method. `B::foo2` automatically calls `B::foo` (*as opposed to* `A::foo`). The power of this mechanism is illustrated in the next example:



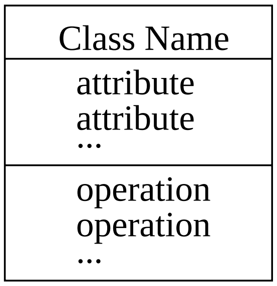
```
bool number::operator>=(const number& n) {
   return  !(*this < n);
}

bool number::operator==(const number& n) {
   return !(*this < n) && !(n < *this);
}

bool number::operator<=(const number& n) {
   return  !(n < *this);
}

bool number::operator>(const number& n) {
   return !(*this < n) && (n < *this);
}

bool number::operator!=(const number& n) {
   return (*this<n) || (n < *this);
}
```

Now the programmer only has to implement `complex::operator<` and `fraction::operator<`. The rest of the comparison operators for `fraction` and `complex` are “already’ implemented.

9.3 The Object Model

Programming in many respects is very similar to modeling. A model is a representation of a real or fictitious enterprise which highlights those aspects that are of particular interest. An object model is a representation which consists of the entities that make up the enterprise and their associations. The OMT (*Object Modeling Technique*) formalism was developed by *James Rumbaugh* and is very popular in both industry and academia[3].

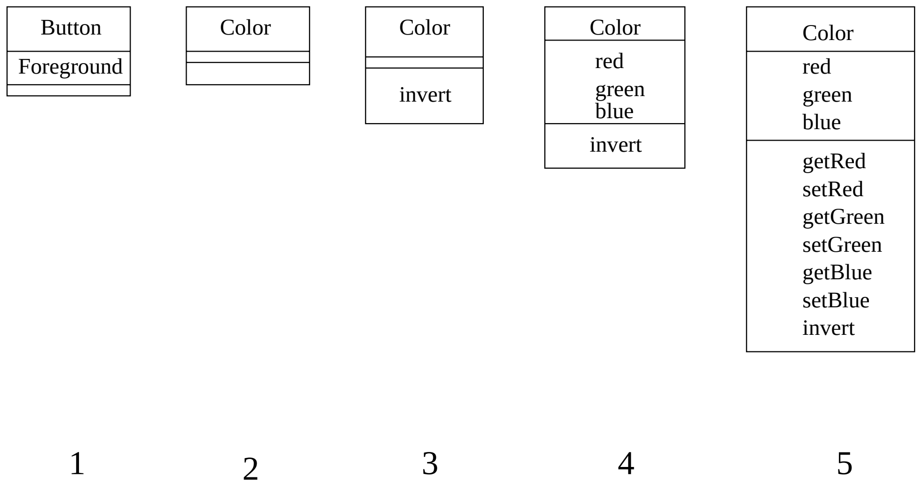
The Class



The attributes are most often instance variables and the operations are methods. An attribute represents a property of an instance of the class. Even though methods may be implemented to set or get the value of the attribute, there is no need to list these methods as operations. Operations are those methods that fundamentally constitute to the functionality of the class and implement the interface to interact with the object. It is very important to hide implementation details. The object model should be explicit that a programmer can use it to implement the application, but it should leave flexibility for the developer to make low level implementation decisions including the language of choice. An attribute is only worth listing if it would be included in a general description of the entity. Only methods that are *public*[4] and descriptive with respect to the class' purpose should be mentioned in the object model.

[3]the other popular formalism was developed by *G. Booch*

[4]if the programming language allows to restrict access privileges on methods



1. `Foreground` is an instance variable of type `Color`. It is a *property* of the user interface component `Button`. The structure of the color class is insignificant and it can be treated as if it were a native type.

2. `Color` is a class with no particular context interpretation. `Color` also plays a role in the application, which is more than a property of other classes . This representation is sufficient for most object models.

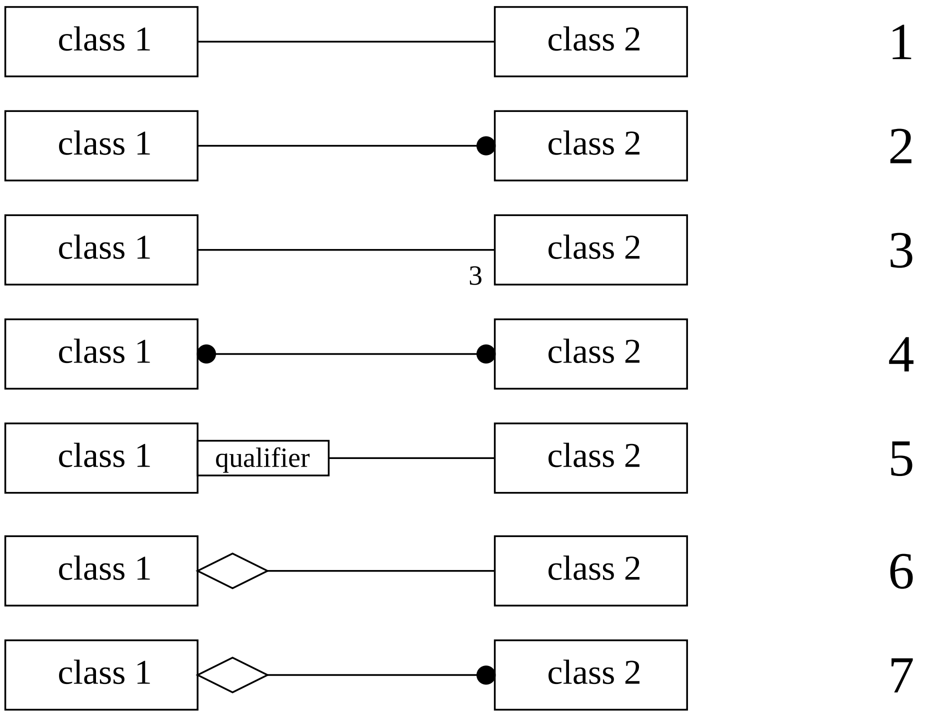
3. An instance of `Color` is equipped with the operation `invert` which can create the *negative* of the color. If the invert operation is fundamental for the application, this is the right representation.

4. An instance of `Color` is implemented using the RGB (*red, green, blue*) color encoding scheme. This representation may be too detailed unless it is fundamental or at least advisable for the application that colors are represented this way. This representation goes beyond design and enforces an implementation decision. For example, it prohibits the developer to use the HVS (*hue, value, saturation*) encoding. This level of detail *may be* appropriate and is likely to be included in the final and most refined versions of the object model.

5. This representation is *redundant*, *unnecessary* and *obsessive*. This level of detail is not desirable and only adds confusion to the design.

