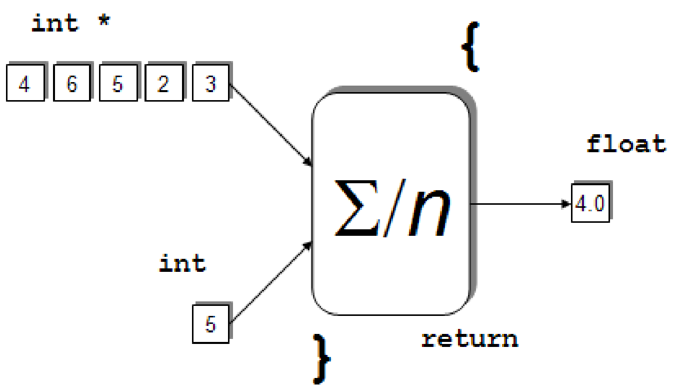
An introduction to programming with C++



Saad Mneimneh

Contents

Chapter 1

A quick overview of computing

1.1 Introduction

It has become clear to me over the years that, whenever students see the word "computer", they visualize this thing:

Figure 1.1: Computer ?

Well, more or less... Of course, the figure that will register in your mind depends on the setting and on the current technology (which is very dynamic by the way). For instance, I am sure that, to you, a computer would range from being a laptop, to possibly an advanced PDA (Personal Digital Assistant) or a smart phone (hence I will not ask you to turn off your cell phone in class).

Nevertheless, Figure 1 is definitely a standard interpretation of the word computer. Next thing you know is that someone is sitting behind the keyboard, looking very eagerly into the screen, and with the only moving parts of the body being the eyes and fingers: That's what a general audience would think computer science is all about. This course will hopefully help us change that image.

The concept that we should focus on is computation. Let's start with a (rather unclear) definition:

com.pu.ta.tion: *noun* **a:** the act or action of computing, **b:** the use or operation of a computer   (Merriam-Webster Dictionary)

Although computation is carried out by a computer (meaning **b**), computation is also an abstract notion (meaning **a**). It is a process by itself that we find everywhere in nature. To some extent is it independent form the underlying physical machine (e.g. the computer). Computation is what makes the field of computer science, a science. We will explore this notion further when we talk about *Algorithms*.

For the time being, let me attempt to *shake* or *blur* Figure 1 in your mind. I will do that by drawing a parallel between a biological process and a few components of a "computer". This will help interpret the biological process as a computation. But more importantly, this computation, being biological in nature, does not require the computer of Figure 1.

1.2 Who said the first computer was invented in the 1940s?

An important question in biological sciences is what makes life? A simple answer is Proteins and Nucleic Acids. Proteins are responsible for almost all body functions. Nucleic acids encode information necessary to produce the proteins and pass this "recipe" to subsequent generations (permanent storage). We have two kinds of nucleic acids: ribonucleic acids RNAs and dioxy-ribonucleic acids DNAs.

A DNA is a chain of simpler molecules, namely sugar molecules. Each sugar molecule is attached to a base and this is what makes it different from the other sugar molecules. We have 4 bases, A, G, C, and T, thus the DNA can be viewed as a long sequence of 4 letters.

The DNA is actually a double stranded helix (discovered in 1953). The two strands hold together because each base in one strand bonds with a complementary base in the other. A$\leftrightarrow$T and C$\leftrightarrow$G. This makes it more stable and suitable as a storage device. The RNA is similar to the DNA but it is single stranded (hence less stable), and every occurrence of a T is replaced by a U. U also bonds with A.

The genome is a collection of long DNAs that are called chromosomes. A gene is a stretch along a chromosome that encode the information for producing a specific protein. Only certain stretches on the chromosomes are genes. If fact, it was believed that 90% of the our DNA is *junk* (well, not anymore!).

How does a gene produce a protein? A process called *transcription* creates an RNA by separating the DNA strands around the gene area (see figure below). The RNA is then synthesized into a protein. This latter process is called *translation*. The protein interacts with other proteins, with RNAs, and with DNA to perform several important functions in the body. The following figure illustrates the two processes of transcription and translation.

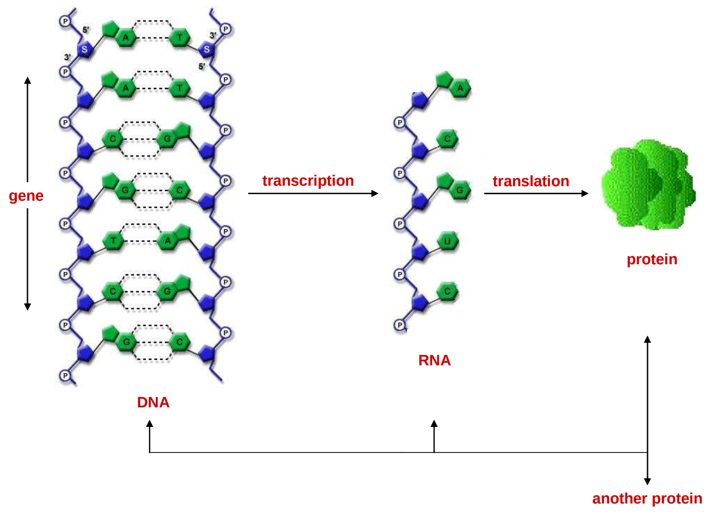


Figure 1.2: Biology in one picture

Here's a parallel of the biological process that is seen in computers. Much like DNA, the hard disk of a computer permanently stores information for later retrieval. The information is stored in some form; for instance, if it is a code, it is usually in a textual format, following a certain syntax, like C++ or Java. The code plays the role of an RNA. While an RNA is synthesized into a protein, the code is compiled into an executable program that is loaded into memory. Now the program is our protein. The program runs and performs some tasks while interacting with other programs (e.g. networks), reading and writing data from and to the hard disk, and possibly updating other codes.

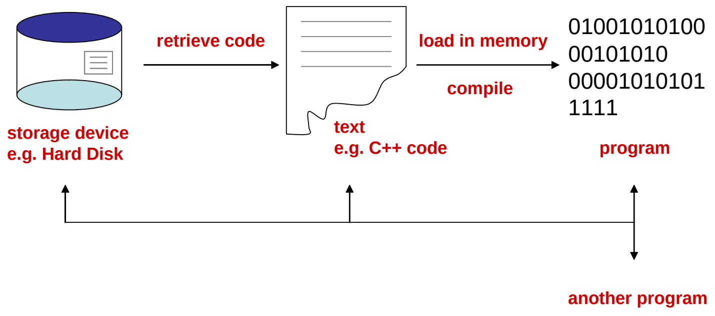


Figure 1.3: Biological process in computer jargon

The following table summarizes this parallel between a *biological computer* and the *electronic computer* that we know:

Biology	Electronics
DNA (chemical)	Hard disk (magnetic)
Transcription	Data retrieval
RNA (chemical data)	Code (textual data)
Translation	Compilation
Protein	Program

Therefore, one can claim that the biological process is actually a computation. Moreover, this makes our body a computer!

1.3 So what is a computer then?

Well, a computer is a *device* (now electronic) that carries out a computation (circular definition?). Without plunging deeply into linguistics, a computer is simply a device that can store, retrieve, and process data. According to this definition, your brain is a computer! Don't be surprised because this is definitely true (I will prove it to you shortly).

We have seen the idea of storage and retrieval when we discussed the parallel between a biological computer and an electronic one. What about processing? Well, in modern computers, processing is done mainly using a memory and a Central Processing Unit (CPU). When the program obtained by the compiler is executed, the following occurs:

- The program is loaded into memory by the Operating System (OS), which is another program already running
- The operating system instructs the CPU to start reading instructions at the given memory location where the program is loaded
- The CPU start fetching instructions and data into special registers
- The registers act like the short term memory of the CPU, they hold data needed to perform the most recently fetched instructions
- An Arithmetic and Logic Unit (ALU) inside the CPU performs the operations

Here's the big picture:

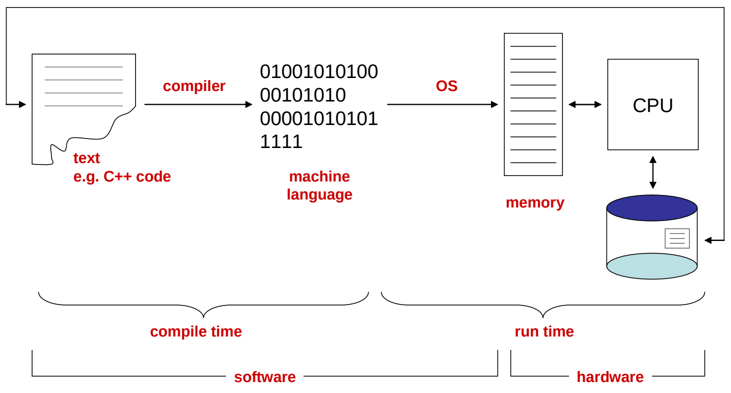


Figure 1.4: The big picture, well not so big...

Since we often require a computer to accomplish a certain task, a computation is not just a number of random operations consisting of data storage, retrieval, and processing. The most fundamental concept in computer science is the *Algorithm* [1]. Informally speaking, an algorithm is a well defined sequence of computational steps that accomplish a certain task. The task can be visualized

[1] The word algorithm comes from the arabian mathematician **Alkowarizmi** 780 - 850 A.D.

as a relation between an input and an output. Therefore, an algorithm is a well defined sequence of computational steps that transforms some input into the desired output. Here are some examples:

- sorting numbers: the input is a set of numbers, the output is the same set of numbers in sorted order (from smallest to largest). The algorithm in this case is a sorting algorithm.
- cooking: the input is a set of ingredients, the output is the dish. The algorithm in this case is a recipe.
- addition: the input is two numbers, the output is one number which is their sum. The algorithm in this case is an adder.

Therefore, before a computer can perform a task, an algorithm for performing that task must be discovered. In fact, designing algorithms is one of the most important fields in computer science. The problem of finding algorithms goes back to the 1930s when mathematicians like Kurt Gődel, Alan Turing, Alonzo Church, and others were trying to answer the fundamental question whether every task has an algorithm. Is is actually then that the stage was set for the emergence of a new discipline known as computer science.

With the power of abstraction, once an algorithm for a particular task has been found, and proven correct, then it can be used safely as a black box or as a building block, without having to worry about how or why this algorithm works (unless otherwise you are interested in such knowledge). Of course, issues such as writing the code, finding an efficient implementation, or adapting to a special machine will eventually arise.

To materialize the notion of algorithm, let's look for a moment at the task of adding two numbers. We all know how to add two numbers (hopefully). Let's try the following (thanks to Yale Patt for the idea):

Add 12 and 8:

$$\begin{array}{r} 12 \\ +\quad 8 \\ \hline \end{array}$$

Add 129 and 17:

$$\begin{array}{r} 129 \\ +\quad 17 \\ \hline \end{array}$$

Read in reverse to see the answers. .641=71+921, 02=8+21 :srewsnA

If everything works as I expect, you will actually perform two different algorithms, one for each addition. For the first addition, you are likely to write down the digit 2 followed by the digit 0. The answer is 20. For this addition, you perform a simple *lookup*. The answer to 12+8 is somehow stored permanently in your brain. When you see the pattern 12+8, you access that area in your brain and retrieve the answer. For the second addition, you are likely to write down the digit 6 first, followed by the digit 4, followed by the digit 1. The answer is 146. For this addition, you perform the standard addition procedure that ripples a carry. Therefore, you add 9 and 7, which gives 16, you write down the 6 and you carry 1, etc... You use your short term memory (or in this case maybe the paper) to keep track of the carry. In either case, you are doing what a computer would do.

You may be unaware of the *representation* of the addition algorithm in your brain, but it is definitely encoded into your brain cells. Similarly, a computer needs a way to represent algorithms in a form that is compatible with the machine. A representation of an algorithm is often called a *program*. You may have seen these programs, written in C++ or Java for instance, and printed on a piece of paper, or displayed on the screen. The whole process of writing the program and encoding it is called *programming*. Programs and the algorithms they represent are called *software*. The machine itself is referred to as *hardware*. Therefore, the software must be compatible with the hardware for it to work. For instance, a PC and a MAC require different software.

1.4 The standard hello world in C++

Here's a program in C++ that outputs "Hello World":

```
#include <iostream>

using std::cout;

int main() {
  cout<<''Hello World\n'';
}
```

Chapter 2

Basic elements of programming

2.1 Introduction

We argued last time that a program is a representation of an algorithm. Therefore, any powerful programming language must provide elements to represents ideas and thoughts. These elements are:

- Expressions: represent simple entities, like integers, characters, strings, etc...
- Combination: provides a way to build compound expressions from simpler ones
- Abstraction: provides ways by which compound expressions can be named

For instance, the following program outputs the number 486.

```
#include <iostream>

using std::cout;

int main() {
  cout<<486;
}
```

To output the character 'c', we could have written the following instead:

```
  cout<<'c';
```

There are special characters that can be outputed using the escape character backslash '\'. Such special characters are called escape sequences:

escape sequence	description
\n	newline
\t	horizontal tab
\r	carriage return
\a	alert (beep)
\\	backslash
\'	single quote
\"	double quote

To output the string “Hello World” followed by a newline, we could have written the following statement:

```
cout<<''Hello World\n'';
```

Basic expressions can be combined together to form more complex expressions. By now you might have noticed that every *statement* in C++ must end with a semicolon. A statement is a form of combining expressions together. For instance “cout<<486” consists of the output stream, which is a built in entity, the << operator, and the number 486. The semicolon provides a way to separate statements. Moreover, a bunch of statements can be combined together using the open { and close }. Here are some examples of combining expressions in statements using the arithmetic operators and numbers.

```
#include <iostream>

using std::cout;

int main() {
  cout<<137+349<<''\n'';
  cout<<1000-334<<''\n'';
  cout<<5*99<<''\n'';
  cout<<10/5<<''\n'';
  cout<<2.7+10<<''\n'';
  cout<<7%2<<''\n'';
}
```

The above example will produce the following:

```
486
666
495
2
12.7
1
```

The following table describes the different operators used above:

operator	description
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
$\%$	modulus

In principle there is no limit to how complicated an expression is, here’s an example:

```
cout<<3*((2*4)+(3+5))+((10-7)+6);
```

This will evaluate to 51. An interesting question is the following: how is the above expression evaluated? Each arithmetic operator must have a value for both of its operands. Therefore, the innermost parentheses are evaluated first. We say that parentheses have the highest precedence among the operators because they enforce an order on the evaluation of the sub-expressions. Without parentheses, the above expression would look like the following:

```
cout<<3*2*4+3+5+10-7+6;
```

This will evaluate to 42. In general, the order of evaluation of an expression depends on the precedence of the operators in that expressions. The following table summarizes the precedence level of various operators:

operator	order of evaluation
()	Evaluated first. If parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs at the same level, they are evaluated left to right.
$*$ / %	They are evaluated second. If there are several, they are evaluated left to right.
$+$ $-$	Evaluated last. If there are several, they are evaluated left to right.

Let's us write a program to output the sum of the integers from 1 to 100, i.e. $\sum_{i=1}^{100} i$.

```
#include <iostream>

using std::cout;

int main() {
  cout<<1+2+3+4+5+6+7+8+9+10+
        11+12+13+14+15+16+17+18+19+20+
        21+22+23+24+25+26+27+28+29+30+
        21+22+23+24+25+26+27+28+29+30+
        31+32+33+34+35+36+37+38+39+40+
        41+42+43+44+45+46+47+48+49+50+
        51+52+53+54+55+56+57+58+59+60+
        61+62+63+64+65+66+67+68+69+70+
        71+72+73+74+75+76+77+78+79+80+
        81+82+83+84+85+86+87+88+89+90+
        91+92+93+94+95+96+97+98+99+100;
}
```

This program definitely works and outputs the correct answer. We can safely claim that this program is a representation of a correct algorithm to compute $\sum_{i=1}^{100} i$. However, it suffers from two drawbacks:

- the length of the representation is proportional to the number 100
- if 100 is to be replaced by another number, the code need to be re-written

Let us investigate how we would perform this particular task using our brain. Without being smart about it like Gauss [1], we keep a running sum and we repeatedly add the next number to it. This means that we actually perform the following operation (...+(((1+2)+3)+4)+...+100), which is identical to how the above program evaluates the expression (left to right). The problem with the above program, however, is the lack of abstraction. The process in our brain uses two place holders x and y that evolve as follows:

[1]Gauss discovered that $\sum_{i=1}^{n} i = n(n+1)/2$ at the age of 10.

```
$x \leftarrow 0$
$y \leftarrow 1$
**repeat**
    $x \leftarrow x + y$
    $y \leftarrow y + 1$
**until** $y > 100$
```

Figure 2.1: Mental process for $\sum_{i=1}^{100} i$

This process illustrates three important concepts:

- Naming: we assign names to things (x and y)
- Initialization: we initialize our brain with some state ($x = 0$ and $y = 1$)
- Repetition: we repeat the same process over and over
- Conditions: we test whether some condition occurs

Therefore, to capture such a process, a powerful programming language must provide a way to name things and initialize them. Without names, we cannot reuse ideas and concepts. Every time we have to express an idea, we have to start from scratch, i.e. re-invent the wheel; very much like what we have to do to re-write the above program to compute $1 + 2 + \ldots + 200$.

2.2 Variable declaration, type, and scope

C++ is an explicitly typed language. This means that every name and expression must have a type. So what is a type? A type gives information about the kind of data a name or an expression represents. Types make code easier to understand. In fact, programmers spend more time on understanding code than on writing code. Consider the following:

```
int main() {
  x=1/3;
}
```

What is x? Is it 0.33333? or simply 0? If x is an integer, then it is definitely 0. If x is a real number, then it is 0.33333. So which one is it? Types are added cost for making the code clear. An explicitly typed language like C++ requires every name to be declared with a type.

```
int main() {
  int x=2; //an integer
  float y=1/3; //a real number
}
```

We call this a variable declaration. In general it has the following form:

type name = *expression*

The *name* identifies a variable of the type *type* whose value is given by *expression.*

```
int main() {
  float pi=3.14159;
  int radius=10;
  float circumference=2*pi*radius;
  cout<<circumference<<''\n'';
}
```

Puzzle: find out what the output of this program is and explain it to yourself:

```
#include <iostream>

using std::cout;

int main() {
  float x=1/3;
  cout<<x;
}
```

A value in a variable declaration is optional, but it is always recommended to initialize the variable; for instance, the following program outputs 134514228.

```
int main() {
  int m;
  cout<<m;
}
```

It is worth mentioning here the conceptual difference between initialization and assignment, as shown below:

```
int main() {
  int m=0; //initialization
  cout<<m;
}
```

```
int main() {
  int m;
  m=0; //assignment
  cout<<m;
}
```

Assignment is done using the assignment operator =. Initialization is done in a number of ways, but the syntax for declaration shown above is the most common (we will later see other ways). We can only assign variables that are already declared. In contrast, initialization occurs simultaneously with declaration. Therefore, we do not refer to the = symbol in a declaration as the assignment operator. Although the above examples do not exhibit any sensible difference between initialization and assignment (they both do the same thing), we will later see examples where the choice of initialization or assignment makes a difference.