The Association

Associations represent the relationship between objects. OMT distinguishes the following association types:



1. `Class 1` and `Class 2` are related. The association is also *one-one* indicating that each instance of `Class 1` is related to exactly one instance of `Class 2`. Example: `King` – `Kingdom`.

2. Each instance of `Class 1` is related to *zero or more* instances of `Class 2`. This association is a `one-many` relationship. Example: `Country` –● `City`.

3. Each instance of `Class 1` is related to exactly 3 instances of `Class 2`. Example: `ThreeLeggedMonster` – 3 `MonsterLeg`.

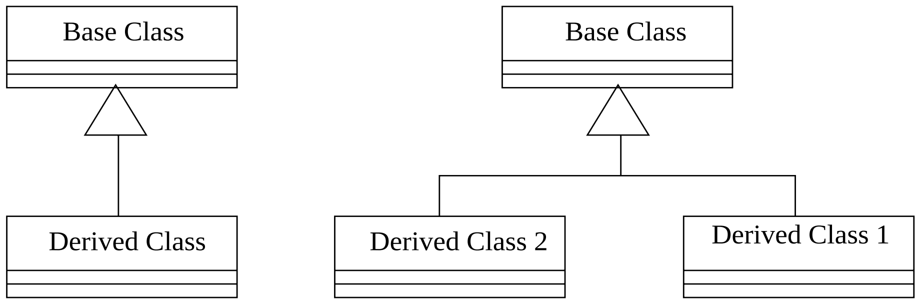
4. Each instance of `Class 1` is related to many instances of `Class 2` *and* each instance of `Class 2` is associated with many instances of `Class 1`. This association is a *many-many* relationship. Example: `Student` ●–● `Course`. Each student is registered in zero or more courses, while each course has many students.

5. Instances of `Class 2` are represented by a unique qualifier in instances of `Class 1`. A *qualified* association is implicitly *one-many*. Example: `University` [`student_number`] – `Student`.

6. Instances of `Class 1` *include* one instance of `Class 2`. This type of association is called *aggregation*. The main difference between an *aggregation* and a regular *association* is that `Class 2` plays a *passive* role. Example: `Monitor` ◇– `Screen`.

7. Instances of `Class 1` *include* many instances of `Class 2`. Example: `Directory` ◇–● `File`.

Inheritance



9.4 Implementing Associations

The particulars of how an association is implemented depends on the context it is used. The following are some possible implementation heuristics.

Kingdom – King

The `Kingdom` and `King` association is intrinsically *one-one*. The implementation enforces that both sides of the association are consistent.

```
class Kingdom {
    King* _king;

  public:
    Kingdom(King *K):_king(K) { K->setKingdom(*this); }
    ...
};

King max;
Kingdom waste_land(&max);
```

Country –● City

A typical *one-many* association. The implementation enforces consistency.

```
class Country {
    list<City*> cities; // list of pointers to cities

  public:
    void addCity(City* c) {
      cities.insert(c);
   }
};

class City {
   Country *belongsTo;

 public:
   City(Country* C):_belongsTo(C) {
      C->addCity(this);
   }
};

Country Canada;
City Ottawa(&Canada);
```

`Course ●–● Student`

Many-many associations usually involve aliasing at both ends of the association. Special care must be taken to avoid infinitely recursive initializations.

```
   class Course {
       list<Student*> students;

     public:
       void register(Student *s) {
          s->addCourse(this);
          students.insert(s);
       }
   };

   class Student {
       list<Course*> courses;

     public:
       void addCourse(Course* C) {
          courses.insert(C);
       }

       void registerIn(Course* C) {
          C->register(this);
       }
```

```
};
```

University[student_number] – Student

An association is qualified if the qualifier attribute plays an important role in distinguishing the instances. This special role many times requires the selection of an appropriate data structure.

```
class University {
    hash_table<Student*> students;

  public:
    void addStudent(Student *s) {
       students.put(s->getStudentNumber(),s);
    }
};
```

Monitor ◇– Screen

```
class Monitor {
  Screen s;
  ...
};
```

Directory ◇–● File

```
class Directory {
   list<File*> files;
   ...
};
```

9.5 Reusable Design Patterns and Frameworks

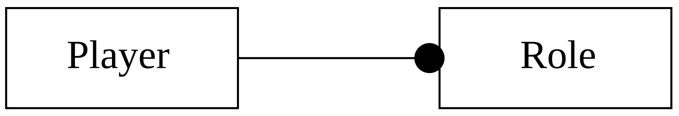
Probably every textbook on software engineering mentions that a good design maximizes *coherence* and minimizes *coupling*. Coherence in a design means that objects, data structures, functions or any other programming constructs are well defined and have a clear purpose. The Object Oriented paradigm is specifically good in maximizing coherence because the problem is first decomposed by isolating objects or actors which play some role in the application. Coupling is a measure of the dependence of such constructs or modules on other ones. In one word, abundance of pointers and propagating the task to be performed from object to object is a bad case of coupling.

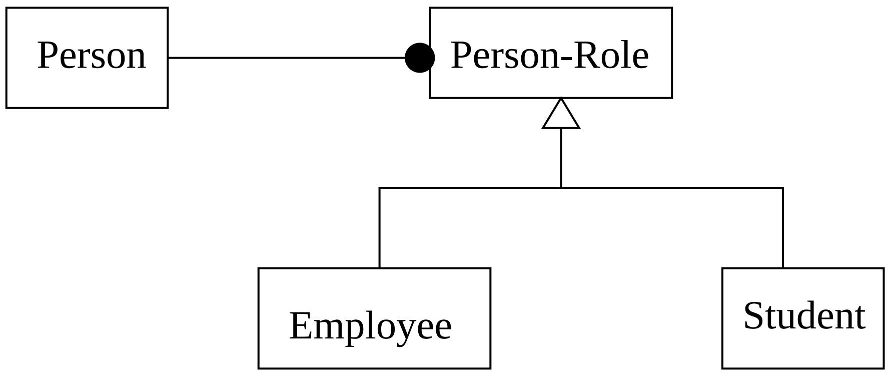
Unfortunately the Object Oriented paradigm by its very nature tends to introduce coupling into design.

Recently a new view emerged which does not mind coupling as far as it is automatically managed and is represented by a simple *design pattern*. Design patterns are usually recursive and most often they involve an association between a derived class and one or more instances of the base class. Via the constructor and destructor mechanism and the exploitation of run-time polymorphism, these patterns can give rise to almost arbitrarily complex run-time topologies and yet they are implicitly managed.

Some design patterns can be frequently found in object models and are well documented and researched. The next few are just a sample and meant to serve as an *eye-opener*. Understanding and using design patterns can significantly reduce development time and maintenance. New design patterns emerge very fast and now there are conferences and journals dedicated to this subject. A good (*but somewhat outdated*) reference is listed on page 234.

Player-Role Framework

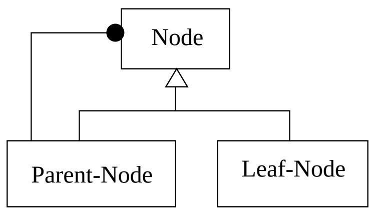


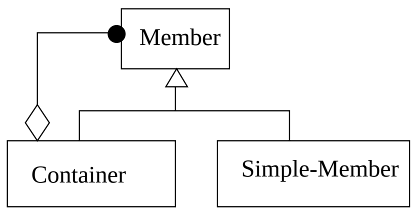


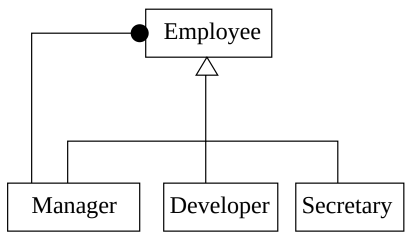
The player-role framework is a nice solution to avoid multiple inheritance. In the example, a `person` may have several roles: `student`, `employee` or *both*. This framework not only avoids the need of multiple inheritance, but it makes the design more extendable. For example a new role `volunteer` can easily be added.

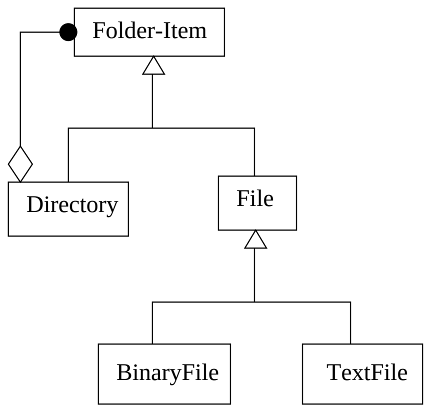
General Hierarchy Framework

There are two versions: one with a regular association and one with aggregation. Each are recursive and can be used to implement arbitrary multi-trees.









The directory structure can be implemented as:

```
class FolderItem {
    FolderItem* parent;  // parent == NULL
                         // implies root directory
  public:
    void setParent(FolderItem* p) {
       parent = p;
    }
};

class Directory : public FolderItem {
    list<FolderItem*> items;
  public:
    void addItem(FolderItem* p) {
      p->setParent(this);
      items.insert(p);
    }
};

class File: public FolderItem { ... }

class BinaryFile: public File { ... }
class TextFile: public File { ... }
```

There are many such reusable design patterns and frameworks, and new ones are constantly being engineered. The design and analysis of architectural frameworks is becoming a field on its own right with dedicated journals and conferences.

9.6 C++ and Object Oriented Trivia

This section is meant to be a review of the C++ and Object Oriented concepts presented in these notes in the form of questions. Most of the answers can be directly found while few (*) can be deduced from the notes and well known programming concepts.

1. list the stages of compilation of C++ projects and explain what each step does!
2. what are the *primitive* data types in C++ ?
3. describe C++ *scoping rules*!
4. what is a *namespace* and how does it benefit software development?
5. describe the iterative constructs available in C++ !
6. how does the `switch` construct work?
7. describe `break`, `continue` and `return`!
8. what are *enumerated* types and how are they used?
9. what is a *pointer*?
10. what are *pointers* used for?
11. how are arrays represented in C++ ?
12. what is the difference in memory layout between *statically* and *dynamically* allocated multi-dimensional arrays?
13. how can one allocate multi-dimensional arrays dynamically?
14. what is *pointer arithmetic*?
15. describe the relationship between pointers and arrays in C++ !
16. * why would one use *pointers to functions*?

17. describe what `new`, `new[]`, `delete` and `delete[]` do?
18. what is *typecasting*?
19. describe *compile-time*, *automatic* and *run-time* memory allocation!
20. describe the different ways of *parameter passing* in C++ !
21. describe the different ways of *returning values* in C++ !
22. what is the difference between a *reference* and a *pointer* parameter?
23. what is the difference between a *pointer* passed *by value* and a *pointer* passed *by reference*? give a concrete example when the latter is needed!
24. describe the different meanings of `static` in C++ !
25. what is a *class*?
26. what is the difference between an *instance* and a *class*?
27. what is an *instance variable*?
28. what is a *class variable*?
29. what is an *instance method*?
30. what is a *class method*?
31. what are the three criteria to be present for a language to be object oriented! describe each!
32. *** for a language to be *pure object oriented* the criterion that the *class* itself is an *object* must hold. what can one achieve by this that otherwise would be impossible?
33. what is the role of *constructors*?
34. what is the role of the *destructor*?
35. what is the role of the *copy constructor*?
36. what is the role of the *assignment operator*?
37. how do private instance variables declared in the base class get instantiated?
38. how should dynamically allocated private instance variables declared in the base class be released?

39. do *constructors* inherit? why?
40. does the *destructor* inherit? why?
41. what is the *initializer*? what is it used for?
42. what can go wrong if a class has a *copy constructor* but no *destructor*?
43. what can go wrong if a class has a *destructor* but no *copy constructor*?
44. explicitly describe what kind of classes do not need a destructor and a copy constructor!
45. describe how and when *constructors* get called implicitly!
46. describe how and when the *destructor* of a class gets called implicitly!
47. under what circumstances do temporary objects get created implicitly?
48. * describe how the copy constructor and the destructor should be implemented for a class which has pointer variables used for aliasing *and* to hold unique dynamically allocated objects!
49. ** suppose there is a class whose instances are shared amongst many other objects via pointers. is there a way to implement the *copy constructor*, *assignment operator* and the *destructor* such that when all the references are gone, the object deallocates itself? what are the challenges? what assumptions does the programmer have to make?
50. * describe a situation where it is advantageous to declare a local object inside an anonymous local scope – *in other words, inside a pair of curly braces which does not designate the body of a loop or a function*!
51. what does `inline` mean?
52. what kind of functions and methods cannot be made `inline` and why?
53. what is the advantage of using embedded classes?
54. what is the difference between classes declared by the `struct`, `class` and `union` keywords?
55. * describe an application where it is a good idea to declare a class with the `union` keyword!
56. describe how *pointers to member functions* differ from regular *pointers to functions*!