Every name has an environment in which it is accessible. For instance, if you re-examine the mental process for computing $1+2+\ldots+100$, you will find that x is a place holder for the sum computed so far. But x did not exist in your head before you asked the question "what is $1+2+\ldots+100$?". Moreover, x lives only in your head and not in someone else's head. What is even more important is that someone else's x if it exists, is not the same x as yours.

We often refer to the "scope" of x. In C++, the scope of a name is the block in which it was first declared, where the block is determined by the open/close brace.

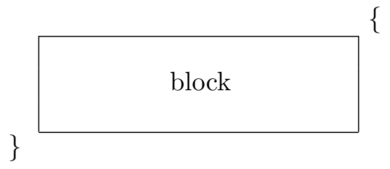


Figure 2.2: A block

A name is not accessible outside its scope. This locality is very important for abstraction as we will see later when we study functions. In the following program, the second reference to y will cause an error.

```
int main() {
  int x=2;
  {
    float y=1/3;
    cout<<x<<’’\n’’;
    cout<<y<<’’\n’’;
  }
  cout<<y<<’’\n’’; //not the scope of y
}
```

Note that two variables cannot be declared with the same name in the same scope (why?).

2.3 Memory

When a variable is declared, a space is allocated in the computer's main memory for it. We say that the variable is created in memory. In fact, that's what the human mind does too (think about that mental process for computing $1 + 2 + \ldots + 100$. This brings us to the following three questions:

- How much space does the variable occupy in memory (in bytes)?
- Where is it in memory?
- How long will it stay there?

The answer to the first question is that it depends on the type of the variables. Below is a list of most of the C++ fundamental types, grouped in two categories: integral types, and floating types. Each unsigned type has the same memory requirement as its corresponding type.

integral types	floating types
bool (true/false)	float
char	double
unsigned char	long double
short	
unsigned short	
int	
unsigned int	
long	
unsigned long	

The size in bytes for each type may depend on the machine; however, they are listed in a non-decreasing order. Usually, bool and char take 1 byte each. On a 32 bit machine, an int takes 32 bits, i.e. 4 bytes. Therefore, we may represent 2^{32} different values with an int. Half of the values represent positive integers and half of the value represent negative integers. Therefore int $\in [-2^{31}, 2^{31}-1]$. On the other hand, unsinged int represents only positive integers. Since both have the same memory requirements, unsigned int $\in [0, 2^{32}-1]$.

Consider the following program:

```
#include <iostream>

using std::cout;

int main() {
  int n=1000;
  cout<<n*n*n*n;
}
```

Theoretically, the answer should be 1000000000000. However, we will see -727379968. The reason for this is that 1000000000000 cannot fit in 32 bits (even when using unsigned int). This is called an overflow error.

While integral types are exact (integer types), floating types are not. Consider the following program:

```
#include <iostream>

using std::cout;

int main() {
  float x=100/3.0;
  cout<<x*3-99;
}
```

Theoretically, the answer should be 1. However, we will see 0.99996. That's because 1/3 cannot be represented accurately with a finite number of bits. This is often called round off error.

To answer the second question, one could obtain the memory address of a variable (the starting byte). Such address is also called a reference or a pointer (because it points to where the variable resides in memory). A pointer is simply an integer value. C++ provides a reference operator '&' for that purpose. For example:

```
#include <iostream>

using std::cout;

int main() {
  int x=2;
  cout<<''x is ``<<x<<''\n''; //value of x
  cout<<''x is located at ``<<&x<<''\n''; //memory address of x
}
```

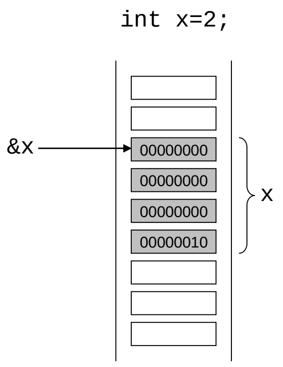


Figure 2.3: Pointer to x

One might need to remember the memory address of a variable. Therefore, pointers can be named and declared with the appropriate types. For example,

```
#include <iostream>

using std::cout;

int main() {
  int x=2;

  //declare a pointer named addr and
  //initialize it to point to variable x
  int * addr=&x;

  cout<<addr;
}
```

Moreover, a variable can be accessed through its pointer using the C++ dereferencing operator '*'. For example,

```
#include <iostream>

using std::cout;

int main() {
  int x=2;
  int * addr=&x;

  cout<<addr; //output the value of the pointer, i.e. x's address
  cout<<*addr; //output the value of x itself
}
```

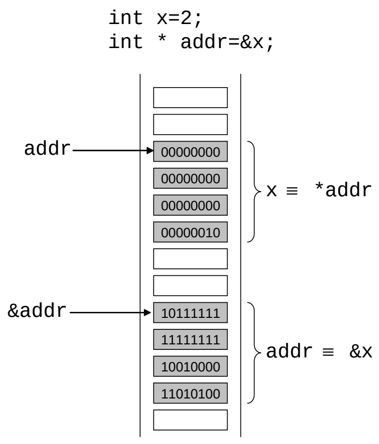


Figure 2.4: Accessing x through its pointer

&x is the address of/pointer to **x**
***p** is the variable stored in address/pointer **p**
x has type **T** ⇔ **&x** has type **T ***
p has type **T *** ⇔ ***p** has type **T**

To answer the third question, a variable lives in memory until the end of the block, i.e. it lives only in its scope. For now, this simple answer will be sufficient. Later on, and through the use of pointers, we will see how a variable can continue to live outside the scope in which it was declared (although we may loose the ability to access it).

Chapter 3

Functions

3.1 Introduction

So far, we have seen elements that must appear in any powerful programming language:

- numbers and arithmetic operations are primitive data and operations
- combinations provide means of combining operations
- declarations that associate names with values provide a limited means of abstraction

A function is a more powerful abstraction technique by which a compound operation can be given a name and then referred to as a unit. Let us consider the idea of "squaring". One might say "To obtain the square of something, return it multiplied by itself", which can be expressed mathematically as

$$square(x) = x * x$$

We can express this idea in C++ as follows:

```
 int        square(int  x) {  return x    *        x;}
  ↑           ↑           ↑       ↑     ↑    ↑          ↑
To obtain the square of something,  return  it multiplied by itself.
```

This declares a function with the name *square*. The function square represents the operation of multiplying something by itself. The thing to multiply is given a local name *x*. The scope of *x* is the body of the function defined by the block between { and }.

3.2 Function definition

In general, a function definition has the following syntax:

type name (*type1 param1*, *type2 param2*, ...) { *body* }

where

- type: represents the type of the result returned by the function, aka the return type, i.e. the type of the expression that appears after the **return** keyword.
- name: the name given to the function and by which it can be called.
- parameters: a list of names with their types used within the body of the function to refer to the corresponding arguments of the function.
- body: statements/operations that will eventually yield the value of the function when each parameter is replaced by it corresponding argument. The value is returned using the **return** keyword.

```
#include <iostream>

using std::cout;

int square(int x) {
  return x*x;
}

int main() {
  cout<<square(21); //21 is the argument corresponding to parameter x
}
```

The above program will output 441, i.e. the square of 21. The function square is called on its argument 21. This means that the parameter of the function corresponding to this argument, i.e. x, is assigned the value 21 in the body of the function (the scope of x). Since the function returns x*x, it returns 21*21 which is 441. There are two important things to note:

- the number of parameters and the number of arguments must be the same (1 in this case)
- the type of the parameter and the type of the argument must match (int in this case)

Here are more examples of using the function square:

```
int main() {
  cout<<square(2+5);
  cout<<square(square(3));
}
```

The above will output:

```
49
81
```

Note that since the argument of the function square is the result of a another call to square in the second statement, the return type of square and the type of its parameter must be the same (int in this case).

Once a function is defined, it can be used as a building block to define other functions. For example, let us define the function (C++ is case sensitive)

$$Pythagoras(x, y) = x^2 + y^2$$

```
#include <iostream>

using std::cout;

int Pythagoras(int x, int y) {
  return square(x)+square(y);
}

int main() {
  cout<<Pythagoras(3,4);
  //there must be 2 arguments to replace the 2 parameters
}
```

This will output 25, i.e the sum of squares of 3 and 4.

3.3 Function evaluation using substitution

How is a function evaluated in C++? The previous section illustrates some examples of how we obtain the result of a function. In general, the following procedure is performed repeatedly until no more evaluation is needed (this does not work when we have assignment):

```
repeat
    evaluate the arguments
    substitute parameters by values of arguments in body of function
until done
```

For this reason, C++ is said to use the call-by-value strategy, i.e. the argument is evaluated first, then the parameter is assigned that value. In other words, the parameter and the argument are two different variables in memory (even if they have the same name). They have different scopes. Consider the following function:

```
int f(int a) {
  return Pythagoras(a+1,a*2);
}
```

A call to f(5) will be evaluated as follows:

- Evaluate: the argument evaluates to 5
- Substitute: a is assigned the value 5 in its scope, the function evaluates to:

  ```
  Pythagoras(5+1,5*2)
  ```

- Evaluate: The arguments evaluate to 6 and 10
- Substitute: x and y are assigned the values 6 and 10 respectively in their scope, the function evaluates to:

  ```
  square(6)+square(10)
  ```

- Evaluate: The arguments evaluate to 6 and 10

- Substitute: x is assigned the value 6 in its scope, and another x is assigned the value 10 in its scope, the function evaluates to:

  ```
  6*6+10*10
  ```

- Evaluate: this evaluates to 136

3.4 Function as a black box

By now it should be apparent that every function acts like a module where only the input/output relationship is important. Moreover, a module can be used from within another module. For instance, in the above example, f uses Pythagoras, and Pythagoras uses square. From the point of view of f, Pythagoras is a black box that transforms an input (in this case two integers) to an output (in this case an integer). The internal operation of Pythagoras (represented by the body of the function) should not be of concern to f.

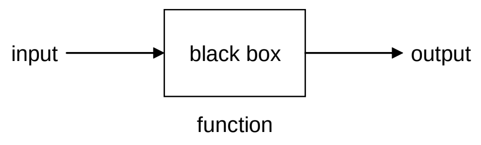


Figure 3.1: Function as a black box

Such a "black box" abstraction is desired. It would be really inconvenient if calling a function from within another would interfere with the operation of the function itself (although such behavior can be obtained if needed). That's why the names of the function parameters are **local** to the function itself, and the scope of the parameters is the body of the function. The following two definitions produce identical functions:

```
int square(int x) {return x*x;}

int square(int y) {return y*y;}
```

Therefore, the choice of names for the parameters should not matter. The following function must remain the same regardless of which definition for square is used.

```
int Pythagoras(int x, int y) {
  return square(x) + square(y);
}
```

Otherwise, the behavior of this function will be confusing. If the first definition for square is used, then what is being computed when square(y) is called from within Pythagoras? It is x*x or y*y? What if the second definition is used? To avoid such confusion, and to enforce the black box concept, names of parameters are only local to the function, and they only refer to the names used inside the body of the function.

Chapter 4

Testing and repetition

4.1 Introduction

The class of functions that we can define at this point is very limited because we have no way to make tests and to perform different operations based on the result of a test. Moreover, we did not cover how to define a process to be repeated several times within a function (although this is possible using recursive functions as we will see later). We cover these two issues here.

4.2 Testing

Without being able to perform a test, we cannot define a function to compute the absolute value of a number x, denoted mathematically by $|x|$.

$$|x| = \begin{cases} x & x > 0 \\ 0 & x = 0 \\ -x & x < 0 \end{cases}$$

C++ provides the **if** keyword for performing such case analysis. Generally, one could write the following:

if (*cond*) {*body*}

where

- *cond*: is a boolean expression (i.e. of type **bool**)
- *body*: is the body of the if statement which is performed if *cond* evaluates to **true**

Note that if the body contains a single statement, the open { and close } can be omitted. The flow chart in Figure 1 describes the operation of an if statement.

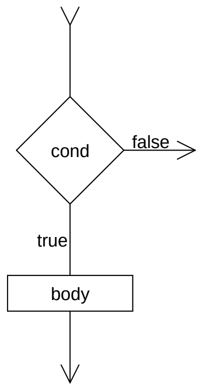


Figure 4.1: The if statement

4.2.1 Relational operators

One way to produce a boolean expression is by using the relational operators <, >, <=, >=, ==, and ! =. The following table lists the relational operators and describes their mathematical equivalence.

relational operator	mathematical equivalence
<	$<$
>	$>$
<=	$\leq$
>=	$\geq$
==	$=$ (equality)
! =	$\neq$

We can now define a function that takes a number and returns its absolute value:

```
float abs(float x) {
  if (x>0)
    return x;
  if (x<0)
    return -x;
  if (x==0)
    return 0;
}
```

Each of the above three conditions produce a boolean expression (i.e. of type **bool**). The value of the boolean expression depends on the value of x: If x is positive, then $x > 0$ is true, $x == 0$ is false, and $x < 0$ is false. If x is zero, then $x > 0$ is false, $x == 0$ is true, and $x < 0$ is false. If x is negative, then $x > 0$ is false, $x == 0$ is false, and $x < 0$ is true. Note that equality is denoted by the operator == (and not the assignment operator =). Testing for equality is a major source of logical errors as a programmer tends to replace the double == with the single =. Here's a famous example about the world's last C++ bug (we will see the while loop in the following section):

```
while (true) {
  int status = GetRadarInfo();
    if (status = 1)
      LaunchMissiles();
}
```

The value of an assignment is the value of the assigned variable after performing the assignment. Therefore, status=1 evaluates to 1, which is considered true when interpreted as boolean (any integer different than 0 is true). What happens if == is replaced by = in the abs function above?

4.2.2 Logical operators

We saw in the previous section how to produce simple boolean expressions using the relational operators. Each relational operator performs a single test. To combine multiple tests in one condition, C++ provides the logical operators AND, OR, and NOT.

logical operator	meaning
&&	logical AND
||	logical OR
!	logical NOT

The operands for logical operators must be of type **bool**. While both && and || are both binary operators (i.e. they require two operands each), ! is a unary operator that reverses the boolean value of its operand. The following tables summarize the operations of these three logical operators:

operand 1	operand 2	&&
false	false	false
false	true	false
true	false	false
true	true	true

operand 1	operand 2	||
false	false	false
false	true	true
true	false	true
true	true	true

operand	!
false	true
true	false

Example 1: Here's a function that tests whether a number x is within a given interval $[a, b]$.

```
bool inRange(int x, int a, int b) {
  if (x>=a && x<=b)
    return true;
  if (x<a || x>b)
    return false;
}
```

The function returns a boolean value depending on whether $x \in [a, b]$ or not. It is worth mentioning here that the first compound condition cannot be replaced by ($a <= x <= b$). While this is mathematically correct, it does not evaluate as one might expect in C++. The expression $a <= x <= b$ is evaluated left to right. Therefore, $a <= x$ is evaluated first, and depending on whether it evaluates to false or true, $a <= x$ is replaced by 0 or 1 respectively, yielding either the condition $0 <= b$ or the condition $1 <= b$ (which is not the programmers intention). It is also worth mentioning that C++ uses *short circuit* evaluation: if $x >= a$ evaluates to false, C++ does not continue evaluating the expression $x >= a$ && $<= b$ because the result is already determined to be false (see the operation of logical AND). This also holds if the expression $x < a$ evaluates to true in the second compound condition.

Note that the above function can be simplified as follows:

```
bool inRange(int x, int a, int b) {
  return (x>=a && x<=b);
}
```

Example 2: Here's a function that tests if two numbers x and y are close enough, i.e. if the absolute value of their difference is not bigger than a certain threshold.

```
bool close_enough(float x, float y) {
  return !(abs(x-y)>=0.0001);
}
```

In most cases, the programmer can avoid using the NOT operator by expressing the condition with an appropriate relational operator. For instance, the above expression can be replaced by ($abs(x - y) < 0.0001$). Theoretically speaking, ! can always be avoided because **!expression** is equivalent to **expression==false**. However, the latter form is more error prone (e.g. accidentally replacing == with =).

Note that the parentheses around the condition $abs(x - y) >= 0.0001$ are needed because the logical operator ! has a higher precedence than the relational operator >=. In general, it is a good idea to have parentheses around every condition to avoid errors due to precedence. The following tables updates that of Chapter 2 and shows the operators in decreasing order of precedence.

operator	evaluation	
()	left to right	
!	right to left	!!a is !(!a)
$*$ / %	left to right	
+ $-$	left to right	
< <= > >=	left to right	
== !=	left to right	
&&	left to right	
||	left to right	
=	right to left	a=b=c is a=(b=c)

4.2.3 Another form of if and the dangling else problem

C++ provides another form of if given by the following syntax:

if (*cond*) {*body1*} *else* {*body2*}

This is explained by the following flow chart:

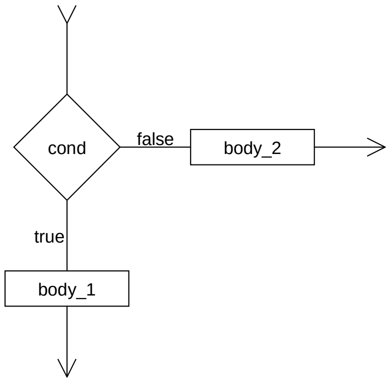


Figure 4.2: The if-else statement

If *cond* evaluates to true, then *body1* is performed; otherwise, *body2* is performed. For instance, we can re-write the abs function as follows:

```
float abs(float x) {
  if (x>=0)
    return x;
  else
   return -x;
}
```

The use of if-else may lead to what is referred to as the *dangling else* problem. Consider the following code:

```
if (lives>0)
  if (score>1000)
    addBonusLives();
else
  gameOver();
```

Obviously, for a human being, the intention of the programmer is the following: if the player has some lives left, then check the score, and if it is greater than 1000, add some bonus lives. If the player has no more lives, the game is over. However, the above code is indistinguishable from the code below for the C++ compiler (which causes a game over if the score is less than 1000).

```
if (lives>0)
  if (score>1000)
    addBonusLives();
  else
    gameOver();
```

Which one is it? Here's the rule: C++ matches an else with the closest unmatched if. Therefore, the above code is a programmer's mistake. To enforce to correct logic, we can use open { and close } as follows:

```
if (lives>0) {
  if (score>1000)
    addBonusLives();
}
else
  gameOver();
```

Another way of enforcing the correct logic is to force every if to have a corresponding else:

```
if (lives>0)
  if (score>1000)
    addBonusLives();
  else ; //empty statement
else
  gameOver();
```

4.3 Repetition

Let us revisit the process of adding numbers 1 through 100. We saw in Chapter 2 that this process can be described mentally as follows:

```
$x \leftarrow 0$
$y \leftarrow 1$
repeat
    $x \leftarrow x + y$
    $y \leftarrow y + 1$
until $y > 100$
```

Figure 4.3: Mental process for $\sum_{i=1}^{100} i$

Therefore, we argued that a programming language must provide means of repetition.