57. ** describe an application where it is appropriate to use *pointers to member functions*!
58. ** describe an application where *multiple inheritance* is not easily avoidable!
59. what is the difference between*error handling* and *exception handling*?
60. describe *exception handling* in C++ !
61. what are *exception handlers*?
62. why would someone catch an exception and re-throw it?
63. what is *polymorphism*?
64. what is *run-time polymorphism* or *dynamic binding*?
65. what is a *virtual method*?
66. * how does C++ implement *run-time polymorphism* (*or dynamic binding*)?
67. ** discuss the advantages and disadvantages of operator overloading!
68. ** describe an application where it is advisable to overload the free store operators (`new`, `new[]`, `delete` *and* `delete[]`)!
69. what is the difference between C 's free store functions (`malloc`, `calloc`, `realloc` *and* `free`) and C++ 's free store operators (`new`, `new[]`, `delete` *and* `delete[]`)?
70. how does *polymorphism* aid software development?
71. what is an *abstract* or *pure virtual* method? why do we need them?
72. what is an *abstract* class?
73. why would it be beneficial to model an application with *abstract* classes?
74. what is *generic programming*?
75. what are *templates*?
76. what are *iterators*?
77. how do *iterators* abstract data structures? why would one use *iterators* to access elements rather than the data structure's own methods?
78. why are container classes implemented with *templates*?

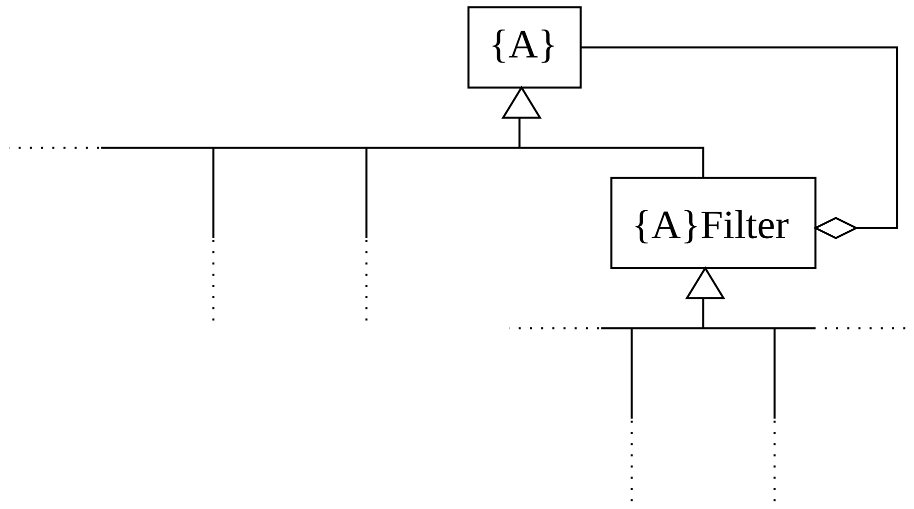
79. describe properties to compare data structures!
80. what kind of functions can/should be made parametric?
81. * describe a class which is not a kind of container but it makes sense to make it parametric!
82. what are *iostreams* in C++ ?
83. what is *serialization*?
84. how do `operator<<` and `operator>>` abstract input-output in C++ ?
85. what are I/O *manipulators*?
86. we have seen file- and string-streams. what else can be abstracted by streams?
87. describe a situation where the solution is to overload `operator<<` and/or `operator>>`!
88. describe a situation where the solution is to subclass `istream` and/or `ostream`!
89. describe a situation where the solution requires to overload `operator<<` and/or `operator>>` and to subclass `istream` and/or `ostream`!
90. describe the conceptual differences between C and C++ I/O!
91. ** it is a well known problem to instantiate objects of a new derived class from a stream without making modifications to the base class. describe where the difficulty lies and suggest solutions!
92. compare the object oriented paradigm to other well known programming paradigms!
93. how does an object model differ from functional or procedural decomposition?
94. what are the steps to build an object model?
95. what are the fundamental building blocks of an object model and how does one go about identifying them?
96. what are the most commonly used associations?
97. give general descriptions how to implement commonly occurring associations!
98. what are *design patterns*?
99. describe *design patterns* you know!
100. how does C++ compare to other languages you know? (*compare on relevant points identified by you*)

9.7 Exercises

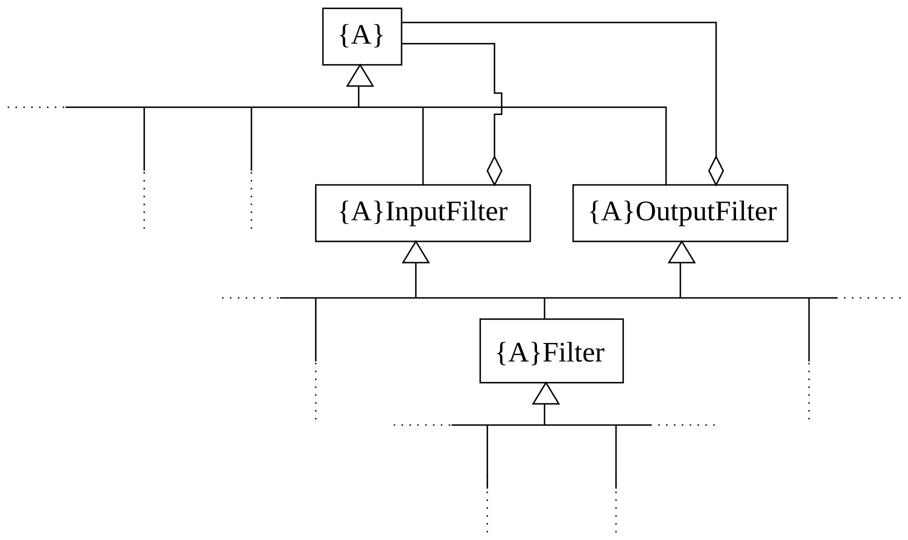
9.1 Explain in your own words the difference between an *operation* and its *method implementation*. Explain why it is a good (*or bad*) idea to create a model in terms of operations. How can this be exploited together with *inheritance*? Give examples of such models!

9.2 Consider the following design patterns and try to explain how they would work! `{A}` stands for a parameter that designates a class name. Write definitions in `C++` for the classes. Give examples how they would be used!!!

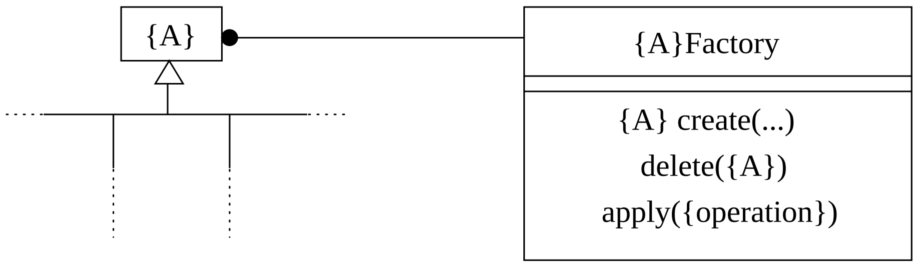
`{A}Filter` has the exact same operations as {A}. Moreover every method of `{A}Filter` **only** calls `{A}`'s implementation.