4.3.1 The while loop

C++ provides the **while** keyword for making repetition.

while (*cond*) {*body*}

This is similar to if, except that the *body* of the *while* loop is performed repeatedly as long as *cond* evaluates to true (as opposed to just once in if). Here's the flow chart for a while loop.

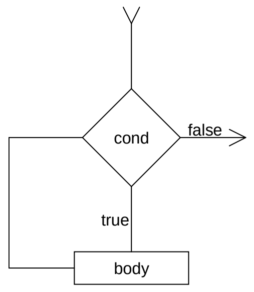


Figure 4.4: The while loop

We can now write our famous sum to 100 program in C++:

```
#include <iosteam>

using std::cout;

int main() {
  int x=0;
  int y=1;

  while (y<101) {
    x=x+y;
    y=y+1;
  }

  cout<<x;
}
```

4.3.2 The for loop

C++ provides another way of expressing repetition using the for loop:

for (*initialization*; *cond*; *update*) {*body*}

where

- *initialization*: is an initialization statement to initialize variable(s) and is performed first and only once. This can also be variable(s) declaration, in which case the scope of the declared variable(s) is the *body* of the for loop.
- *cond*: is a boolean expression (i.e. of type **bool**)
- *body*: is the body of the for loop which is performed as long as *cond* evaluates to **true**
- *update*: is an update statement that may change the value of *cond* and is performed after every performance of *body*.

Figure 5 shows the flow chart of the for loop.

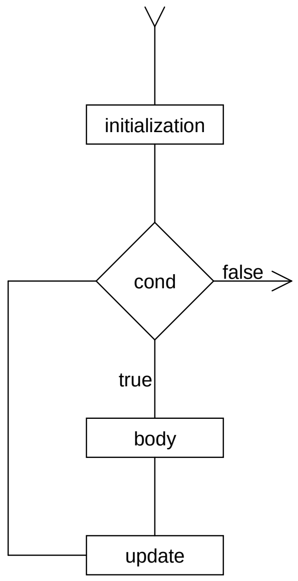


Figure 4.5: The for loop

Let's re-write the above code using a for loop.

```
#include <iostream>

using std::cout;

int main() {
  int x=0;

  for(int y=1; y<101; y=y+1)
    x=x+y; //body of for loop

  //y is not accessible in this scope
  cout<<x;
}
```

Here's another way:

```
#include <iostream>

using std::cout;

int main() {
  int x=0;
  int y; //y is declared but not initialized (usually bad)

  for(y=1; y<101; y=y+1)
    x=x+y; //body of for loop

  //y is accessible in this scope
  cout<<x;
}
```

And yet, here's another way:

```
#include <iostream>

using std::cout;

int main() {
  int x;
  int y;
  for(x=0,y=1; y<101; x=x+y,y=y+1)
    ; //body of for loop is empty

  cout<<x;
}
```

4.3.3 The do while loop

Yet another way to express repetition is the do while loop:

do {*body*} *while* (*cond*)

where

- *cond*: is a boolean expression (i.e. of type **bool**)
- *body*: is the body of the do while loop which is performed once first, and then as long as *cond* evaluates to **true**

Therefore, the body of the do while loop is performed at least once (before the condition is evaluated). The do while loop has the following flow chart:

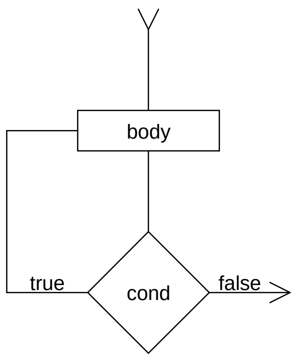


Figure 4.6: The do while loop

4.4 Exercise: computing the square root

Let's say we would like to compute the square root y of a number x, i.e. $y >= 0$ and $y^2 = x$. As we have seen in Chapter 1, to write a program for computing the square root, we must know how to do it ourselves. Luckily, Newton's method tells us the following:

If y is a guess for the square root of x, then

$$\frac{y + x/y}{2}$$

is a better guess.

Newton's method suggests the following repetative process.

- start with a guess y for the square root of x
- repeatedly update the guess using the above formula
- stop when y^2 and x are close enough

Example: Let's compute the square root of 2, starting with 1 as a guess.

$$\frac{1 + 2/1}{2} = 1.5$$
$$\frac{1.5 + 2/1.5}{2} = 1.4167$$
$$\frac{1.4167 + 2/1.4167}{2} = 1.4142$$
$$\cdots$$

Here's a function called sqrt that returns an estimate of the square root of x, given an initial guess:

```
float sqrt(float x, float guess) {
  while (!close_enough(guess*guess, x))
    guess=(guess+x/guess)/2;
  return guess;
}
```

There are some issues that need to be discussed here, namely the possibilities that $x < 0$ or $guess \leq 0$. If $x < 0$, then the square root of x does not exist and the guess will never converge. Similarly, if we start with $guess = 0$, then the next guess will be infinity and will never converge. If we start with a negative guess, then we converge to a negative root (not the definition of square root). All these issues can be solved by adding the following statements before the while loop:

```
x=abs(x); //make x positive
guess=abs(guess); //make guess positive
guess=guess+0.0000001; //avoid guess being zero
```

Chapter 5

Handling multiple values

5.1 Introduction

Consider the problem of working with rational numbers of the form a/b where both a and b are integers. For instance, we would like to add two rational numbers and obtain the result in fractional form. If we want to define a function for performing this addition, then the function must return two integers: the numerator and the denominator of the result. Adding a/b and c/d requires passing four integers as parameters to the function.; however, each function can specify only one type for its return.

```
___?___ addRat(int a, int b, int c, int d) {

    . . .

    return ___?___;
}
```

How can we make such function return two integers? In general, how can we define a function that returns multiple values? The immediate answer is that we can't! Instead, we make the function store the numerator and the denominator of the result in two variables that are specified in advance. Let's call these two variables *numer* and *denom*. For this approach to work, two issues must be considered:

- Both numer and denom must be declared in a scope that contains the function definition; otherwise, numer and denom cannot be accessed from within the function.
- There is no need for the function to return any value.

We can declare numer and denom in the global scope, i.e. not inside any block. Such variables are called *global variables*. It is generally not a good idea to have global variables because they can be accessed from anywhere in the program and, therefore, it is hard to keep track of which parts of the program are affecting their values. To specify that a function does not return any value, we make it return the special type **void** [1].

[1] We could still make the function return something that we will ignore.

Here's an example:

```
#include <iostream>

using std::cout;

int numer;
int denom;

void addRat(int a, int b, int c, int d) {
  //note: a/b + c/d = (ad + bc)/bd
  numer=a*d+b*c;
  denom=b*d;
}

int main() {
  int a=2;
  int b=3;
  int c=4;
  int d=5;
  addRat(a,b,c,d);
  cout<<numer<<''/''<<denom<<''\n'';
}
```

Besides the fact that global variables are not desired, there is a potential problem with the above approach. Since the result is always stored in the global variables numer and denom, we can only compute one number at a time. Every call to the function addRat will change the values of numer and denom. Therefore, one has to save those values in other variables before issuing another call to the function addRat; otherwise, the last result will be overridden. A better approach is described below.

5.2 Passing by reference using pointers

A better approach for the above problem is to "tell" the function where to store the result. This can be achieved by providing the function with the memory addresses of the variables that must hold the result. The function then stores the result into the desired memory addresses. Therefore, we have to consider the following:

- The only way to "tell" a function something is to pass the information in a parameter. Therefore, since every parameter must have a type, we have to figure out what types to use for the new parameters.
- We have to be able to "write" to a specific memory address.

```
void addRat(int a, int b, int c, int d, type1 addr1, type2 addr2) {

  //write result to addr1 and addr2

}
```

Luckily, we should be able to deal with the above issues by reviewing material from Chapter 2.

Since the numerator and the denominator of the result are both integers, their memory addresses will have the type **int ***, i.e. a pointer to an int. Moreover, given a pointer, we can access its content using the dereferencing operator *. Therefore, our addRat function will be the following:

```
void addRat(int a, int b, int c, int d,
            int * numer, int * denom) {

  //numer is a pointer to the numerator
  //let's dereference it to set its content
  *numer=a*d+b*c;

  //denom is a pointer to the denominator
  //let's dereference it to set its content
  *denom=b*d;
}
```

Here's an example on how to use the function:

```
int main() {
  int a=2;
  int b=3;
  int c=4;
  int d=5;

  //declare variables to hold the result
  int x;
  int y;

  //provide memory addresses of x and y
  //using the referencing operator &
  addRat(a,b,c,d,&x,&y);

  cout<<x<<''/''<<y<<''\n'';
}
```

In the above example, x and y are said to be passed by reference using pointers. Note that the usual pass by value technique does not correctly solve the problem. For instance, consider the following:

```
#include <iostream>

using std::cout;

void addRat(int a, int b, int c, int d,
            int numer, int denom) {
  numer=a*d+b*c;
  denom=b*d;
}
```

```
int main() {
  int a=2;
  int b=3;
  int c=4;
  int d=5;

  //declare variables to hold the result
  int x;
  int y;

  addRat(a,b,c,d,x,y);

  cout<<x<<''/''<<y<<''\n'';
}
```

The variables x and y will not change after calling the function addRat. This is because the values of x and y are simply stored into the variables numer and denom which are local to the function. The scope of numer and denom is the body of the function and, therefore, any change to these variables is only visible within the body of the function. This holds even if x and y are renamed numer and denom respectively.

Note that passing a variable by reference using a pointer is the same as passing its address by value. For instance, the parameters numer and denom of type **int** * are also local to the function, except that they are pointers. Therefore, chaning numer or denom inside addRat does not have any effect outside the function. Here's another example:

```
void f(int * ptr) {
  *ptr=2;
  ptr=10000;
}

int main() {
  int x=1;
  int * p=&x; //p points to x
  f(p);
  cout<<x<<''\n''; //x changed to 2
  cout<<p<<''\n''; //p unchanged
}
```

When function f is called, ptr takes the value of p (by substitution). Therefore, ptr local to f) and p are now two different pointers with the same value. Modifying *ptr or *p will have the same effect on x (both point to the same memory address). However, modifying ptr will only modify the value of ptr itself, i.e. ptr will simply point to another address.

5.3 Arrays

Consider the following simple task: define a function that computes the average of two integers. Of course, this should be trivial by now:

```
float average(int a, int b) {
  return (a+b)/2;
}
```

While this is not the point of the exercise, the above function contains an error. Since a, b, and 2 are all integers, (a+b)/2 is interpreted as integer division and, therefore, will produce an integer, i.e. truncated result (the integer is then converted to the return type float). To solve this problem, we may re-write the function as follows:

```
float average(int a, int b) {
  return (a+b)/2.0;
}
```

Now let us define a function that computes the average of three integers:

```
float average(int a, int b, int c) {
  return (a+b+c)/3.0;
}
```

While both functions have the same name, they can coexist in the same scope. The compiler can distinguish between them because they have different parameter lists. A call to average can be resolved without ambiguity by simply counting the number of arguments supplied to the function. This is called function overloading. In general, two functions in the same scope can have the same name if they differ in their parameter lists:

- either by number of parameters
- or by types of parameters
- or both, of course

Note that the return type is not considered for function overloading.

Now let us define a function that computes the average of four integers:

```
float average(int a, int b, int c, int d) {
  return (a+b+c+d)/4.0;
}
```

By now, it should be clear what we are getting into. Although the concept of average is unique, it seems that we have to define a separate function for every possible number of integers. That would be really dramatic!

We need a way to represent a collection of integers as one entity, and pass it to the function. C++ provides a way to create such entities using arrays. An array is declared as follows:

$$\textit{type name}[\textit{size}];$$

where

- *type*: is the type of each element in the array (arrays are homogeneous)
- *name*: is the name of the array
- *size*: is the size of the array

For instance, one could declare an array of eight integers by writing:

```
int a[8];
```

The different elements (integers) of this array can be referred to as:

```
a[0], a[1], ..., a[7].
```

They are stored in consecutive locations in memory, each occupying a certain number of bytes (e.g. 4 for int).

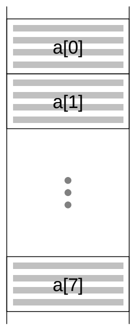


Figure 5.1: Elements of array are stored consecutively

Since the array a must be passed as one parameter to the average function, we must ask ourselves the following question: what is the type of a? We know that a[i] has type **int** for $i = 0 \ldots 7$. But what is the type of a itself? The answer is: **int ***. This means that a is a pointer to int. So what is a really?

5.3.1 Arrays and pointers

An array a is actually a pointer to (memory address of) the first element of the array a. Therefore,

```
T a[size];
```

creates a variable a of type T *. Moreover, a will be pointing to a[0]. Therefore, a[0] is just a syntactic sugar for *a. Similarly, a is simply &a[0]. In general:

a[i] is equivalent to *(a+i)
a+i is equivalent to &a[i]

How come (a+i) points to the i^{th} element of the array if each element occupies a number of bytes? The reason for this is that C++ takes care of this kind of pointer arithmetics. Since the type of the elements is known, incrementing the pointer by 1 increments it by the appropriate number of bytes. However, if we force C++ to interpret the pointer as an integer using casting, we will observe a different behavior.

```
int main() {
  int * a;
  int * b=a+1;
  cout<<b-a;  //1
  cout<<(int)b-(int)a; //4, int takes 4 bytes
}
```

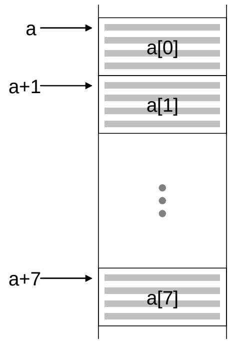


Figure 5.2: Array as pointer

Now that we know the type of an array, we can pass it as a parameter to the average function. Before we define our general purpose average function, we have to mention one technical detail: In general, there is no way of telling how large the array is from the array itself. After all, an array is just a pointer to the first element. Therefore, we must also pass the size of the array when calling the function. The size is known by the programmer at this point from the declaration.

Given that the parameters of the average function are a and size denoting the array and its size respectively, the average function has to simply add the elements a[0], ..., a[size-1], and then divide the obtained sum by size. This can be achieved by:

- starting with a sum of 0
- setting up a loop and keeping track of a count
- repeatedly adding a[count] to sum
- until count > size-1
- return sum divided by size

Therefore, we need two variables to be declared inside the function to maintain the sum and to keep track of the count.

```
float average(int * a, int size) {
  int sum=0;
  int count=0;
  while (count<size) {
    sum=sum+a[count];
    count=count+1;
  }
  return (float)sum/size;
}
```

Here's a variation using a for loop:

```
float average(int * a, int size) {
  int sum=0;
  for(int count=0; count<size; count=count+1)
    sum=sum+a[count];
  return (float)sum/size;
}
```

Note the use of casting in the return statement to avoid the integer division. Alternatively, sum could have been declared as float.

Another way of specifying an array as a parameter to the function is the following form (which will be useful later when we consider multi-dimensional arrays):

```
float average(int a[], int size) {
  int sum=0;
  for(int count=0; count<size; count=count+1)
    sum=sum+a[count];
  return (float)sum/size;
}
```

In any case, here's how the average function can now be used:

```
int main() {
  int a[8];
  a[0]=2;
  a[1]=8;
  a[2]=7;
  a[3]=1;
  a[4]=3;
  a[5]=5;
  a[6]=6;
  a[7]=4;
  cout<<average(a,8);
}
```

An alternative way of initializing the array is the following:

```
int main() {
  int a[8]={2,8,7,1,3,5,6,4};
  cout<<average(a,8);
}
```

5.3.2 Dynamic memory allocation

Most compilers (e.g. not g++) do not accept the declaration of an array unless the size of the array is known at **compile time**. This has to do with the ability of the compiler to create the function in memory. Therefore, the following program will not compile on many compilers:

```
#include <iostream>

using std::cin;

int main() {
  int n;
  cin>>n;
  int a[n];

  . . .
}
```

Like **cout** which is the standard output stream (outputs to the screen), **cin** is the standard input stream which reads information from the keyboard. The operator >> is the one used in conjunction with input streams and the statement **cin>>n** extract information from the stream (in this case the keyboard) and stores it in variable n. Obviously, the intention of the programmer was to:

- get information from the user (keyboard)
- store that into variable n (a number)
- declare an array of integers of size n

However, since the value of n is not known at compile time (it depends on what the user will input), the compiler will complain. This is the case even with the following program:

```
int main() {
  int n=8;
  int a[n];

  . . .
}
```

The compiler has no way of telling whether n will have the value 8 at the point of declaring the array. Obviously it will, but the compiler has no such intelligence. For instance, consider the following wicked scenario:

```
void f(int * x) {
  *x=*x*2;
}

int main() {
  int n=8;
  f(&n);
  int a[n];

  . . .
}
```

Obviously the compiler has to develop some intelligence to figure out that the value of variable n is doubling. To avoid this necessity, the compiler simply complains. To make the compiler happy, we need to give it some assurance.

```
int main() {
  const int n=8;
  int a[n];

  . . .
}
```

The keyword **const** declares n as a constant variable. Variable n cannot change. Any attempt to use the assignment operator on n will produce a compilation error. Therefore, in this case the compiler knows what n is at compile time and accepts the array declaration. Note that since a constant variable cannot be assigned, declaring a constant variable without initialization is an error.

```
int main() {
  const int n; //error

  . . .
}
```

The use of **const** is very tricky and elaborate in C++, but for now the above will be enough for our purpose. Coming back to our problem, it seems that the only way to declare our array appropriately is the following:

```
#include <iostream>

using std::cin;

int main() {
  const int max_size=100;
  int a[max_size];
  int n;
  cin>>n;
  if (n>max_size) {
    //do the hard work
  }

  //think of a as an array of size n

  . . .
}
```

There are two problems associated with the above approach:

- What is a large enough value for max_size?
- What if n is still larger than max_size?
- What if n is too small (wasting memory)?

A better solution is for the programmer to allocate enough memory for the array himself. This means that the compiler will not worry about the size of the array at compile time, but the array will be created at run time in a special part of the memory, called the *free store*. The downside is that in doing so, the compiler will not be able to free the allocated memory when the array goes out of scope. It becomes the responsibility of the programmer to do so. Memory allocated on the free store persists until it is explicitly freed. Failing to do so will result in what is known as *memory leak*. While this may not be a severe problem because memory is released when the program ends, a continuously running program that repeatedly allocates but fails to free memory may cause the system performance to deteriorate. C++ provides the two operators **new** to allocate memory and **delete** to free memory.

```
#include <iostream>

using std::cin;

int main() {
  int n;
  cin>>n;

  int * a=new int[n];
  //a is now an array of n integers

  . . .

  //free the memory
  delete[] a;
}
```

Since an array is a pointer to the first element, the statement

int * a=new int[n]

declares a pointer to int and assigns it to an address in memory where a space for n integers is allocated.

It is also possible to dynamically allocate memory for one element:

```
int main() {

  . . .

  int * a=new int;

  . . .

  delete a;
}
```

We will see why this is useful later in the course when we talk about classes and inheritance. In the meantime, remember to always delete[ ] what you new[ ] and delete what you new, but never mix them.

One last word about dynamic memory allocation. It might seem that using a compiler like g++, that does not require the size of the array to be known at compile time, can avoid the use of dynamic memory allocation. However, we will see later that sometimes the size of the array is not known (not even through a variable name) to the programmer at the point of declaration.

Chapter 6

Sorting numbers: putting it all together

6.1 Introduction

In this chapter we will try to put everything we have learned so far in a useful application: Sorting numbers. This application illustrates the concepts of arrays, dynamic memory allocation, random number generator, loops, passing by value, and passing by reference.

6.2 Declaring an array of unknown size and initializing it randomly

```
#include <iostream> //needed for cout and cin
#include <cstdlib>  //needed for rand()

using std::cout;
using std::cin;

int main() {
  cout<<''enter the size of the array:'';
  int n;
  cin>>n;
  int * list=new int[n];
  for (int i=0; i<n; i=i+1)
    list[i]=rand()%100;

  . . .

  delete[] list;
}
```

An alternative is to make the initialization a function on its own as follows:

```
void init(int * a, int n) {
  for (int i=0; i<n; i=i+1)
    a[i]=rand()%100;
}

#include <iostream> //needed for cout and cin
#include <cstdlib>  //needed for rand()

using std::cout;
using std::cin;

int main() {
  cout<<''enter the size of the array:'';
  int n;
  cin>>n;
  int * list=new int[n];
  init(list,n);

  . . .

  delete[] list;
}
```

One subtle point here is whether this change in the array will be visible outside the function. We know that **a** is a local name for function **init** and, therefore, must have the body of the function as its scope. Does a change in **a[i]** affect **list[i]**? The answer is yes, because **list** is passed by reference using a pointer (recall that an array is simply a pointer to its first element). To further emphasize this, we restate it in bold: **A C++ array is always passed by reference**. As we discussed before, **a[i]** is nothing but a syntactic sugar for ***(a+i)**, i.e. increment the pointer by i locations in memory and then dereference it. Therefore, **a[i]= . . . ;** is equivalent to ***(a+i)= . . . ;**. As a result, the content of the memory is altered inside the function and the effect is visible to the outside. On the other hand, n is passed by value, so any attempt to change n inside function **init** will not be visible to the outside.

6.3 Finding the index of the smallest number

Here's a function that returns the index of the smallest number of an array passed as a parameter.

```
int minimum(int * a, int n) {
  int index=0;
  for (int i=0; i<n; i=i+1)
    if (a[i]<a[index])
      index=i;
  return index;
}
```

The index of the smallest element is set to zero initially. A loop is used to check every element of the array. Every time a smaller element is discovered, the index is updated. Finally, the index is returned. An alternative way of defining the function minimum is to make it return the index of the smallest element of a sub-array, given by its starting and ending positions.

```
int minimum(int * a, int start, int end) {
  int index=start;
  for (int i=start; i<=end; i=i+1)
    if (a[i]<a[index])
      index=i;
  return index;
}
```

This version of the function may seem more general. Well, conceptually it is; for instance, **minimum(a, 0, n-1)** emulates **minimum(a, n)**. However, **minimum(a, i, j)** can be achieved using **minimum(a+i, j-i+1)** (why?). Therefore, both have the same power.

6.4 Swapping two elements

The following is a function that takes an array as a parameter, as well as two integers i and j, and swaps $a[i]$ and $a[j]$. Note that this works because a C++ array is always passed by reference (it is a pointer). Therefore, a change to the array is always visible outside the function.

```
void swap(int * a, int i, int j) {
  int temp=a[i];
  a[i]=a[j];
  a[j]=temp;
}
```

The function first saves the value of $a[i]$ in a temporary local variable called **temp**. Then assigns $a[i]$ the value stored in $a[j]$. Finally, assigns $a[j]$ the value stored in **temp** (the original value in $a[i]$).

6.5 Sorting elements

We can now use what we have developed so far as building blocks to define a function that sorts an array. Two variations discussed in class are shown below. Both are based on the idea of repeatedly finding the smallest element, swapping it with the first element, and ignoring the first element of the array.

1. find the smallest element in the array
2. swap it with the first element
3. repeat with the first element removed

Here's an example:

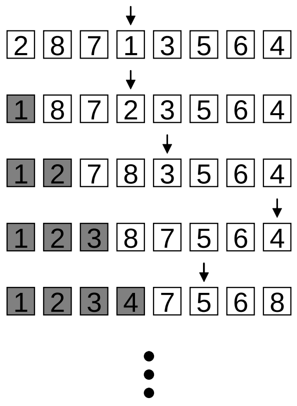


Figure 6.1: Example of sorting the array {2,8,7,1,3,5,6,4}

```
void sort1(int * a, int n) {
  for (int i=0; i<n; i=i+1)
    //swap a[i] and a[minimum(a,i,n-1)]
    swap(a, i, minimum(a, i, n-1));
}

void sort2(int * a, int n) {
  for (int i=0; i<n; i=i+1)
    swap(a+i, 0, minimum(a+i, n-i));
}
```

What may be surprising is that our function to sort an array turned out to be really small. This is because we relied on building blocks that we developed earlier. In general, it is always a good idea to break the problem into smaller ones and identify abstract concepts that can be tackled separately. For instance, sorting definitely requires some kind of swapping because the elements in the array must change places. Moreover, finding the smallest elements is a useful concept on its own. Sorting can be implemented in terms of these two concepts.

Chapter 7

Abstraction with data: the class

7.1 Introduction

Let us revisit the problem of working with rational numbers of the form a/b where both a and b are integers. Previously, we handled this problem by explicitly keeping track of two integers for each rational number. Moreover, we had to use passing by reference to compensate for the fact that a function cannot return two *things*. For instance, a function to add two rational numbers would have to be defined as follows:

```
void addRat(int a, int b, int c, int d,
            int * numer_addr, int * denom_addr) {

  //store numer and denom of a/b + c/d
  //in the specified memory addresses

  *numer_addr=a*d+b*c;
  *denom_addr=b*d;
}
```

Let us now *imagine* a better way. It would be very nice if we could handle rational numbers more conveniently as follows:

```
int main() {
  Rat x=Rat(2,3);
  cout<<''x is:''<<numer(x)/denom(x);
}
```

In the above *imaginary* piece of code, we assumed the following:

- **Rat** is a type (stands for rational number)
- **Rat(n,d)** *magically* constructs a rational number whose numerator is the integer n and whose denominator is the integer d
- **numer(x)** returns the numerator of a rational number x
- **denom(x)** returns the denominator of a rational number x

In our *imaginary* code, we declare a variable of type **Rat** whose name is x, and initialize it with a value given by the result of **Rat(2,3)**, which *magically* constructs the rational number 2/3. Then we output the value of x using **numer(x)** and **denom(x)** which return the numerator and denominator of the rational number x respectively.

What we have done so far is some wishful thinking. But if we assume that this *magic* is given to us, then we can define the addRat function in a much convenient way:

```
Rat addRat(Rat x, Rat y) {
  Rat z=Rat(numer(x)*denom(y)+numer(y)*denom(x),
            denom(x)*denom(y));
  return z;
}
```

Or simply:

```
Rat addRat(Rat x, Rat y) {
  return Rat(numer(x)*denom(y)+numer(y)*denom(x),
             denom(x)*denom(y));
}
```

This is a function that takes two parameters of type Rat and returns a Rat. The function uses Rat(n,d) to construct a rational number whose numerator and denominator are given by the expressions

numer(x)*denom(y)+numer(y)*denom(x)

and

denom(x)*denom(y)

respectively. Therefore, it returns what is conceptually interpreted as the sum of two rational numbers.

The addRat function can now be used as follows:

```
int main() {
  Rat x=Rat(2,3);
  Rat y=Rat(4,5);
  Rat z=addRat(x,y);
  cout<<''z is:''<<numer(z)<<''/''<<denom(z);
}
```

This represents another powerful technique for abstraction: the abstraction with data. In deed what we were missing is the abstract notion of a rational number. But it all relies on the *magic* given by **Rat**, **Rat(n,d)**, **numer(x)** and **denom(x)**. Fortunately, C++ provides a way to do the *magic* with enough infrastructure to create our own Abstract Data Types (ADTs). This is done using what is known as a **class**.

7.2 The class

The following represents the Rat class:

```
class Rat {

 public:
  int numer;
  int denom;

  Rat(int n, int d) { . . . }
};
```

In general

$$\textit{class Name}\ \{\ .\ .\ .\};$$

creates a new type with the name *Name*. That's why we often refer to a C++ class rather than a C++ type, but we can use both terms interchangeably. Let's dissect the above code one line at a time:

- **class** is a keyword in C++ and is used to name a new type, e.g. **Rat** in this case. This is followed by **{** to start the class definition
- **public** is also a keyword in C++ and signifies that what follows is public and, therefore, accessible outside the class. We will see the importance of this later on.
- **int numer;** and **int denom;** declare two variables of type int that will hold the values for the numerator and denominator respectively. Both numer and denom are called *member data* of the class. Note that numer and denom are declared but not initialized (initializing them would give an error, see below).
- **Rat(int n, int d) { . . . }** is the constructor for the class. Therefore, the constructor is simply a function that returns nothing (not even void). This is because it implicitly returns the *thing* that it constructs, which has a predetermined type, in this case **Rat**. Note also that the name of this function (the constructor) is the same as the name of the class. In deed, it is because of this naming convention that a function is recognized as the constructor for the class. The constructor must perform the appropriate initialization for member data.
- **};** ends the class definition. Note that ';' at the end is required.

Let us implement the constructor. To construct a rational number, we must initialize its numerator and denominator to the desired values passed as arguments to the constructor. Therefore, our constructor will be the following:

```
Rat(int n, int d) {
  numer=n;
  denom=d;
}
```

This completes our class definition:

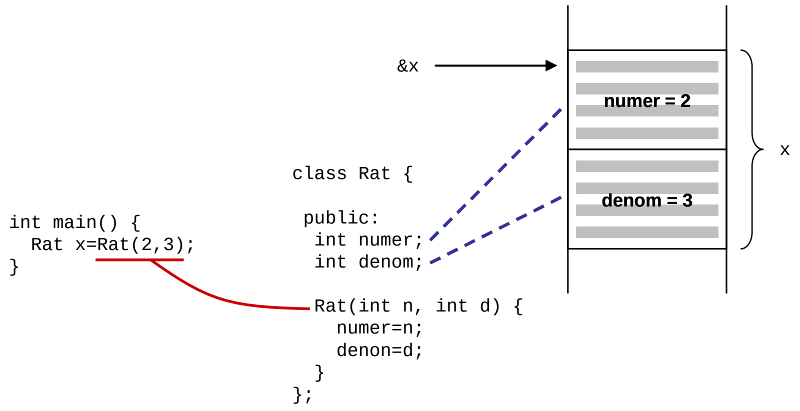
```
class Rat {

 public:
  int numer;
  int denom;

  Rat(int n, int d) {
    numer=n;
    denom=d;
  }
};
```

7.3 Scope and dot notation

There is a subtle point about scope that is worth mentioning here: what is the scope of the member data? Well, as usual, it is the block in which they are first declared. This means the entire class. Therefore, accessing them from within the constructor should not be surprising (and is needed for initialization). However, because they were declared as **public**, we can also access them outside the class. This is not an exception to the scoping rules if we have a good understanding of what really happens when we declare a variable of the newly created type (or class). So let's investigate the following line of code:

```
Rat x=Rat(2,3);
```

This declares a variable of type Rat whose name is x [1]. The variable x is also initialized to the rational number 2/3 (that's the result of calling the constructor). Therefore, we think of x as having a numerator equal to 2 and a denominator equal to 3. But we argued in previous chapters that when a variable is declared, it is created in memory. So how is that rational number with a numerator of 2 and a denominator of 3 is represented in memory? The following figure illustrates the idea (although it is not completely accurate):



```
int main() {
  Rat x=Rat(2,3);
}
```

```
class Rat {

 public:
  int numer;
  int denom;

  Rat(int n, int d) {
    numer=n;
    denon=d;
  }
};
```

Figure 7.1: Object x is an instance of class Rat

The class provides a recipe for rational numbers. Every rational number must contain two integers (called numer and denom), and that's how it is represented in memory. Both numer and denom are part of the representation.

[1]We also say that x is an object of class Rat or an instance of class Rat.

The constructor (called upon initialization of the object) performs the required initialization for each member datum (numer and denom in this case).

In general, every member datum is part of the representation of the object. Therefore, as long as the object is accessible, all member data are theoretically accessible (they are in memory). Back to our example, both numer and denom are accessible from within the scope of x, but only through x. In deed C++ provides a way to **access the member through the object** using the dot notation. The following syntax

object.member

provides access to the member through the object. Therefore, x.numer refers to the numerator of x, and x.denom refers to the denominator of x. C++ permits this kind of access for a member datum only when the member datum is declared **public**. Here's an example:

```
int main() {
  Rat x=Rat(2,3);
  cout<<''x is ''<<x.numer<<''/''<<x.denom;
}
```

We are now very close to finishing our *magic*. We still have to implement the functions numer(x) and denom(x). Now that we know how to access the public member data, it is easy to define these two functions.

```
int numer(Rat x) {
  return x.numer;
}

int denom(Rat x) {
  return x.denom;
}
```

Here's the complete work:

```
class Rat {

 public:
  int numer;
  int denom;

  Rat(int n, int d) {
    numer=n;
    denom=d;
  }
};

int numer(Rat x) {
  return x.numer;
}

int denom(Rat x) {
  return x.denom;
}
```

7.4 Restricting access

Assume that we would like (for one reason or another) our rational numbers to be reduced to their lowest terms. For instance, 6/9 can be reduced to 2/3. Obviously, we cannot rely on the user to maintain this property. Consider the following code:

```
int main() {
  Rat x=Rat(1,3); //x is 1/3
  Rat y=addRat(x,x); //y must be 2/3
  cout<<''y is:''<<numer(y)<<''/''<<denom(y);
}
```

The above program will output 6/9 (why?). One possible way to solve this problem is by changing the constructor to always initialize the numerator and denominator to their reduced forms. To do that, it is enough to divide both numer and denom by their greatest common divisor. The greatest common divisor can be obtained using Euclid's algorithm (you can search for this online if you have not heard of it).

```
Rat(int n, int d) {
  int z=gcd(n,d); //assuming the gcd function exists
  numer=n/z;
  denom=d/z;
}
```

While this ensures that every rational number is in its reduced form upon construction, it does not protect against direct access to the member data. Consider the following code:

```
int main() {
  Rat x=Rat(1,3);
  x.numer=2; //now x is 2/3
  x.denom=4; //now x is 2/4 (not reduced)
}
```

Note that the above is possible because numer and denom are declared public in class Rat (i.e. each is a member datum that is accessible through the object using the dot notation). To remedy this problem, one must prevent this type of access by moving numer and denom from the public section:

```
class Rat {

  int numer;
  int denom;

 public:
  Rat(int n, int d) {
    numer=n;
    denom=d;
  }
};
```

Both numer and denom are now called **private** member data. As a result, our numer(x) and denom(x) functions will now fail:

```
int numer(Rat x) {
  return x.numer; //error, numer is not public in Rat
}

int denom(Rat x) {
  return x.denom; //error, denom is not public in Rat
}
```

Since access has been restricted by making numer and denom private in class Rat, the only way to access numer and denom is now from within the class. This means our functions numer(x) and denom(x) must be moved inside the class. They will become **member functions**. Before we actually perform this move, we stop and ask ourselves the following question: what is really meant by data? Is rational number a data? It seems that we are eventually reaching a point where a rational number is not only composed of two integers, but also consists of a bunch of functions!

We may think of data as defined by some collection of functions (including constructors), together with specified conditions that these functions must fulfill in order to be a valid representation. For instance, our rational numbers satisfy the following condition: if x is constructed using Rat(n,d) then the result of the division numer(x)/denom(x) is equal to n/d (note that this is true regardless of which implementation of the constructor we use: reduced vs. non-reduced). Therefore, implementation details are not important. What is important is the interface with the outside world.

Chapter 8

More on data abstraction

8.1 Introduction

In the previous chapter, we learned how to restrict access to data members by making them private. If not declared under the keyword **public**, member data (and functions) in a class are private by default, but we may also use the keyword **private** as a reminder.

```
class Rat {

 private:
  int numer;
  int denom;

 public:
  Rat(int n, int d) {
    numer=n/gcd(n,d);
    denom=d/gcd(n,d);
  }
};
```

This approach ensures that the internal representation of a rational number is *protected*. The only way of constructing a rational number is through the constructor **Rat(n,d)**, which guarantees that **numer** and **denom** are reduced to their lowest terms. After that point, the programmer has no direct access to **numer** and **denom** using the dot notation (because they are declared private in the class). Unfortunately, this also means that the following two functions will fail:

```
int number(Rat x) {
  return x.numer; //error, numer is declared private
}

int denom(Rat x) {
  return x.denom; //error, denom is declared private
}
```

The only way to access **numer** and **denom** is now from within the class. This means the above two functions **numer(x)** and **denom(x)** must be moved inside the class. They become *member functions*.