What about this one? (*Hint: what if {A} is a data structure?*) Is multiple inheritance a good idea here?



{A}`Factory` has many instances of {`A`} associated with it. The operation `create(...)` creates a new instance of {`A`} using the supplied parameters. Operation `delete` detaches the instance from the *factory*. Operation `apply(`{`operation`}`)` calls the operation on each and every instance of the factory.



9.3 Write sample implementations of the design patterns of the previous exercise for the application of your choice.

Chapter 10

Appendices

10.1 The C and C++ Preprocessor: cpp

C and C++ source code files are passed to the *preprocessor*, `cpp` , before they are compiled. The task of the preprocessor is to strip C and C++ comments and to replace macro names with their expansion. There are predefined and user defined macros. The preprocessor also understands *preprocessor statements.*

The following table contains the most commonly used and standard predefined macros.

Macro	Expansion	Example
`__FILE__`	expands to the name of the current input file	`"blah.cc"`
`__LINE__`	expands to the current line number	`123`
`__DATE__`	expands to the date when the preprocessor is run	`Dec 3 1997`
`__TIME__`	expands to the time when the preprocessor is run	`16:52:01`
`__STDC__`	expands to 1 for ANSI C	`1 or 0`
`__GNUC__`	expands to 1 for gcc	`1 or 0`
`__GNUG__`	expands to 1 for g++	`1 or 0`
`__cplusplus`	expands to 1 for C++	`1 or 0`
`__BASE_FILE__`	expands to the name of the main input file	`"main.cc"`
`__INCLUDE_LEVEL__`	the nesting depth of inclusions	`3`

Through *preprocessor directives*, the programmer can define his own macros and can strip or include code fragments using the *conditional preprocessor directives.* The following table contains the most frequently used preprocessor directives.

Directive	**Argument**	**Example**

`#define`	*symbol*
	symbol definition
		`#define _DEBUG_`
		`#define pi 3.14`
		`#define n2 ((n)*(n))`

`#undef`	*symbol*	`#undef pi`
`#ifdef`	*symbol*	`#ifdef pi`
`#ifndef`	*symbol*	`#ifndef _FOO_H_`
`#if`	*expression*	`#if __LINE__ > 23`
`#elif`	*expression*	`#elif defined vax`
`#else`	*none*	`#else`
`#endif`	*none*	`#endif`

`#include`	*<file>*
	"file"
		`#include <iostream.h>`
		`#include "foo.h"`

`#line`	*number*
	number filename
		`#line 25`
		`#line 24 "foo.cc"`

`#error`	*message*	`#error vax not supported`
`#warning`	*message*	`#warning this may crash`
`#pragma`	*option*	`#pragma once`

The one directive which almost all C and C++ programs must use is `#include`, which usually includes a header file. No identifier can be used without a definition. Definition of classes, functions, macros and references of external variables which reside in libraries or object files are always provided as *header* or *.h* files. When the preprocessor encounters the `#include` directive, it literally includes the file. If the file name is specified in angular brackets, then it looks for a file which resides in a directory which is in the *include path*. If the filename is specified within double quotes, the preprocessor looks for the file specified by the file path. For example:

```
#include "/usr/local/matlab/include/mc.h"
```

includes the file "/usr/local/matlab/include/mc.h", while

```
#include "foo.h"
```

includes the file from the current directory.

As `#include` directives can be nested many files deep, it is necessary to avoid multiple inclusions of the same header file because every identifier can have only one definition. The preprocessor directives `#ifndef`, `#define` and `#endif` can be used to guard a header file to be included multiple times. The standard template of guarding the header file `foo.h`:

```
#ifndef _FOO_H_
#define _FOO_H_
```

```
    ...  // body of foo.h
#endif
```

When the first time `foo.h` is included, `#ifndef _FOO_H_` is true (*if not defined _FOO_H_*). The very next line immediately defines _FOO_H_. Every conditional directive or cascade of directives must have a corresponding `#endif` directive. When the second time `foo.h` is the argument of an `#include` directive, `_FOO_H_` is defined, hence everything is skipped until the `#endif` directive.

Conditional directives can be used to skip lines of code depending whether a macro is defined. The *expression* argument of a conditional directive may be any constant integer expression. Expressions can be integer literals, macros and they can be combined using operators `==, <, >, <=, >=, !, !=, +, -, *, %, /, &&,` || and parentheses to explicitly specify precedence of evaluation. The `defined` symbol may be used to test if a macro is defined before it would be expanded.

The `#error` directive forces the preprocessor to terminate with non-zero error code. In other words, it can be used to prevent further preprocessing, and hence compilation. The `#warning` directive, on the other hand, prints the warning message, but does not stop the preprocessor.

The user can also use the `#define` directive to define more complicated expansions. The macro can have parameters, *like a function*, and the arguments with the current instantiations will be literally expanded. For example:

```
#define bug(A,B) A*B
```

is a macro which would expand `bug(4,7)` to `4*7`, while `bug(2+3,6)` would expand to `2+3*6`. To avoid the latter situation, *unless it is the expected behavior*, explicit bracketing should be used.

```
#define max(A,B) ((A)>(B)?A:B)
```

The `2*max(2+3,-5*3)` expands to `2*((2+3)>(-5*3)?2+3:-5*3)`.

Macros enclosed in double quotes are not expanded! (*"line number:* `__LINE__`*" does not expand*)

The following example demonstrates the use of the directives and macros.

```
#ifndef _FOO_C_
#define _FOO_C_

#ifdef WARNING
  #warning This is a warning
#else
  #if defined ERROR
    #error This is an error
  #endif
#endif

#define max(A,B) ((A)>(B)?A:B)
#define pi 3.14159

#include <iostream.h>

int main(void)  {

 int i = 0;

 #ifdef __cplusplus
    cout << "You are running a c++ compiler" << endl;
 #else  // #ifdef __cplusplus
    cout << "You are not running a c++ compiler" << endl;
 #endif // #ifdef _cplusplus

 #ifdef ONE
   i = 1;
 #elif defined TWO
   i = 2;
 #else
   i = -1;
 #endif

 cout << "The maximum of " << i << " and 3 is: " << max(i,3) << endl;
 cout << "The value of pi is " << pi << endl;

 #ifdef _DEBUG_
   cout << "You are in debug mode: " << endl;
   cout << "Line: " << __LINE__ << endl;
   cout << "File: " << __FILE__ << endl;
   cout << "Date: " << __DATE__ << endl;
   cout << "Time: " << __TIME__ << endl;
 #endif

 return 0;
}
#endif // #ifndef _FOO_H_
```