8.2 Member functions

A class can contain functions other than a constructor. Unlike a constructor, these functions may have a return type. The scope of a member function is the entire class; therefore, a member function can be called from within another. Moreover, if these functions are declared public, they may be accessed outside the class but only through the object, using the dot notation (like member data) [1]. As we would expect, a member function has full access to all member data. Here's our first attempt to make **numer(x)** and **denom(x)** two member functions:

```
class Rat {

 private:
  int numer;
  int denom;

 public:
  Rat(int n, int d) {
    numer=n/gcd(n,d);
    denom=d/gcd(n,d);
  }

  int numer(Rat x) {
    return x.numer;
  }

  int denom(Rat x) {
    return x.denom;
  }
};
```

8.2.1 Naming conflict

By doing so, we have introduced a naming conflict. It just happened that the names of our two functions coincide with those given originally to the member data. For instance, the name **numer** is now declared twice in the same scope: once as an **int** and once as a function. Similarly for the name **denom**. This will cause a compile time error. To fix this problem, we can simply rename our functions.

```
class Rat {

 private:
  int numer;
  int denom;
```

[1] This is not true for a constructor. Once the object is constructed, we cannot explicitly invoke the constructor on it again using the dot notation.

```
 public:
  Rat(int n, int d) {
    numer=n/gcd(n,d);
    denom=d/gcd(n,d);
  }

  int getNumer(Rat x) {
    return x.numer;
  }

  int getDenom(Rat x) {
    return x.denom;
  }
};
```

8.2.2 Shadowing

In the event that we like to keep the original names of our functions (and this is justifiable because they are public while the member data are private), then we would rename the member data instead. For instance, we can replace **numer** and **denom** with **n** and **d** respectively for every occurrence of these names.

```
class Rat {

 private:
  int n;
  int d;

 public:
  Rat(int n, int d) {
    n=n/gcd(n,d);
    d=d/gcd(n,d);
  }

  int numer(Rat x) {
    return x.n;
  }

  int denom(Rat x) {
    return x.d;
  }
};
```

By doing so, we have created another error. This time, however, it is a logical error and, therefore, the compiler will not complain. The error is that the local names given to the parameters of the constructor are also **n** and **d**. The scope of these two parameters is the body of the constructor. As a result, every occurrence of **n** and **d** in the body of the constructor refers to those parameters. This prevents us from being able to refer to the member data from within the constructor. Such behavior is known as *shadowing*. The local names for the parameters shadow the names for the member data. In general shadowing occurs when a variable in an inner scope is given the same name as a variable in an outer scope. Therefore, a name refers to the declaration found in the inner most scope that contains such declaration.

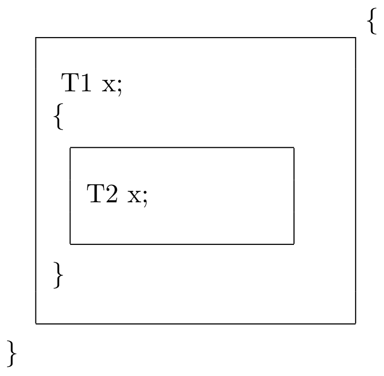


Figure 8.1: Shadowing: in the inner scope, x refers to the second declaration

To solve the shadowing problem, we can simply rename the parameters (we will see another way later).

```
class Rat {

 private:
  int n;
  int d;

 public:
  Rat(int a, int b) {
    n=a/gcd(a,b);
    d=b/gcd(a,b);
  }

  int numer(Rat x) {
    return x.n;
  }

  int denom(Rat x) {
    return x.d;
  }
};
```

Our class is now almost ready. However, since public member functions must be accessed through the object using the dot notation, the syntax for using our two newly added functions will look a bit weird. Here's an example:

```
int main() {
  Rat x=Rat(2,3);
  cout<<x.numer(x)<<''/''<<x.denom(x)<<''\n'';
}
```

The variable x appears twice with each function call: once to access the function itself using the dot notation, and once as an argument to the function. Even more weird is the semantics of using these functions.

```
int main() {
  Rat x=Rat(2,3);
  Rat y=Rat(4,5);
  cout<<x.numer(y)<<''/''<<x.denom(y)<<''\n'';
}
```

In the example above, we access the function through object x using the dot notation, but we use object y as an argument. Therefore, we will output the numerator and denominator of y, although we are accessing the function through x. To avoid such wired syntax and semantics, the argument to the function should be *implicit*. In other words, the functions **numer()** and **denom()** should not have any parameters. The parameter should be *understood* from the object used in the dot notation.

```
class Rat {

 private:
  int n;
  int d;

 public:
  Rat(int a, int b) {
    n=a/gcd(a,b);
    d=b/gcd(a,b);
  }

  int numer() { //parameter is implicit
    return n;
  }

  int denom() { //parameter is implicit
    return d;
  }
};

int main() {
  Rat x=Rat(2,3);
  Rat y=Rat(4,5);
  cout<<x.numer()<<''/''<<x.denom()<<''\n'';
  cout<<y.numer()<<''/''<<y.denom()<<''\n'';
}
```

Without an *implicit* parameter, there is no way for **numer()** and **denom()** to distinguish one call from another, unless every object has its own set of member functions (like it has its own set of member data). However, that would be inefficient and, therefore, it is not how it is done. Therefore, given that all objects share the same set of member functions, **x.numer()** and **y.numer()** retrieve the same function stored in memory. But this function behaves differently in the two calls. This is only possible if there is some *information* being communicated to the function every time. Such information of course passed as a parameter, the *implicit* one! The next section exposes this mechanism.

8.3 Exposing the class, self reference

Every member function (including constructors) in a class has a *hidden* or *implicit* parameter added by C++ at the end of the list of parameters explicitly defined by the programmer. That parameter is known as the *self reference* parameter and it has the name **this**. Since **this** is a parameter, we must ask the what type does **this** have? The answer is that **this** is a **constant** pointer of the type defined by the class. Therefore, in our example, **this** is declared at the end of the parameter list as follows:

```
( ... , Rat * const this)
```

A small section on the basic use of const is included at the end of this chapter. For now, a constant pointer means that the pointer cannot be changed, i.e. cannot be assigned another address.

C++ uses the self reference parameter **this** to identify the object on which the dot notation was used to invoke the function (or the objet being constructed in case of a constructor). Therefore, another way of visualizing the class is as follows (what is really happening):

```
//the nice look                                //what is really happening

class Rat {

 private:
  int n;
  int d;

 public:
  Rat(int a, int b) {                           Rat::Rat(int a, int b, Rat * const this) {
    n=a/gcd(a,b);                                 (*this).n=a/gcd(a,b);
    d=b/gcd(a,b);                                 (*this).d=a/gcd(a,b);
  }                                             }

  int numer() { //parameter is implicit         Rat::numer(Rat * const this) {
    return n;                                     return (*this).n;
  }                                             }

  int denom() { //parameter is implicit         Rat::numer(Rat * const this) {
    return d;                                     return (*this).d;
  }                                             }
};

int main() {
  Rat x=Rat(2,3);
  Rat y=Rat(4,5);
  cout<<x.numer()<<''/''<<x.denom()<<''\n'';     cout<<Rat::numer(&x)<<''/''<<Rat::denom(&x)<<''\n'';
  cout<<y.numer()<<''/''<<y.denom()<<''\n'';     cout<<Rat::numer(&y)<<''/''<<Rat::denom(&y)<<''\n'';
}
```

Although the **this** pointer is *hidden*, we can actually use it in the body of the function as the rest of the parameters. For instance, this provides a way to overcome the shadowing problem without having to rename parameters (or member data):

```
class Rat {

 private:
  int n;
  int d;
 public:
  Rat(int n, int d) {
    (*this).n=n/gcd(n,d);
    (*this).d=d/gcd(n,d);
  }

  int numer() {
    return n;
  }

  int denom() {
    return d;
  }
};
```

In this new version of the constructor, first the object is obtained by dereferencing the **this** pointer. Then the member data is accessed using the dot notation. Therefore, while **n** in the body of the constructor refers to the parameter, **(*this).n** refers to the member datum **n** declared in the class.

C++ provides the operator $->$ to simplify the combined use of dereferencing and the dot notation.

$$\mathbf{A}- >\mathbf{B} \equiv (\mathbf{*A}).\mathbf{B}$$

```
class Rat {

 private:
  int n;
  int d;

 public:
  Rat(int n, int d) {
    this->n=n/gcd(n,d);
    this->d=d/gcd(n,d);
  }

  int numer() {
    return n;
  }

  int denom() {
    return d;
  }
};
```

8.4 Adding more constructors, default constructor

Now we have enough infrastructure to handle rational numbers. But it would be even nicer if we can represent integers as rational numbers too. This can be easily achieved by constructing a rational number with a denominator of 1.

```
int main() {
  Rat x=Rat(2,1); //construct the rational number 2
}
```

Alternatively, a programmer is always looking for simplicity, and one may like to do the following:

```
int main() {
  Rat x=2; //WOW!
}
```

Luckily, C++ interprets this as:

```
int main() {
  Rat x=Rat(2);
}
```

Therefore, C++ assumes the existence of a constructor that takes only one parameter (an integer). Of course we don't have one for our Rat class. Therefore, we face a compile time error. One way to make the compiler happy is by giving a default value for one of the parameters. In this case, when only one argument is passed, the default value is assumed for the second one (this technique applies in general to any function and not only in the context of a class).

```
class Rat {

  . . .

 public:
  Rat(int n, int d=1) {
    this->n=n/gcd(n,d);
    this->d=n/gcd(n,d);
  }

  . . .
};

int main() {
  Rat x=Rat(2,1);

  Rat x=Rat(2); //equivalent to Rat x=Rat(2,1);

  Rat z=2; //equivalent to Rat z=Rat(2)
}
```

Another way is to actually provide another constructor for the class that takes only one integer as a parameter. This is acceptable by the function overloading rules: two functions with the same name and in the same scope must have different parameters (number of parameters and/or types of parameters).

```
class Rat {

  . . .

 public:
  Rat(int n, int d) {
    this->n=n/gcd(n,d);
    this->d=n/gcd(n,d);
  }

  Rat(int n) {
    this->n=n;
    d=1;
  }

  . . .
};

int main() {
  Rat x=Rat(2,1); //calling the first constructor

  Rat y=Rat(2); //calling the second constructor

  Rat z=2; //equivalent to Rat z=Rat(2);
}
```

What if we combine both techniques. Well, in that case we create a combination that is not very useful. The following program fails to compile.

```
class Rat {

  . . .

 public:
  Rat(int n, int d=1) {
    this->n=n/gcd(n,d);
    this->d=n/gcd(n,d);
  }

  Rat(int n) {
    this->n=n;
    d=1;
  }

  . . .
};

int main() {
  Rat x=2; //is this calling Rat(2,1) i.e. first constructor
           //or calling Rat(2) i.e. second constructor

  Rat y=Rat(2); //same problem
}
```

While both constructors are distinguishable according to the function overloading rules, the compiler cannot resolve which constructor must be called in the example above. Therefore, the only way to construct a rational number in this case is by calling the first constructor with two arguments explicitly all the time!

Yet, another programmer may want to do the following:

```
int main() {
  Rat x;
}
```

And as you might correctly suspect, C++ interprets this as follows:

```
int main() {
  Rat x=Rat();
}
```

Therefore, C++ would be looking for a constructor with no parameters. This is a special constructor and is known as the *default constructor*. We don't have a default constructor for our Rat class; therefore, the above program does not compile. We can simply add one, but the question is what do we want a default constructor to do? One possibility is to construct the rational number zero by default.

```
class Rat {

  . . .

 public:
  Rat(int n, int d) {
    this->n=n/gcd(n,d);
    this->d=n/gcd(n,d);
  }

  Rat(int n) {
    this->n=n;
    d=1;
  }

  Rat() {
    n=0;
    d=1;
  }

  . . .
};
```

It is always a good idea to have a default constructor. In some cases, it is necessary to have a default constructor; for instance, when declaring an array of rational numbers, the compiler looks for a default constructor to construct every element:

```
Rat myarray[10]; //calls default constructor
```

Therefore, it is possible to obtain a compile time error just because the default constructor is missing. We will look at other cases later.

If a class has no constructors at all, then C++ adds a default constructor to it that does nothing. Here's an example:

```
class Rat {

 private:
  int n;
  int d;

 public:
  int numer() {
    return n;
  }

  int denom() {
    return d;
  }
};
```

This is equivalent to the following:

```
class Rat {

 private:
  int n;
  int d;

 public:
  Rat() { }

  int numer() {
    return n;
  }

  int denom() {
    return d;
  }
};
```

8.5 Avoiding excessive construction

So far our Rat class allows us to construct rational numbers (in different ways) and to extract the numerator and denominator of a rational number. Because we have chosen to make the member data private, the class does not allow us to modify the *value* of a rational number [2]. In particular, the following code fails to compile:

[2]Note that assigning a value for x.numer(), as in **x.numer()=4;** for instance, is not allowed because x.numer() simply returns the value of a variable and not the variable itself to be assigned. We say that x.numer() is not an l-value (something that can appear to the left of an assignment).

```
int main() {
  Rat x=Rat(2,3);

  . . .

  //attempt to modify x
  x.n=3;
  x.d=4;

  . . .
}
```

The only way we can modify a rational number is to assign it to another one as in the following example:

```
int main() {
  Rat x=Rat(2,3);

  . . .

  x=Rat(4,5);

  . . .
}
```

However, this means that almost every time we have to modify a rational number, we have to construct another one. The above code is equivalent to the following.

```
int main() {
  Rat x=Rat(2,3);

  . . .

  Rat temp=Rat(4,5);
  x=temp;

  . . .
}
```

While this may be acceptable in some cases, it is not acceptable if the object to be constructed is very large (wasting memory). Moreover, with such a strategy, we have to carefully examine the meaning of an assignment. What does assignment of objects mean? For instance, consider the following program:

```
int main() {
  Rat x=Rat(2,3);
  Rat y;
  y=x;
}
```

Clearly, if x and y were simply integers for instance (or any other basic type), then the semantics of the assignment are intuitive: copy the value of x into y.

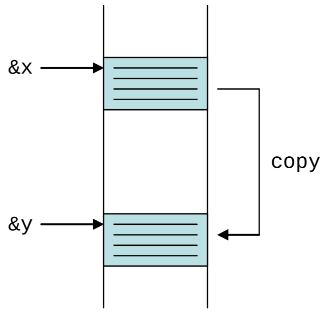


Figure 8.2: Assignment is a copy operation

This means copy the integer in location &x (the address of x) into location &y (the address of y).

When x and y are objects of some defined class, then assignment is still a copy operation, except that the copying is done on a **member by member** basis. Therefore, in the above program, x.n is copied into y.n and x.d is copied into y.d. This is generally the default operation of assignment.

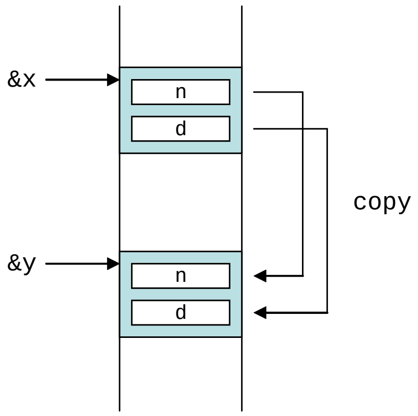


Figure 8.3: Assignment is a member-by-member copy operation

Therefore, if the object is large and only a small part of it needs to be modified, assignment is not the right way. Before we suggest a way to modify our rational number note that the following statement is NOT an assignment (it is an initialization, see Section 2.2).

```
int main() {
  Rat x=Rat(2,3);
  Rat y=x; //this is not an assignment
}
```

The statement above represents an initialization, a way to *construct* y from x and, therefore, is not an assignment (this is different from the previous example where y was already constructed using the default constructor). A special constructor known as the **copy constructor** is involved in this initialization. Such constructor is part of every class by default and it performs exactly the operation of an assignment. We will see later that the default behavior (member-by-member copy) of both the assignment and the copy constructor can be changed.

The copy constructor is also involved in constructing the parameters of a function using the arguments, and in copying the result returned by the function. Here's an example:

```
Rat dummy(Rat x) {
  return x;
}

int main() {
  Rat x=Rat(2,3);
  dummy(x);
}
```

When the function *dummy* is called, the local parameter x is constructed from its argument using the copy constructor. Moreover, when the function returns, and before the local parameter x goes out of scope, the return value is copied using the copy constructor.

To allow the modification of a rational number without using assignment we must provide a public function that performs the modification. Such function can take for instance two parameters and set the numerator and denominator accordingly (like the constructor). This is shown below:

```
class Rat {
 private:
  int n;
  int d;

 public:
  Rat(int n, int d) {
    set(n,d);
  }

  Rat(int n) {
    set(n,1);
  }

  Rat() {
    set(0,1);
  }

  int numer() {
    return n;
  }

  int denom() {
    return d;
  }

  void set(int n, int d) {
    this->n=n/gcd(n,d);
    this->d=d/gcd(n,d);
  }
};
```

```
int main() {
  Rat x=Rat(2,3);

  . . .

  x.set(4,5);

  . . .
}
```

Note that the constructors have been rewritten in terms of the new function. This is possible since a member function of the class can be called from within another member function.

8.6 Basic use of const

We have previously seen that the keyword **const** is used to declare a constant variable. A simple use of const can be as follows:

```
int main() {
  const int i=2;

  . . .
}
```

However, the use of const is more versatile and can be tricky. Here are some basic variations.

8.6.1 Constant object

To declare a constant object we use the following syntax:

```
int main() {
  const T x=...;

  . . .
}
```

This means that x cannot be assigned after it is declared. Any attempt to assign x will cause a compile time error.

8.6.2 Constant pointer

To declare a constant pointer we use the following syntax:

```
int main() {
  T * const p=...;

  . . .
}
```

This means that p cannot be assigned after it is declared. Any attempt to assign p, i.e. change the address to which is is pointing, will cause a compile time error.

8.6.3 Pointer to constant object

To declare a pointer to a constant object we use the following syntax:

```
int main() {
  const T * p; //note: no need to initialize p, it’s not constant

  . . .
}
```

This means that the object cannot change through pointer p. Any attempt to change the object by dereferencing p will cause a compile time error.

```
int main() {
  int x;
  const int * p;
  p=&x; //ok, p is assigned the address of x
  x=2; //ok, x is not constant
  *p=3; //error, attempt to change x through p
}
```

8.6.4 Constant pointer to constant object

To declare a constant pointer to a constant object we use the following syntax:

```
int main() {
  const T * const p=...;

  . . .
}
```

This means that the pointer is constant and it is a pointer to a constant object (the previous two cases combined). Therefore, p must be initialized.

```
int main() {
  int x;
  int y;
  const int z=1;
  const int * const p=&x; //ok, p must be initialized because it is constant
  x=2; //ok, x is not constant
  *p=3; //error, attempt to change x through p
  p=&y; //error, attempt to assign p another address
  z=4; //error, z is constant;
}
```

Chapter 9

Operator overloading

9.1 Introduction

Consider the problem of adding rational numbers and outputting the result. So far, we can do this task using our Rat class our addRat function as follows:

```
class Rat {

  . . .
};

Rat addRat(Rat x, Rat y) {
  return Rat(x.numer()*y.denom()+y.numer()*x.denom(),
             x.denom()*y.denom());
}

int main() {
  Rat x=Rat(2,3);
  Rat y=Rat(4,5);
  Rat z;
  z=addRat(x,y);
  cout<<z.numer()<<''/''<<z.denom();
}
```

It would be more convenient if we could carry our abstraction one step further and add rational numbers in the same way we add integers. Moreover, to output a rational number we need to explicitly output its numerator, a "/" symbol, and its denominator. It would be more convenient if we do not have to worry about this formatting.

```
int main() {
  Rat x=Rat(2,3);
  Rat y=Rat(4,5);
  Rat z;
  z=x+y;
  cout<<z;
}
```

While the code above looks more natural, unfortunately operator + does not support our newly defined types. Similarly, operator << does not know how to send a rational number to an output stream. Luckily, C++ provides a way to *redefine* these operators to deal with new types. This is called **operator overloading**.

9.2 Operator overloading

Most C++ operators can be overloaded. There are about 70 operators 5 of which are not overloadable. These are **::**, **.**, **sizeof**, **.***, and **?:** (we did not see all of them). To overload an operator, we simply define it as a function with the appropriate number of parameters. For instance, the + operator takes either one or two parameters. The function name must start with the word **operator** followed by the operator itself:

```
Rat operator+(Rat x, Rat y) {
  return Rat(x.numer()*y.denom()+y.numer()*x.denom(),
             x.denom()*y.denom());
}

//unary +
Rat operator+(Rat x) {
  return x;
}
```

Similarly, we can overload the other operators:

```
Rat operator-(Rat x, Rat y) {
  return Rat(x.numer()*y.denom()-y.numer()*x.denom(),
             x.denom()*y.denom());
}

//unary -
Rat operator-(Rat x) {
  return Rat(-x.numer(),x.denom());
}

//can be unary but then it is meant to be
//the dereferencing operator by convention
Rat operator*(Rat x, Rat y) {
  return Rat(x.numer()*y.numer(),x.denom()*y.denom());
}

//must take exactly two parameters
Rat operator/(Rat x, Rat y) {
  return Rat(x.numer()*y.denom(),y.numer()*x.denom());
}
```

```
int main() {
  Rat x=Rat(2,3);
  Rat y=Rat(4,5);
  z=x+y; //equivalent to operator+(x,y);
  z=x-y; //equivalent to operator-(x,y);
  z=-y+x; //equivalent to operator+(operator-(y),x);
  z=x+x*y; //equivalent to operator+(x,operator*(x,y));
  z=x/y*x; //equivalent to operator*(operator/(x,y),x);

}
```

Note that the precedence of the operators is preserved when operators are overloaded and cannot be changed. So far, what we have done offers us more than what we would expect. For instance, all the following will work:

```
int main() {
  Rat x=Rat(2,3);
  Rat z;
  z=x+x;
  z=x+2;
  z=2+x;
  z=2+2; //this does not call the overloaded operator
         //the result is converted from int to Rat
         //before assignment (see below)
}
```

This is because because in operator+(Rat x, Rat y), both parameters x and y can be constructed from ints using the second constructor in the Rat class. So let us observe what is really happening with a code such as the following:

```
Rat operator+(Rat x, Rat y) {return Rat(...);}

int main() {
  Rat x=Rat(2,3);
  Rat z;
  z=x+2;
}
```

1. x is constructed using the first constructor in class Rat [1].
2. z is constructed using the default constructor in class Rat
3. the local parameter x of operator+ is constructed using the default copy constructor in class Rat (memberwise copy from x)
4. the local parameter y of operator+ is constructed using the second constructor in class Rat with 2 being the argument
5. a local Rat is constructed within the operator+ function to form the result.

[1]Actually a temporary is constructed then copied into x using the copy constructor of the class. The C++ standard allows an implementation to omit creating a temporary which is only used to initialize another object of the same type. This optimization in enabled by default in g++ for instance and, therefore, this step is not done. Specifying the option -fno-elide-constructors disables that optimization, and forces g++ to call the copy constructor in all cases. Another way to avoid the extra construction in Rat x=Rat(2,3) is to replaced it with Rat x(2,3).

6. the copy constructor is used upon return to construct a temporary Rat from the result, see footnote 1. [2].
7. the local parameter x and the local parameter y are destroyed
8. the temporary Rat in step 6 is used by the assignment operator to assign z
9. the temporary Rat is destroyed
10. the local Rat is destroyed (placed in this order in case step 6 is not done)
11. x and z are destroyed

If we did not have the second constructor (or if the constructor was declared *explicit* as we will see later on), then step 4 of the above process will fail. Instead, C++ will be looking for a + operator defined as operator+(Rat, int). Therefore, it is a simply type mismatch. To solve the problem, we can overload the + operator many times, once for every possible combination of parameter types:

```
Rat operator+(Rat x, Rat y) {
  return Rat(x.numer()*y.denom()+y.numer()*x.denom(),
             x.denom()*y.denom());
}

Rat operator+(Rat x, int y) {
  return Rat(x.numer()+y*x.denom(),x.denom());
}

Rat operator+(int x, Rat y) {
  return Rat(x*y.denom()+y.numer(),y.denom());
}

//this last one does not compile
Rat operator+(int x, int y) {
  return Rat(x+y,1);
}
```

Unfortunately, the last option does not compile. The reason for this is that C++ prevents from overloading the default behavior of operators for basic types. If the last one were to succeed, then every addition of two integers will produce a Rat instead of an integer (most likely an undesired behavior).

9.3 Rules for operator overloading

These are the rules for operator overloading:

- Every operator must take a certain number of parameters (different from one operator to another)

[2]The C++ standard allows an implementation to omit creating a temporary which is only used to initialize another object of the same type. This optimization in enabled by default in g++ for instance and, therefore, this step is not done. Specifying the option -fno-elide-constructors disables that optimization, and forces g++ to call the copy constructor in all cases.

- At least one parameter must be of non-pointer type and of non-basic type (e.g. user defined class), unless the overloading is done inside the class (see below)
- When overloaded within a class (we will see this shortly), one parameter is implicit, i.e. the **this** pointer; therefore, the only restriction in this case is the number of parameters (the number includes the implicit one)

9.4 The increment/decrement operators and introduction to references

Consider the problem of overloading the increment operator ++. For instance, we might want to write the following program:

```
int main() {
  Rat x=Rat(2,3);
  Rat y;
  y=++x;
}
```

In the above program, y should be the rational number 5/3 (borrowing the semantics from the ++ operator for integers and floating numbers). The pre increment operator ++ increments a number by 1 and returns the result of the increment. To achieve this behavior with rational numbers, we need to overload the pre increment operator ++ to accept a parameter of class Rat. The ++ operator is a unary operator; therefore, we have to provide the appropriate overloading with just one parameter:

```
Rat operator++(Rat x) {
  return Rat(x.numer()+x.denom(),x.denom());
}
```

While the above returns the correct thing, which is a rational number equal to $1 + x$, it does not increment the argument itself. Therefore, the statement **y=++x** assigns the correct value for y, but leaves x unchanged. At this point, the dilemma is clear: How can we change x without passing it by reference using a pointer?

```
Rat operator++(Rat * x) {
  x->set(x->numer()+x->denom(),x->denom());
  return x;
}
```

Obviously, such an attempts breaks the rules of operator overloading (the compiler will complain about passing a pointer, see Section 3).

9.4.1 Why does C++ forbid pointers for operators?

The reason for introducing operator overloading into the language is to treat user defined types in a similar way to basic types. Therefore, the key issue here is form. One would like for instance to create a type for rational numbers where arithmetic operators (and other operators) preserve their way of use. Allowing pointers as arguments for operator overloading adds more flexibility but defeats that purpose. For example, such overloading for the pre increment operator would have to be invoked using an expression like **++&x**, which does not look right.

For this reason, C++ provides a way to pass arguments by reference **without pointers**. Such a scheme is simply called **passing by reference**. It's a way to provide pointer semantics with non-pointer syntax or form. A parameter declared this way is called a **reference**.

9.4.2 References

At some level, one might think of a reference as an alternative name of something. Here's a simple example that illustrates the idea:

```
int main() {
  int i=1;

  //pointer
  int * p=&i; //p is a pointer to i
  *p=2; //change i to 2 by dereferencing p

  //reference
  int& r=i; //r is a reference to i
  r=2; //change i to 2 through its reference
       //pointer semantics
       //non-pointer form
}
```

Therefore, while * denotes a pointer, & denotes a reference. A reference can be used in the same way as a pointer; howerver, it needs not be de-referenced (using the de-referencing operator *). A reference represents another name for the thing it is a reference to. Therefore, changing a reference **means** changing that thing. As a result, a reference provides the semantics of a pointer while keeping the syntax of a non-pointer.

Nevertheless, a reference is not a pure magic. It is actually implemented using pointers. Here's the equivalence:

```
//provided syntax                  //internal implementation

int main() {
  int& r=i;                        int * const r=&i; //constant
                                                     //pointer
  r=2; //changing pointer
       //is not the intention      *r=2;
}
```

Therefore, a reference is a constant pointer that is automatically dereferenced every time it is used. Since a reference is a constant pointer in reality, it must be initialized.

References as types of function parameters

A reference can be the type of a function parameter. Here's an example:

```
void f(int x, int& y) {
  x=2;
  y=2;
}

int main() {
  int i=1;
  int j=1;
  f(i,j);

  //i is still 1
  //j is now 2

  . . .
}
```

It should not be surprising that the value for j is changing in the above program. After all, that's what a reference must do: it provides pointer semantics with non-pointer syntax. The above program can be better understood if we replace the reference with its internal implementation (when an argument is passed by reference, a pointer is passed instead).

```
void f(int x, int * const y) {
  x=2;
  *y=2;
}

int main() {
  int i=1;
  int j=1;
  f(i,&j);

  //i is still 1
  //j is now 2
}
```

Here's another example of a concept that we have seen before:

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

int main() {
  int x=1;
  int y=2;
  swap(x,y);
  //now x is 2 and y is 1
}
```

And it's equivalence:

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

int main() {
  int x=1;
  int y=2;
  swap(&x,&y);
  //now x is 2 and y is 1
}
```

References as function return types

A reference can also be the return type of a function. Here's an example:

```
int& dummy(int& x) {
  return x;
}

int main() {
  int x=1;
  dummy(x)=2; //in this case,
              //equivalent to x=2
}
```

Normally, what a function returns cannot be assigned, because it is simply a value (not an l-value). With a reference as the return type, the function returns the actual thing. In the above example, x is being returned, which is the parameter of the function, which is also a reference to the argument. Therefore, what the function returns is the variable x itself which is declared in the main() function. With a reference as the return type, one must always be careful not to return a reference to something that is local to the function, and will go out of scope as soon as the function returns. Here's an example:

```
int& dummy(int x) {
  return x;
}
```

Here the local variable x is being returned, which ceases to exist after the function returns. We end up with a reference to something that does not exist!

9.4.3 Overloading operators ++ and −− (almost) correctly

[3] Now that we know how to achieve pointer semantics without pointers, let's overload the pre increment operator:

```
Rat operator++(Rat& x) {
  x.set(x.numer()+x.denom(),x.denom());
  return x;
}
```

What if we would like to overload the post increment operator ++? The post increment operator is similar to the pre increment operator except that it returns the value before the actual increment is performed. For instance, in the following program, y is assigned the value 1 and then x becomes 2.

```
int main() {
  int x=1;
  int y=x++;
}
```

Therefore, we have two issues to consider. First, since both operators (pre and post increment) are unary, we must find a way to distinguish them from each other. Having resolved this issue, we must find a way to return the value of the argument before increment.

To distinguish pre increment from post increment, C++ assumes that the post increment operator takes an additional parameter of type int, which is ignored. Therefore, we are looking at overloading the following operator:

```
Rat operator++(Rat& x, int) {

  //int being ignored

  . . .
}
```

To return the value of the argument before the increment, we can simply use a temporary variable as follows:

```
Rat operator++(Rat& x, int) {
  Rat temp=x; //copy constructor
  ++x; //call the pre increment operator
  return temp; //copy constructor will copy the result
               //before temp goes out of scope
}
```

Here's a similar overloading for operator −−:

```
//pre decrement
Rat operator--(Rat& x) {
  x.set(x.numer()-x.denom().x.denom());
  return x;
}
```

[3]See further notes to understand why **almost**.

```
//post decrement
Rat operator--(Rat& x, int) {
  Rat temp=x;
  --x;
  return temp;
}
```

9.4.4 Overloading operator <<

[4] The << operator takes two parameters. If we want the following program to work:

```
int main() {
  Rat x=Rat(2,3);
  cout<<x;
}
```

then we must overload operator << with the first parameter being an output stream (the type of cout) and the second parameter being a Rat.

```
#include <iostream>

using std::ostream;

___?___ operator<<(ostream s, Rat x) {
  s<<x.numer()<<''/''<<x.denom()<<''\n'';
  return ___?___;
}
```

The important question here is what should operator<< return? The answer really depends on how the operator << is intended to be used. Generally, with such an operator, chaining is desired. For instance, the following statement should work.

```
cout<<x<<y<<z;
```

Therefore, since **cout<<x<<y<<z;** is equivalent to **((cout<<x)<<y)<<x;**, we conclude that calling the operator << must return an output stream, so that the remaining part of the chain works. Here's a first attempt:

```
ostream operator<<(ostream s, Rat x) {
  s<<x.numer()<<''/''<<x.denom()<<''\n'';
  return s;
}
```

Unfortunately, this specific way of overloading the << operator will cause a compiler error due to a particular implementation of the ostream class. To understand this point, let's examine what happens with the following statement:

```
cout<<x;
```

[4] See further notes to learn a better way for overloading this operator.

The function operator<< is called with two arguments: **cout** (an ostream object) and x (a Rat object). Focusing on the first argument, the local parameter s in operator<< is constructed using the copy constructor of class ostream. Similarly, when s is returned, this copy constructor is involved again. However, the copy constructor of class ostream is private (not really, it's that of its base class but we are not concerned about this at the moment). The key point is that C++ forbids the call to the copy constructor of class ostream. The reason for this is efficiency to avoid excessive copying of a large object such as an output stream.

Consequently, we must refrain from copying such an object. The only way is, therefore, to pass it (and return it) by reference. Here's the final version.

```
ostream& operator<<(ostream& s, Rat x) {
  s<<x.numer()<<''/''<<x.denom()<<''\n'';
  return s;
}
```

The following would have worked also (why?) but is not desired (why?):

```
ostream * operator<<(ostream * s, Rat x) {
  (*s)<<x.numer()<<''/''<<x.denom()<<''\n'';
  return s;
}
```

We will discuss passing by reference for efficiency in the next chapter.

9.5 Making friends and moving operators inside the class

Consider just for the sake of illustration that our Rat class is defined without the public member functions **numer()** and **denom()**. Moreover, assume that we would still like to provide the + operator. This is problematic because operator+ (being defined outside the class) cannot access the numerator and denominator of a rational number since they are decalred private. Nevertheless, the only way we can now define operator+ is as follows (which will not compile):

```
Rat operator+(Rat x, Rat y) {
  return Rat(x.n*y.d+x.d*y.n,x.n*y.d);
}
```

Since the designer of class Rat is likely also the person providing this operator overloading, C++ provides an escape strategy for this situation. We can declare operator+ as a **friend** of class Rat. In general, a friend function of a class can access the private members (date or functions) of the class. Here's how this *friendship* can be achieved:

```
class Rat {

  friend Rat operator+(Rat x, Rat y);

  . . .
};
```

Here's another example to re-instate **numer()** and **denom()**:

```
class Rat {

  friend int numer(Rat x);
  friend int denom(Rat x);

  . . .
};

int numer(Rat x) {
  return x.n; //ok, it's a friend
}

int denom(Rat x) {
  return x.d; //ok, it's a friend
}

Rat operator+(Rat x, Rat y) {
  return Rat(numer(x)*denom(y)+denom(x)*numer(y),denom(x)*denom(x));
}
```

Note that only the designer/implementer of a class can decide which functions must be friends. Therefore, the technique of friend functions cannot jeopardize the class.

An alternative to using friend functions is to move operator+ inside the class, i.e. to make it a member function (recall that member functions have access to private member data). But every member function has an implicit parameter (the **this** constant pointer). Therefore, the implicit pointer provides the **first** operand for the operator [5]. Here's an example:

```
class Rat {
  int n;
  int d;
 public:

  . . .

  Rat operator+(Rat y) {
    return Rat(n*y.d+d*y.n,d*y.d);
  }

  . . .
};
```

Note that the above function is internally implemented as follows:

```
Rat::operator+(Rat y, Rat * const this) {
  return Rat(this->n*y.d+this->d*y.d,this->d*y.d);
}
```

[5]When an operator is defined as a member function, the restriction on the types of its parameters is removed, since the constant **this** pointer is always one of the parameters and is passed **implicitly**.

Now the operator can be used as before:

```
int main() {
  Rat x=Rat(2,3);
  Rat y=Rat(4,5);
  Rat z;
  z=x+y; //equivalent to z=x.operator+(y);
         //(internally, z=Rat::operator+(y,&x);)
}
```

With this technique, however, we break some functionality that used to work when the operator was defined outside the class. Consider the following:

```
int main() {
  Rat x=Rat(2,3);
  Rat z;
  z=x+2; //ok, equivalent to z=x+Rat(2);
  z=2+x; //error, 2.operator+(x) meaningless
}
```

Before, what made **x+2** and **2+x** work was the fact that C++ was implicitly converting the **2** into a rational number using the appropriate constructor (the one that takes one integer as parameter). Now, however, a function operator+ that takes two rational numbers as parameters does not exists. We only have a member function operator+ in class Rat. This means that this function must be called through a Rat object. The statement **2+x** does not give C++ enough information on which object to use (compiler is not intelligent enough to switch the order of parameters, and of course it shouldn't in general).

In this situation, to fix the inconsistent behavior of **x+2** and **2+x**, one could only disable **x+2** from working. This is done by declaring the constructor Rat(int) as **explicit**, which tells C++ not to use it implicitly for conversion.

```
class Rat {

  . . .

 public:

  . . .

 explicit Rat(int n) {set(n,1);}

 . . .
};
```

Based on the above scenario, it may be better to overload the + operator outside class Rat. This raises the question: which operators should be overloaded as member functions? Well, here's a rule of thumb:

- unary operators (the implicit **this** parameter becomes the operand)
- when the first operand is always the object itself
- when we have to (some operators must be member functions, e.g. the assignment operator which we will study later, the [] operator, ...)

```
class Rat {
  int n;
  int d;

 public:

  . . .

  Rat operator++() {
    n=n+d;
    return *this;
  }

  Rat operator++(int) {
    Rat temp=*this;
    n=n+d;
    return temp;
  }

  int operator[](int i) {
    if (i==0)
      return n;
    else
      return d;
};

Rat operator+(Rat x, Rat y) {
  return Rat(x[0]*y[1]+x[1]*y[0],y[1]*y[1]);
}

ostream& operator<<(ostream&s, Rat x) {
  s<<x[0]<<''/''<<x[1]<<''\n'';
  return s;

}
```

Chapter 10

Constness

10.1 Introduction

So far, we have seen the use of references in two contexts:

- to obtain pointer semantics while maintaining non-pointer syntax (e.g. parameters for operator overloading)
- to avoid excessive copying (e.g. avoid calling the copy constructor for an ostream object)

While the first use of references deals with form, the second deals with efficiency. The problem is that both uses are indistinguishable and, therefore, fail to convey the intention of the programmer. For instance, consider the following function (the body of the function is intentionally unrevealed):

```
void secret(Rat& x) {

  . . .
};
```

The user of such a function may write the following:

```
int main() {
  Rat x=Rat(2,3);
  secret(x);
}
```

The user knows that passing by reference provides pointer semantics (well, assuming he/she took CSCI135). Therefore, to be on the safe side, the user may assume that the secret function is going to modify the Rat argument (he/she cannot rely on *x* remaining the same after calling the function). However, the programmer might have just used a reference to avoid calling the copy constructor of Rat, and had no intention of changing the argument.