The `-D` flag can be used to define a macro for the preprocessor when running `g++` .

```
$ g++ -DERROR foo.cc
$ foo.cc:8:  #error This is an error
```

Executable not generated.

```
$ g++ -DWARNING -DTWO foo.cc
$ foo.cc:5:  warning:  #warning This is a warning
$ a.out
You are running a c++ compiler
The maximum of 2 and 3 is:  3
The value of pi is 3.14159
```

```
$ g++ -D_DEBUG_ foo.cc
$ a.out
You are running a c++ compiler
The maximum of -1 and 3 is:  3
The value of pi is 3.14159
You are in debug mode:
Line:  40
File:  foo.cc
Date:  Dec 3 1997
Time:  20:05:12
```

10.2 ANSI C++

The ANSI standardization effort of C++ has produced an even more complex language than the one described in these notes. As of today (*writing these notes*), ANSI C++ compilers are not readily available, compilers partially support the ANSI C++ standard and many of the standard features are *not* portable. The foundation classes are still being built and debated. The standard template library (*STL*) is becoming more popular, but many C++ applications still use in-house development (*like custom container classes*). Many currently being developed C++ applications utilize C libraries and use C style I/O (*as opposed to streams*) and mix C style memory management with C++ 's `new` and `delete` operators. The concept of *namespace* and *exception handling* are most certainly will be supported by all future C++ compilers and will play an important role in C++ development.

True Boolean Type

C++ has a true boolean type called `bool`. Logical operators in ANSI C++ hence should return `bool` as opposed to `int`. There are also two boolean constants: *true* and *false*. An integer is *implicitly* type casted into a `bool` if it is used in the context of a boolean, so code that uses `int`s as opposed to `bool` still works as expected. New development should use `bool` instead of `int`.

True String Class

C++ has a true string class (`string`) which can be used to replace `char*` in many contexts.

Using compiled C code in C++ applications

C++ is not only a superset of C , but it also changed some fundamental components of the C language. When libraries compiled with a C compiler are combined with C++ applications, the interface definitions should be appropriately changed. In particular, the header file of a library should be put into the following *linkage block*:

```
extern "C" {
 #include "oldcode.h"
 #include <math.h>
}
```

10.3 Library vs. Systems Calls

C was originally designed to implement the UNIX operating system and to be used as a systems programming language as well as for implementing general purpose applications. Most modern operating systems are written in C and C++ . C++ can also be used to write system level programs, but the object oriented paradigm is not the most suitable for low level functional tasks that must be able to run fast and with limited space requirements.

As C and C++ are also systems programming languages, they can call system functions directly. A system call can be thought of as an atomic block of functionality which is executed within the operating system *kernel*. System calls are operating system and device dependent, so their use is only suggested for low level operating system tasks and for special device access. The system interface is also wrapped into *library functions* that provide a more portable but somewhat less efficient way of performing the same tasks.

The full bundle of system calls available on a UNIX platform are listed in the second chapter of the *man pages*. The third chapter describes the *library wrappers* for these system calls.

10.4 Passing An Arbitrary Number of Parameters

It is sometimes convenient to pass an arbitrary number of parameters to a function. For example, the `printf` and `scanf` functions can take an arbitrary number of parameters. The catch is that the compiler has no way of checking the type of the parameters, hence the programmer must implement (*or assume*) a type for each. The type `va_list` is used to declare a variable `ap` that represents the parameter list and the *macro* `va_start` initializes `ap` to point to the first *unnamed* argument. There must be at least one named argument. In C++ , a function that can accept a variable number of parameters must use . . . (*ellipsis*) to indicate the unnamed arguments. `va_arg` can be used to step `ap`, and it also needs the assumed type (*or class*). When the unnamed arguments have been stepped through, `va_end` *must be* called, otherwise the function may not be able to return properly. There is no way to know how many arguments were passed, so either a named argument with the expected number of arguments must be passed or one of the named arguments must encode the expected number of arguments somehow or the last argument must be reserved to signal the end.

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

double multi_arithmetic(char oper,...) {
    // executes oper on the arguments
    // the last argument must be 0

    va_list ap; // ap can be used to step through ...

    double num,result;

    va_start(ap,oper); // initialize ap to point
                       // to the first argument after oper

    int isFirst = 1;
    while (1) {
      num = va_arg(ap,double);
      if (num==0) break;
      if (isFirst) {
         result = num;
         isFirst = 0;
         continue;
      }
      switch(oper) {
         case '+': result+=num;
                   break;
         case '-': result-=num;
                   break;
         case '*': result*=num;
```

```
                    break;
         case '/': result/=num;
                    break;
      }
    }

    va_end(ap);
    return result;
}


int main(void) {

  double a=1,b=2,c=3,d=4;

  cout << "+ 1 2 3 4 = " << multi_arithmetic('+',a,b,c,d,0) << endl;
  cout << "- 1 2 3 4 = " << multi_arithmetic('-',a,b,c,d,0) << endl;
  cout << "* 1 2 3 4 = " << multi_arithmetic('*',a,b,c,d,0) << endl;
  cout << "/ 1 2 3 4 = " << multi_arithmetic('/',a,b,c,d,0) << endl;

  return 0;
}
```

10.5 String Functions

This section is a sample of the most commonly used string functions defined in `string.h`.

Concatenating strings

```
char *strcat(char *dst, const char *src)
char *strncat(char *dst, const char *src, size_t n)
```

`strcat` appends the characters of `src` after the null byte (`'0'`) of the string `dst`. `dst` is assumed to have enough memory allocated to hold the resulting string. `strncat` is analogous to `strcat` but it copies *at most* `n` characters from `src` but stops if it encounters the null byte in `src` earlier.

Finding the position of a character

```
char *strchr(const char *s, int c)
char *strrchr(const char *s, int c)
char *strpbrk(const char *s1, const char *s2)
```

`strchr` returns a pointer to the first occurrence of `c` (*converted to a char*) in string `s` or a null pointer if `c` does not occur in the string. `strrchr` returns a pointer to the last occurrence of `c`. The null character terminating a string is considered to be part of the string.

`strpbrk` returns a pointer to the first occurrence in string `s1` of any character from string `s2`, or a null pointer if no character from `s2` exists in `s1`.

Substrings