10.2 Using const (again)

To convey that the argument is passed by reference for efficiency with no intention to modify, we can use the **const** keyword (as we have said before, the use of const in C++ is versatile).

```
void secret(const Rat& x) {

  . . .
};
```

In fact, const is not just a way to convey intention, it is used by the compiler as a **guarantee**: any attempt to modify x within the body of the function will generate a compiler error.

Let's see how we can benefit from this technique. For instance, we can rewrite our operator+ function to pass the arguments by reference for efficiency (copy constructor is not involved in constructing the parameters). Moreover, since operator+ is not supposed to modify the operands, we can use const in the declaration of the parameters.

```
//does not compile
Rat operator+(const Rat& x, const Rat& y) {
  return Rat(x.numer()*y.denom()+x.denom()*y.numer(),
             x.denom()*y.denom());
}
```

The compiler must guarantee that the body of the function operator+ does not modify parameters x and y. Obviously not such attempt is identified. At least for us. But how can the compiler really tell that, for instance, calling the member functions **x.numer()** and **x.denom()** does not modify object x? To make this issue clear, consider the following two functions:

```
void f(const Rat& x) {
  x.set(2,3);
}

void g(const Rat& x) {
  int n=x.numer();
}
```

In both functions, x is not supposed to change. However, both functions call a member function of class Rat through x. Clearly, the two behaviors are indistinguishable by the compiler. Yet, the compiler is required to develop some sort of *smartness* to decide that function **f(const Rat&)** should not compile because **x.set(int,int)** modifies x, while function **g(const Rat&)** should compile because **x.numer()** does not modify x. This kind of *smartness* is not possible in general. Therefore, in both cases the compiler will complain.

Coming back to our revisited overloading of operator+ using constant references, we now understand why the compiler will complain. To make the compiler happy, the programmer must provide some kind of assurance that member function **numer()** does not modify its object (same for **denom()**). Such a function is called **constant member function**.

10.3 Constant member functions

To declare a member function as constant (i.e. does not modify its object), we simply add const at the end of the function prototype (no kidding!).

```
class Rat {
  int n;
  int d;

 public:

  . . .

  int numer()const {
    return n;
  }

  int denom()const {
    return d;
  }

  void set(int n, int d) {
    this->n=n/gcd(n,d);
    this->d=d/gcd(n,d);
  }

  . . .
};

Rat operator+(const Rat& x, const Rat& y) {
  return Rat(x.numer()*y.denom()+x.denom()*y.numer(),
             x.denom()*y.denom());
             //ok, numer() and denom() are const
}
```

Declaring a member function as constant forbids any attempt to modify the object inside the function. This is why it can be interpreted by the compiler as an assurance for the *constness* of the object. In fact, this technique (of constant member function) is not much different from that of declaring a function parameter as constant.

```
class Rat{
  int n;
  int d;

 public:

  int numer()const {
    //cannot modify the object here
    //i.e. neither n nor d can change
    return n;
  }

};

void numer(const Rat& x) {
  //cannot modify object x here
}
```

What seems to be a complicated behavior is simply achieved by a simple typing convention: when a member function is declared const, the type of its implicit parameter (the **this** pointer) is changed from a "constant pointer to an object" to a "constant pointer to a constant object".

```
//what you see                      //what you get

class Rat {
  int n;
  int d;

 public:

  . . .

  int numer()const {                Rat::numer(const Rat * const this) {
    return n;                         return this->n;
  }                                 }

  int denom()const {                Rat::denom(const Rat * const this) {
    return d;                         return this->d;
  }                                 }

  void set(int n, int d) {          void set(int n, int d, Rat * const this) {
    this->n=n/gcd(n,d);               this->n=n/gcd(n,d);
    this->d=d/gcd(n,d);               this->d=d/gcd(n,d);
  }                                 }

  . . .
};

//recall that a reference          //is a constant pointer
//therefore, Rat& x                //is simply Rat * const x

void f(const Rat& x) {             void f(const Rat * const x) {
  x.set(2,3);                        Rat::set(2,3,x); //error, type mismatch for x
}                                  }

void g(const Rat& x) {             void g(const Rat * const x) {
  int n=x.numer();                   int n=Rat::numer(x); //ok
}                                  }
```

Here's another example of how this typing convention works:

```
int main() {

  const Rat x=Rat(2,3);
  //x is a constant rational number

  x.numer();
  //equivalent to Rat::numer(&x)
  //&x is pointer to const, and numer(const Rat * const)
  //accepts pointer to const
  //ok

  x.set(4,5);
  //equivalent to Rat::set(4,5,&x)
  //&x is pointer to const, but set(int, int, Rat * const)
  //accepts pointer to non-const
  //error
}
```

If member function **numer()** were not declared as const, the second statement of the above program would have failed to compile. Therefore, it is important to declared every member function that does not modify the object as const.

10.4 Moral of the story

We can summarize this whole discussion as follows:

- we may pass an argument by reference for efficiency (to avoid the copy constructor), in which case it is better to declare its corresponding parameter as a constant reference
- if a parameter is declared as a constant reference, it cannot be changed in the body of the function (this is a direct consequence of the fact that a reference is internally a pointer and is automatically dereferenced, so a constant reference is a pointer to const)
- a pointer to const argument cannot be passed for a pointer to non-const parameter (the reverse is ok)
- every member function that does not modify the object must be declared as const; this declares its implicit parameter as a constant reference

10.5 Redoing operator overloading

```
class Rat {
  int n;
  int d;

 public:

  . . .

  int numer()const {
    return n;
  }

  int denom()const {
    return d;
  }

  int set(int n, int d) {
    this->n=n/gcd(n,d);
    this->n=d/gcd(n,d);
  }

  Rat operator++() {
    n=n+d;
    return *this;
  }
  Rat operator++(int) {
    Rat temp=*this;
    n=n+d;
    return temp;
  }

  int operator[](int i)const {
    if (i==0)
      return n;
    else
      return d;
};

Rat operator+(const Rat& x, const Rat& y) {
  return Rat(x[0]*y[1]+x[1]*y[0],y[1]*y[1]);
}

ostream& operator<<(ostream& s, const Rat& x) {
  s<<x[0]<<''/''<<x[1]<<''\n'';
  return s;

}
```

10.6 Yet a better way

Since we would like our Rat objects to behave in a similar way to numbers, we might want to restrict some functionality. For instance, operator+ returns a Rat object. Objects are usually considered to be l-values; therefore, they can be assigned. As a result, the following code is legal:

```
int main() {
  Rat x=Rat(2,3);
  Rat y=Rat(4,5);
  Rat z=Rat(7,6);
  x+y=z;
}
```

Probably no one would want to make an assignment to the sum of two numbers, but it would be illegal if x, y, and z were of a built-in type. To be consistent with built-in types, we can declare the return type of operator+ as const and, therefore, forbid such an assignment to be made. We can do the same for operator++ and operator−−.

```
class Rat {
  int n;
  int d;

 public:

  . . .
  int numer()const {
    return n;
  }

  int denom()const {
    return d;
  }

  int set(int n, int d) {
    this->n=n/gcd(n,d);
    this->n=d/gcd(n,d);
  }

  const Rat operator++() {
    n=n+d;
    return *this;
  }

  const Rat operator++(int) {
    Rat temp=*this;
    n=n+d;
    return temp;
  }

  int operator[](int i)const {
    if (i==0)
      return n;
    else
      return d;
};

const Rat operator+(const Rat& x, const Rat& y) {
  return Rat(x[0]*y[1]+x[1]*y[0],y[1]*y[1]);
}

ostream& operator<<(ostream& s, const Rat& x) {
  s<<x[0]<<''/''<<x[1]<<''\n'';
  return s;

}
```

Chapter 11

C strings: some string theory

11.1 Introduction

To declare a character, we simply need to specify the type **char**, for example:

```
int main() {
  char c='a'; //use single quotation for character

  . . .
}
```

But what does it take to declare a string. So far we have been using strings as expressions between double quotation marks like for instance "hello". As we will see shortly, these expressions represent **constant** strings, i.e. values to be assigned to strings. But how do we declare a string as a variable that can be assigned? The first thing we have to figure out is the type of such a variable. In C, a string is simply an **array** of characters. This implies that a string has the type **char ***, a pointer to the first character (element) of the string (array).

```
int main() {
  char * s=''hello'';
  cout<<s; //outputs the characters of ''hello''
}
```

There are two important issues with the above program:

- In general, there is no way to determine the size of an array simply from a pointer to its first element. Therefore, how does cout know when to stop outputting characters?
- According to the C++ standard, "hello" is a constant string, i.e. the type of "hello" is **const char ***. Therefore, how come we can initialize a **char *** variable with a string like "hello" (violation of constness rules)?

The answer to the first question is that every string expression "..." is terminated by a special character known as the null character '\0'. Therefore, "hello" is actually "hello\0". The null character is interpreted as the end of the string. That's why we call C strings *null terminated strings.*

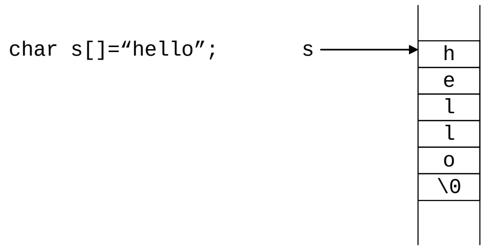


Figure 11.1: The null character

The answer to the second question is that C++ grants a special dispensation for initializations like that because the practice is so common. Nevertheless, they should be avoided. For instance, the following program generates a run-time error:

```
int main() {
  char * s=''hello'';
  s[0]='b'; //s is pointer to non-const, ok at compile time
            //but it points to a const, error at run-time
}
```

Therefore, a better way to declare a string is the following:

```
int main() {
  char s[]=''hello'';
  s[0]='b';
}
```

which is equivalent to the following:

```
int main() {
  char s[6]={'h','e','l','l','o','\0'};
  s[0]='b'; //ok at compile time and run-time
}
```

An array of six characters is declared, and each character of the array is assigned the corresponding character of the string “hello”. Note that we could have explicitly specified the number 6 between the brackets, but any smaller number will cause a compiler error.

11.2 Assignment and pointer semantics

The C++ standard forbids assignment of arrays. Since strings are arrays of characters, we cannot simply assign a string to another. Here’s an example:

```
int main() {
  char s[]=''hello'';
  char t[]=s; //error, invalid initializer
  char r[6];
  r=s; //error, C++ forbids assignment of arrays
}
```

Therefore, the only way to assign strings is to treat them as pointers (recall that an array is a pointer to its first element). C++ does not forbid the assignment of pointer of course; however, this means that assignment of strings has pointer semantics: one pointer is assigned the value of another. Here's an example:

```
int main() {
  char s[]=''hello'';
  char * t=s; //ok
}
```

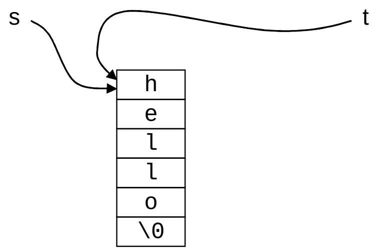


Figure 11.2: Strings (pointers) **s** and **t** share the same physical memory

As illustrated in Figure 2, any change in **s** changes **t** and vice versa. In other terms, **t[i]** is now simply another name for **s[i]**. This is probably not the desired effect wanted from an assignment. What we really want is to copy each character (element) of string (array) **s** into the corresponding character (element) of string (array) **t**. This can be done using a loop as follows:

```
int main() {
  char s[6]=''hello'';
  char t[6];
  for (int i=0;i<6;i++)
    t[i]=s[i]; //assignment of type char
}
```

Another way to accomplish the same result is to use the built in function **strcpy**, which copies one string into another using essentially a similar loop based algorithm.

```
char * strcpy(char * t, const char * s) {
  //copies the string pointed by s into
  //the array pointed by t including
  //the terminating null character

  . . .

  return t;
}

int main() {
  char s[6]=''hello'';
  char t[6];
  strcpy(t,s);
}
```

11.3 Strings as parameters to functions

In the event of copying string **s** into string **t**, what if the length of **s** is unknown? This can happen for instance if **s** is passed as an argument to a function:

```
void f(const char * s) {
  char t[?];
  strcpy(t,s);

  . . .
}
```

One possibility is to make **t** large enough (of course there is always a question of what is enough):

```
void f(const char * s) {
  char t[100];
  strcpy(t,s);

  . . .
}
```

However, the code above is not totally safe. What if the length of **s** is 100 or more? Then we would need at least 101 characters in **t** including the terminating null character. Soon we realize that there is no escape from determining the length l of string **s**. But if **s** is null terminated, we can do that easily and then copy $min(l+1,100)$ characters from **s** into **t**.

```
void f(const char * s) {
  char t[100];
  int length=0;
  while(s[length]!='\0')
    length++;
  for(int i=0;i<min(100,length+1);i++)
    t[i]=s[i];
  t[99]='\0'; //in case the null character is not copied

  . . .
}
```

We can accomplish the same result using the built in functions **strlen** and **strncpy**. **strlen** determines the length of a string and **strncpy** copies a number of characters from one string into another. Both use essentially loop based algorithms similar to above.

```
int strlen(const char * s) {
  //returns number of characters until the
  //terminating null character (not included)

  . . .
}
```

```
char * strncpy ( char * destination, const char * origin, int num) {
  //copies the first num characters of source to destination
  //no null-character is implicitly appended to the end of
  //destination, so destination will only be null terminated
  //if the length of the string in source is less than num

  . . .

  return t;
}

void f(const char * s) {
  char t[100];
  strncpy(t,s,min(100,strlen(s)+1);
  t[99]='\0'; //in case the null character is not copied

  . . .
}
```

Since the length of a string **s** can be determined, one might want to replace the above code with the following:

```
void f(const char * s) {
  char t[strlen(s)+1];
  strcpy(t,s);

  . . .
}
```

While this approach is appealing, we have seen that many compilers will complain because the length of string (array) **t** is not known at compile time. Therefore, we can keep such an approach if we use dynamic memory allocation:

```
void f(const char * s) {
  char * t=new char[strlen(s)+1];
  strcpy(t,s);

  . . .
  delete[] t; //if needed
}
```

11.4 A person class

Consider the following class:

```
class Person {
  char name[100];

 public:
  Person(const char * s) {
    strncpy(name,s,min(100,strlen(s)+1));
    name[99]='\0';
  }
  void print()const {
    cout<<name<<''\n'';
  }
};
```

The class Person has one private member datum, the name, which is a string of 100 characters. For the public interface, it has a constructor which takes a string as parameter (note the const), and a function to print the name on the screen. Therefore, this class can be used as follows:

```
int main() {
  Person saad=Person(''saad'');
  saad.print();
}
```

A better way to define the class Person is by overloading operator << to output the name instead of the function print.

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char name[100];

 public:
  Person(const char * s) {
    strncpy(name,s,min(100,strlen(s)+1));
    name[99]='\0';
  }
};

ostream& operator<<(ostream& s, const Person& p) {
  s<<p.name; //friend can access private data
  return s;
}
```

With this operator overloading added, the class Person can now be used as follows:

```
int main() {
  Person saad=Person(''saad'');
  cout<<saad;
}
```

Regardless of the choice used for printing a Person object, let us conduct a little "experiment":

```
int main() {
  Person saad=Person(''saad'');
  Person clone1=saad;
  Person clone2;
  clone2=saad;
}
```

Here's a line by line description of the above “experiment”:

- **Person saad=Person(”saad”);**: declares a Person object called saad using the class constructor with the string ”saad” as argument.
- **Person clone1=saad;**: declares a Person object called clone1 using the **default copy constructor** of the class
- **Person clone2;**: declares a Person object called clone2 using the class **default constructor**, and gives a compile time error because class Person has no default constructor (we add one below)
- **clone2=saad;**: assigns saad to clone2 using the **default assignment operator**

Here's the added default constructor to class Person that will remove the compile time error generated by the third line described above:

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char name[100];

 public:
  Person(const char * s) {
    strncpy(name,s,min(100,strlen(s)+1));
    name[99]='\0';
  }

  Person() {
    strcpy(name,''john'');
  }
};

ostream& operator<<(ostream& s, const Person& p) {
  s<<p.name; //friend can access private data
  return s;
}
```

The question that we have to ask now is the following: What do the default copy constructor and the default assignment operator do? We know from before that both perform a memberwise copy. Since class Person has only one member datum, the name, the default copy constructor copies the name from object **saad** to object **clone1**. Similarly, the default assignment operator copies the name from object **saad** to object **clone2**.

But what does copying the name mean? In general, here's the rule depending on the type of the member datum:

- The member datum is a built in type, a pointer, or a reference: copying is identical to a simple assignment operation
- The member datum is an instance of a class: copying is done using the copy constructor of the class (default copy constructor if none is defined)
- The member datum is a static array (size known by compiler): copying is done on each element in a manner appropriate to its type

Since the member datum **name** is a static array, each element of name is copied separately. But each element is a character (built in type); therefore, copying the name is equivalent to a simple assignment on a character by character basis. As a result, we have the correct behavior that we expect from a Person class.