```
char *strstr(const char *s1, const char *s2)

char *strtok(char *s1, const char *s2)
char *strtok_r(char *s1, const char *s2, char **lasts)
```

`strstr` locates the first occurrence of the string `s2` (*excluding the terminating null character*) in string `s1`. `strstr` returns a pointer to the located string or a null pointer if the string is not found. If `s2` points to a string with zero length (*that is, the string* `""`), the function returns `s1`.

`strtok` can be used to break the string pointed to by `s1` into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by `s2`. `strtok` considers the string `s1` to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `s2`. The first call (*with pointer* `s1` *specified*) returns a pointer to the first character of the first token, and will have written a null character into s1 immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (*which must be made with the first argument being a null pointer*) will work through the string s1 immediately following that token. In this way subsequent calls will work through the string `s1` until no tokens remain. The separator string `s2` may be different from call to call. When no token remains in `s1`, a null pointer is returned.

`strtok_r` has the same functionality as `strtok` except that a pointer to a string place-holder `lasts` must be supplied by the caller. The `lasts` pointer is to keep track of the next substring in which to search for the next token.

Comparing strings

```
int strcmp(const char *s1, const char *s2)
int strncmp(const char *s1, const char *s2, size_t n)
```

`strcmp` compares two strings a character at a time, according to the ordering of your machine's character set. The function returns an integer greater than, equal to, or less than 0, if the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2` respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared. `strncmp` makes the same comparison but looks at a maximum of `n` bytes. Bytes following a null byte are not compared.

Copying strings

```
char *strcpy(char *dst, const char *src)
char *strncpy(char *dst, const char *src, size_t n)

char *strdup(const char *s1)
```

`strcpy` copies string `src` to `dst` including the terminating null character, stopping after the null character has been copied. `strncpy` copies exactly `n` bytes, truncating `src` or adding null characters to `dst` if necessary. The result will not be null-terminated if the length of `src` is `n` or more. Each function returns `dst`.

`strdup` returns a pointer to a new string that is a duplicate of the string pointed to by `s1`. The space for the new string is obtained using `malloc`. If the new string cannot be created, a null pointer is returned.

String length

```
size_t strlen(const char *s)

size_t strcspn(const char *s1, const char *s2)
size_t strspn(const char *s1, const char *s2)
```

`strlen` returns the number of bytes in `s`, not including the terminating null character.

`strcspn` returns the length of the initial segment of string `s1` that consists entirely of characters not from string `s2`. `strspn` returns the length of the initial segment of string `s1` that consists entirely of characters from string `s2`.

10.6 The make Utility

The make utility was invented to simplify the compilation and assembly of large projects. Even for a few files it is worth to generate a *makefile*. The make utility is significantly more complex and powerful than we present it here. Our major concern is to write *makefiles* which can be used to compile C++ projects. The power of make stems from its declarative style. A *makefile* contains *dependency rules* and the make utility, using the dependencies, builds the project with the minimum number of calls to the compiler.

A dependency rule has the following form:

target : *dependency*$_1$ *dependency*$_2$ ...
"tab character" rule

The target and the dependencies can be file names. The make utility checks the file creation date of the target and all of the dependencies. If any of the dependency files has a later creation date than the target, then the rule is executed. Every rule must be preceded by one *tab* character, which must also be the first character of the line. For example:

```
foo.o:  foo.cc foo.h
    g++ -c foo.cc
```

is a dependency rule which executes `g++ -c foo.c` if `foo.o` is older than `foo.c` of `foo.h`.

Macros can be defined to avoid repetitions and to parameterize commonly used utilities within make . Macros can be defined with the following syntax:

```
name = text string
```

For example `PROG = foo` defines the macro `PROG`, and make replaces every occurrence of `${PROG}` with `foo`. The $ sign can be used to dereference macros. For example, the previous dependency rule could have been written as:

```
CC = g++
foo.o:  foo.cc foo.h
    ${CC} -c foo.cc
```

make also provides the following predefined macros:

Macro	**Meaning**
`$?`	Those elements of a dependency rule which are younger than the target
`$@`	The name of the current target
`$<`	The name of the current prerequisite in a suffix rule
`$*`	The name, without the suffix, of the current prerequisite in a suffix rule

make also allows the definition of *suffix rules* which automatically build a prerequisite if it has a matching suffix rule. A suffix rule has the form:

```
.<source-suffix>.<target-suffix> :
"tab character" rule
```

For example the suffix rule:

```
.c.o:
    ${CC} -c $<
```

defines how to build a `.o` file from a `.c` file. Once this rule is specified, make can automatically build object files and it can also do it implicitly if a dependency is a `.o` file and there is a C source file with the same stem and suffix `.c`.

The following is a *makefile* for the digit recognition application discussed in the first lecture (*University of Ottawa, "csia" SunOS server*).

```
#!/bin/sh

# CC is the C++ compiler
CC = g++
# -g is the debug flag, another useful flag is -O
# -O is for "optimization"
CFLAGS = -g

# -L. says that the libraries are in the current directory
LDLIBS = -L. -lmath -lnnet

# AR is the archiving program
AR = ar

# rc passed to ar forces  the creation of a new archive
ARFLAGS = rc

# add the suffix .cc to the suffices
.SUFFIXES: .SUFFIXES .cc


# the suffix rule specifies how to build
# an object (.o) file from a .cc file
.cc.o:
         ${CC} -c $< ${CFLAGS}

# digitrec is the application
digitrec: digitrec.o main.o libnnet.a libmath.a
         ${CC} -o $@ digitrec.o main.o ${LDLIBS}

# dependencies for the math library
libmath.a: matrix.o random.o digitrec.o nnet.o
```

```
        ${AR} ${ARFLAGS} $@ matrix.o random.o digitrec.o nnet.o
        ranlib $@

# dependencies for the neural networks library
libnnet.a: nnet.o
        ${AR} ${ARFLAGS} $@ nnet.o
        ranlib $@

# dependencies of object files
main.o: main.cc
digitrec.o: digitrec.cc math.h nnet.h digitrec.h
nnet.o: nnet.cc nnet.h math.h
ode.o: ode.cc ode.h calc.h
calc.o: calc.cc calc.h
matrix.o: matrix.cc matrix.h calc.h
random.o: random.cc random.h
```

`make` looks for the file `Makefile` in the current directory, hence you have to save the file under the name `Makefile`[1]. Finally, to run make , you only need to type make at the prompt:

```
$ make
```

And the project will be built.

[1]it is also possible to specify another file. for the complete list of options refer to the man pages on make

10.7 Huggy & Muggy

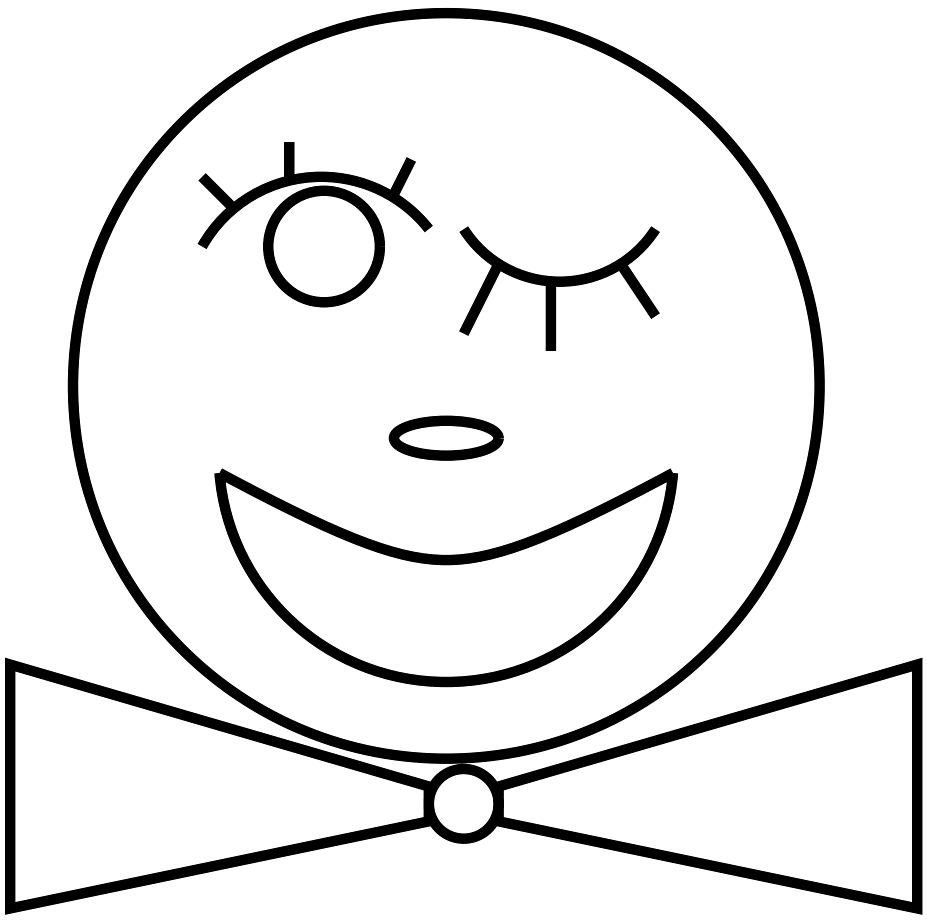


Figure 10.1: Muggy

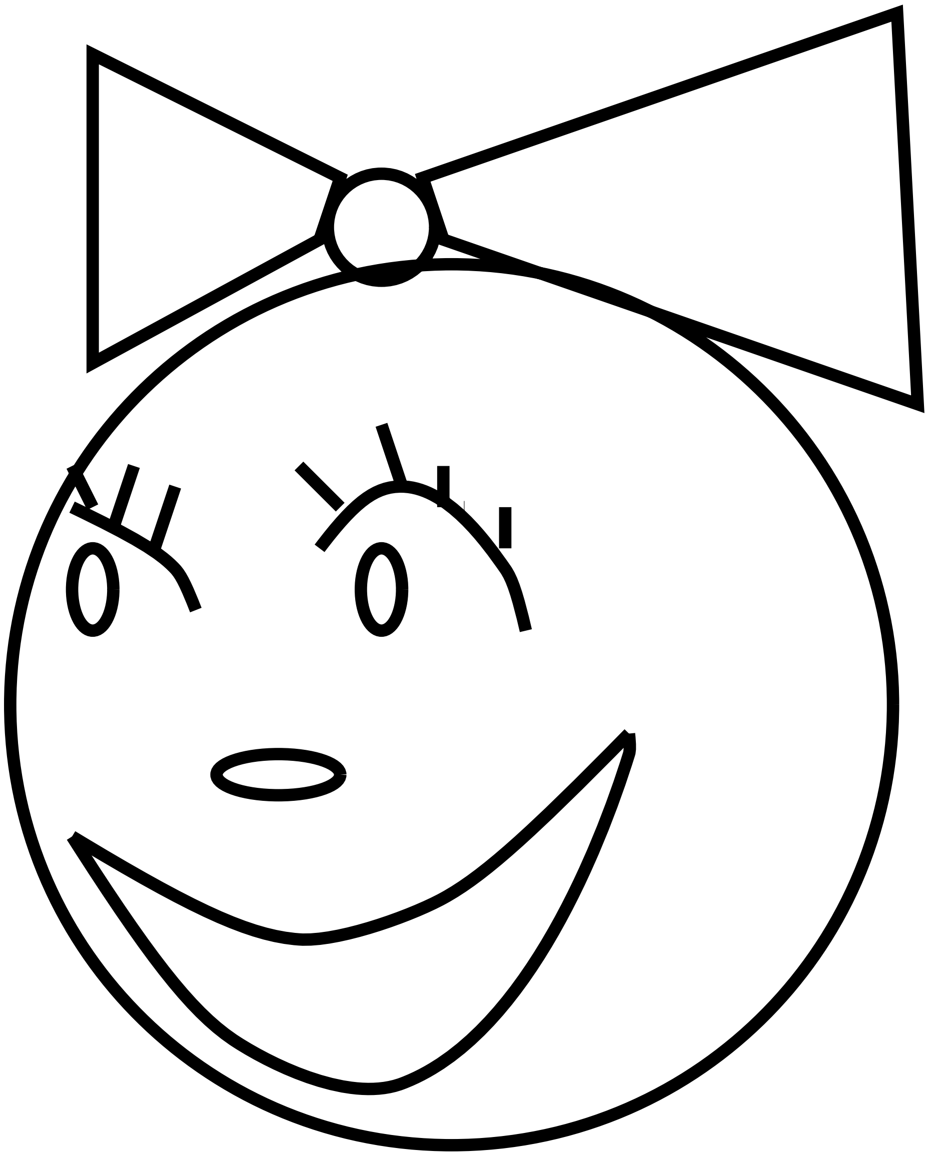


Figure 10.2: Huggy

10.8 References

M. Ellis, B. Strostroup, *The Annotated C++ Reference Manual*, Addison & Wesley, 1990

E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison & Wesley, 1995

A. Oram, S. Talbott, *Managing Projects with* `make`, O'Reilly & Associates, 1991

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991

A. Stepanov, M. Lee, *The Standard Template Library*, Hewlett-Packard Company, 1995

B. Strostroup, *The Design and Evolution of C++*, Addison & Wesley, 1994

B. Strostroup, *The C++ Programming Language*, Addison & Wesley, *Second edition:* 1991, *Third edition:* 1997

Index