But there is one small problem. One might argue that the 100 characters are enough to fit any name. That is likely true; however, we can't claim that this is completely safe. Bill Gates once said "*640K ought to be enough for anybody*". Maybe a person with a more than 100 character name exists! To be completely safe, we must declare name as a pointer and use dynamic memory allocation:

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char * name;

 public:
  Person(const char * s) {
    name=new char[strlen(s)+1];
    strcpy(name,s);
  }

  Person() {
    name=new char[5];
    strcpy(name,''john'');
  }
};

ostream& operator<<(ostream& s, const Person& p) {
  s<<p.name; //friend can access private data
  return s;
}

int main() {
  Person saad=Person(''saad'');
  Person clone1=saad;
  Person clone2;
  clone2=saad;
}
```

With this modification, we now have three issues to consider:

- We must free the memory that we allocated for each Person object when this object is no longer in use
- The member datum **name** is now declared as a pointer, so the default copy constructor will now simply assign pointers (see Section 2 on assignment and pointer semantics)
- Same point above holds for the default assignment operator

The following figure illustrates the problem introduced by dynamic memory allocation as exhibited by the default copy constructor and the default assignment operator:

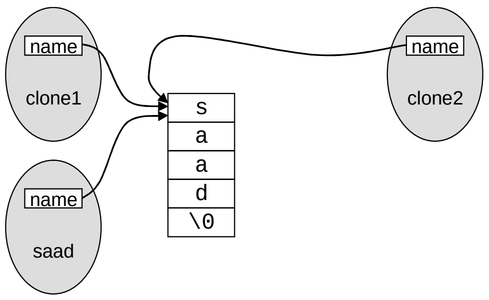


Figure 11.3: Clones are not really clones

We will see how to deal with the three issues mentioned above in the next chapter.

Chapter 12

Destructor, copy constructor, and assignment operator

12.1 A Person class

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char * name;

 public:
  Person(const char * s) {
    name=new char[strlen(s)+1];
    strcpy(name,s);
  }

  Person() {
    name=new char[5];
    strcpy(name,''john'');
  }
};

ostream& operator<<(ostream& s, const Person& p) {
  s<<p.name; //friend can access private data
  return s;
}
```

We have seen last time that we must deal with three issues:

- We must free the memory that we allocated in the constructors for each Person object when this object is no longer in use
- The member datum **name** is declared as a pointer; therefore, the default copy constructor will now simply assign pointers when performing the memberwise copy
- Same point above holds for the default assignment operator

```
int main() {
  Person saad=Person(’’saad’’);
  Person clone1=saad; //default copy constructor
  Person clone2;
  clone2=saad; //default assignment operator
  //memory is not freed
}
```

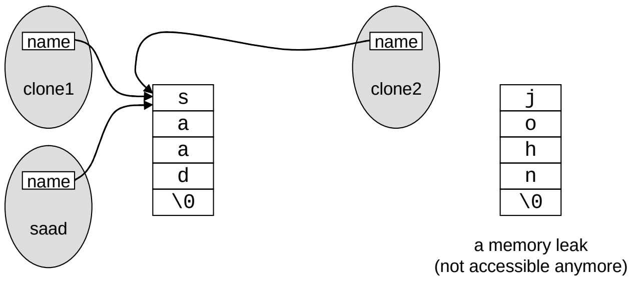


Figure 12.1: Clones are not really clones

12.2 Declaring the destructor

When constructing a Person object, we dynamically allocate memory. Therefore, an important question is the following: when should we de-allocate or free that memory? The obvious answer is of course whenever that Person object is no longer in use. But how do we actually determine that? When defining the class, we have no control over times when objects are instantiated nor on how long they live. Therefore, we need some help from the compiler.

C++ guarantees to call a special member function in the class right before an object goes out of scope (that’s when that object can be guaranteed to be no longer in use [1]). This special member function is called the *destructor*. Every class has a default destructor that does nothing; however, we can redefine the destructor and use it to free the memory allocated by the constructor. The destructor is identified by C++ according to the following:

- The destructor has the same name as the class name preceded by the ’˜’ symbol
- The destructor takes no parameters
- Like a constructor, the destructor returns nothing, not even void

From the second point above, we conclude that a destructor cannot be overloaded, i.e. we can have at most one destructor for a class (otherwise, the multiple destructors will have identical list of parameters).

Let us now declare our destructor for the Person class:

[1]This is not true for objects that are dynamically allocated using the new operator.

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char * name;

 public:

  . . .

  ~Person() {
    delete[] name;
  }
};
```

The following code:

```
int main() {
  Person saad=Person(''saad'');
}
```

is equivalent to:

```
int main() {
  Person saad=Person(''saad'');
  saad.~Person(); //this is implicitly added by compiler
}
```

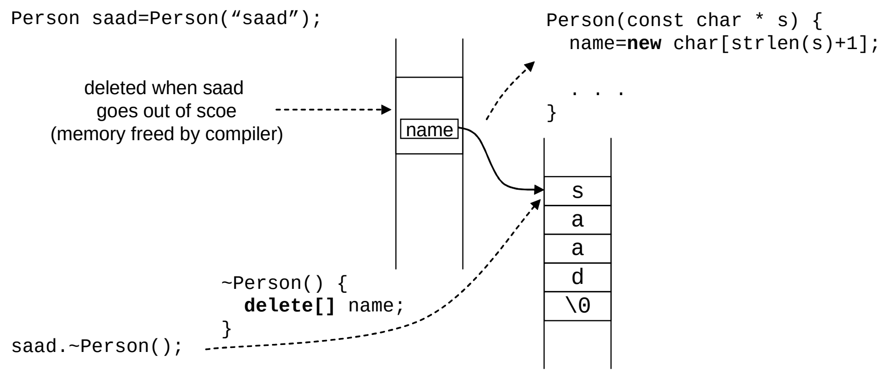

Person saad=Person("saad");
Person(const char * s) {
name=new char[strlen(s)+1];
. . .
}
deleted when saad goes out of scoe (memory freed by compiler)
name
s
a
a
d
\0
~Person() {
delete[] name;
}
saad.~Person();

Figure 12.2: One new[] and one delete[]

For dynamically allocated objects (declared using the new operator), the destructor is not called until that object is explicitly freed (using the delete operator). This is because a dynamically allocated object must live until explicitly deleted.

For instance, the following code:

```
int main() {
  Person * saad=new Person(''saad'');
}
```

is equivalent to:

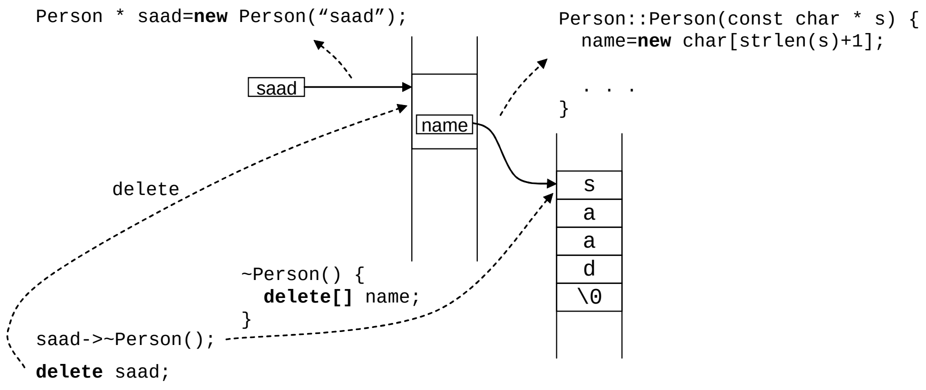
```
int main() {
  Person * saad=new Person(''saad'');
}
```

But the following code:

```
int main() {
  Person * saad=new Person(''saad'');
  delete saad;
}
```

is equivalent to:

```
int main() {
  Person * saad=new Person(''saad'');
  saad->~Person();
  delete saad; //by now there are two news and two deletes
}
```



Person * saad=new Person("saad");
Person::Person(const char * s) {
  name=new char[strlen(s)+1];
  . . .
}
saad
name
s
a
a
d
\0
delete
~Person() {
  delete[] name;
}
saad->~Person();
delete saad;

Figure 12.3: Two news and two deletes

In case of arrays, the destructor is called on every element of the array. The following code:

```
int main() {
  Person people[10]; //default constructor
}
```

is equivalent to:

```
int main() {
  Person people[10]; //default constructor
  for (int i=0; i<10; i++)
    people[i].~Person();
}
```

Similarly, the following code:

```
int main() {
  Person * people=new Person[10]; //default constructor
  delete[] people;
}
```

is equivalent to:

```
int main() {
  Person * people=new Person[10]; //default constructor
  for (int i=0; i<10; i++)
    people[i].~Person();
  delete[] people;
}
```

While it is possible to explicitly call the destructor (as a public member function), it is not recommended to do so. The reason is that C++ will call the destructor again when the object is out of scope (or deleted in case of a dynamically allocated object). This means that memory will be freed twice. The behavior of such a code is undefined and may cause a run time error. Therefore, it is a good idea to **NULL the pointer after deletion**. This means to set the value of the pointer to 0. Deleting a NULL pointer causes no harm (no action is done).

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char * name;

 public:

  . . .

  ~Person() {
    delete[] name;
    name=0;
  }
};
```

Finally (and somewhat related to the issue above), there is a small problem with the way we defined the destructor for class Person. Consider the following program again:

```
int main() {
  Person saad=Person(''saad'');
  Person clone1=saad; //default copy constructor
  Person clone2;
  clone2=saad; //default assignment operator
}
```

which is equivalent to:

```
int main() {
  Person saad=Person(''saad'');
  Person clone1=saad; //default copy constructor
  Person clone2;
  clone2=saad; //default assignment operator
  saad.~Person();
  clone1.~Person();
  clone2.~Person()
}
```

Since **saad**, **clone1**, and **clone2** share the same string (i.e. **saad.name**, **clone1.name**, and **clone2.name** are all equal, see Figure 1), the memory for the string "saad" if freed three times (once per each destructor call). Nulling the pointer in this case does not help because each object has its own pointer. This problem will disappear once we redefine the copy constructor and overload the assignment operator to produce real clones.

12.3 Declaring the copy constructor

One might argue that the default scenario of Figure 1 (except for the memory leak produced by assignment) is acceptable and, therefore, there is no need to declare the copy constructor. In deed, one might live with the fact that modifying the name of a clone automatically modifies the name of the original object (and vice-versa). However, with the newly declared destructor (see above), such a behavior becomes dangerous. Consider for instance the following code:

```
void dummy(Person p) {
}

int main() {
  Person saad=Person(''saad'');
  dummy(saad);
  cout<<saad;
}
```

This code looks very "innocent". The body of the function **dummy** is empty and, therefore, nothing is done (hence the name of the function). Consequently, one might naively think that this code is simply equivalent to the following:

```
int main() {
  Person saad=Person(''saad'');
  cout<<saad;
}
```

Unfortunately, it's not! Let's examine what really happens: When **dummy** is called with object **saad** as argument, the local parameter **p** of the function is constructed from **saad** using the copy constructor (the default one in this case). Now **p** and **saad** share the same string, i.e. **p.name==saad.name**. The parameter **p** goes out of scope when function **dummy** returns; therefore, **p.~Person()** (the newly declared destructor) is called. This means **p.name** is freed, which also means that **saad.name** is freed. As a result, when function **dummy** returns, object **saad** has no name. The statement **cout<<saad;** is likely to produce a run-time error because **operator<<** will be accessing a place in memory (i.e. pointer **saad.name**) that is not properly allocated anymore.

```
void dummy(Person p) {
  //p is constructed using
  //the default copy constructor

  //p is destructed using
  //the declared destructor
  p.~Person();
}

int main() {
  Person saad=Person(''saad'');
  dummy(saad);
  //saad lost its name
  cout<<saad;
}
```

To avoid such anomalies, the copy constructor must be properly declared and defined to allocate separate memory for the newly constructed object, and to copy the string pointed to by **name** character by character to the newly allocated memory. Since a copy constructor is a constructor after all, it must have the name of the class and no return type. But how can a constructor be identified as the copy constructor. The only way is by its list of parameters. C++ recognizes a copy constructor as a constructor that takes a constant reference to the class (why constant? and why reference?), i.e. in our case it must be declared as:

```
Person(const Person&);
```

In general, for a class **X**, the copy constructor is: **X(const X&)**. Let's declare a copy constructor for our Person class:

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char * name;

 public:

  . . .

  Person(const Person& p) {
    name=new char[strlen(p.name)+1];
    strcpy(name,p.name);
  }

  . . .
};
```

Note that we can access **p.name** inside the constructor even if **name** is declared as a private member: the unit of protection is the class and not the object.

12.4 Overloading the assignment operator

Referring to Figure 1, there are two problems to consider with the default assignment operator. First, after an assignment, objects share the sane physical memory (memberwise copy, same as default copy constructor). Second, there is a memory leak produced as a result of this default operation. Therefore, we must overload the assignment operator and define it in a way to avoid the two problems mentioned above.

The first thing to note when overloading the assignment operator is that C++ requires that it must be a member function (hence enforcing the first operand to be an instance of a user-defined class). The second thing to note is that the assignment operator is a binary operator (it takes two operands). Hence, besides the implicit **this** parameter (the first operand) as a consequence of being a member function, it takes one additional parameter. Therefore, we are looking at something like the following:

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char * name;

 public:

  . . .

  ---?--- operator=(---?---) {

    . . .
  }

  . . .
};
```

Of course, the questions are now what type should the assignment operator take and what type should it return? To answer these questions we must determine how the assignment operator should be used. For instance, the following should work:

```
int main() {
  Person saad=Person(''saad'');
  Person clone; //default constructor
  clone=saad; //equivent to clone.operator=(saad);
}
```

So we would like the parameter to be of type Person. Moreover, it would be better if we pass the argument by reference for efficiency (to avoid calling the copy constructor). We conclude that the type of the parameter should be a reference to Person, i.e. Person&. However, the following code should also work:

```
int main() {
  const Person saad=Person(''saad'');
  Person clone; //default constructor
  clone=saad; //equivalent to clone.operator=(saad);
              //saad is const by clone is not
}
```

In other words, we must be able to pass a constant object as argument to **operator=**. Since a const argument cannot be passed for a non-const parameter, a better way is to declare the parameter as a constant reference. This is a good idea anyway because we have no intention to modify the argument in an assignment (the reference is for efficiency only). So far, we figured that much:

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char * name;

 public:

  . . .

  ---?--- operator=(const Person& p) {

    . . .
  }

  . . .
};
```

For the return type, one can simply choose void. In this case, however, the following code will fail:

```
int main() {
Person x;
Person y;
Person z=...;
x=y=z; //equivalent to x=(y=z);
       //equivalent to x.operator=(y.operator=(z));
}
```

If such chaining is to be accepted (it is for built-in types), then **operator=** must return a Person object, e.g. **y=z** returns either **y** or **z**. But to avoid calling the copy constructor, we can make the return type a reference as well; for instance, **operator=** could return the object on which it is invoked (**y** in the above example of **y=z**) by dereferencing the **this** pointer.

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char * name;
```

```
 public:

  . . .

  Person& operator=(const Person& p) {

    . . .
    return *this;
  }

  . . .
};
```

One might suggest to return the second operand instead as in the following:

```
Person& operator=(const Person& p) {

  . . .
  return p;
}
```

While conceptually both options are equivalent (because of the assignment), the second option does not compile. The reason is that the return type is **Person&** but **p** has type **const Person&**. Again, a const cannot be passed for a non-const. This leads to the following question: should the return type be **const Person&** instead? Well, not if the following code should work:

```
int main() {
  Person x;
  Person y;
  Person z=...;
  (x=y)=z;
}
```

If **x=y** returns a **const Person&**, then it cannot be assigned and the code fails to compile. Since such code compiles if **x**, **y**, and **z** were built-in types (although no one would actually write such code), it is a good idea to maintain this behavior for user-defined classes. We conclude that

X& operator=(const X&)

is the preferred prototype for the assignment operator for class **X**.

Now let us implement the assignment operator:

```
Person& operator=(const Person& p) {
  //free memory to avoid memory leak
  delete[] name;
  //allocate new memory
  name=new char[strlen(p.name)+1];
  //copy
  strcpy(name,p.name);
  //return for chaining to work
  return *this;
}
```

Finally, there is a small problem with the way we defined the assignment operator for class Person. Consider the following program:

```
int main() {
  Person saad=Person(''saad'');
  saad=saad;
  cout<<saad;
}
```

Obviously, once the memory for object **saad** is freed inside **operator=**, the information is permanently lost! Therefore, we must first check for self-assignment. Here's the final version (and the correct one):

```
class Person {
  friend ostream& operator<<(ostream& s, const Person& p);

  char * name;

 public:

  . . .

  Person& operator=(const Person& p) {
    if (this==&p)
      return *this;
    delete[] name;
    name=new char[strlen(p.name)+1];
    strcpy(name,p.name);
    return *this;
  }

  . . .
};
```

12.5 Points to remember

- Declare a destructor, a copy constructor, and an assignment operator whenever your class allocates memory dynamically
- Destructor: **~X()**
- Copy constructor: **X(const X&)**
- Assignment operator (preferred): **X& operator=(const X&)**
- Always NULL pointers when you can (at initialization and after delete)
- Always check for self-assignment in **operator=**
- If your class allocates memory dynamically but you don't want to declare a copy constructor or an assignment operator (object may be too complicated to copy or must be unique), then declare them as private (without defining them), C++ won't be looking for their definitions (why?):

```
class X {

   . . .

 private:
  X(const X&);
  X& operator=(const X&);

   . . .
};
```

Chapter 13

Multidimensional arrays

13.1 A bit of a review

We have seen previously how to declare a one dimensional array; for example, here's an array of ten integers:

```
int main() {
  int a[10];

  . . .
}
```

We have also argued that an array is simply a pointer to its first element and, as a consequence, the above array has type **int ***. Therefore, to pass an array as an argument to a function, we simply have to specify the correct type:

```
void f(int * a, int n) {
  //do something
}
```

The second parameter (the size) is needed because there is no way of determining the size of an array simply from a pointer to its first element. The above function can be also defined as follows:

```
void f(int a[], int n) {
  //do something
}
```

The syntax **int a[]** for a function parameter is the same as writing **int * a**. While a bit misleading, the following syntax can also be used:

```
void f(int a[10]) {
  //assume a has size 10
  //do something
}
```

The reason why this is misleading is that the 10 between brackets is meaningless. The compiler interprets the parameter **int a[10]** as **int a[]**, i.e. it does not enforce any type on the argument except it being a pointer to **int**. Therefore, the following code compiles perfectly:

```
void f(int a[10]) {
  //assume a has size 10
  //do something
}

int main() {
  int a[5];

  . . .

  f(a); //ok
}
```

We will see later how such type enforcement can be achieved. For now, the compiler treats an array as a pointer to its first element (does not care about its size). Therefore, the size of the array is usually provided as an additional argument when passing the array as an argument to a function (unless the size is known to the programmer in advance). Having said that, we must also emphasize the fact that the compiler must know the type of the individual elements of an array. This is particularly important when dealing with multidimensional arrays.

13.2 Tic-Tac-Toe

Suppose we want to create a tic-tac-toe game. Therefore, we need to represent a 3×3 grid of symbols (possibilities include ' ', 'X', and 'O'). Such a representation can be achieved using an array as follows:

```
int main() {
  char grid[9]; //9 characters

  . . .
}
```

So far, what was nice about arrays (one dimensional arrays that is) is their ability to capture our mental image, which is a linear sequence of things. While nine characters are all we need to store the state of a tic-tac-toe game, such a linear array is not quite the mental picture that we usually have for tic-tac-toe.

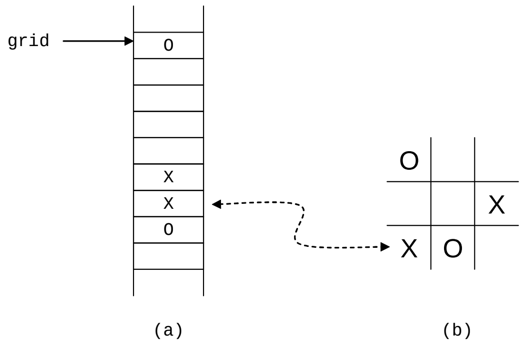


Figure 13.1: (a) Computer representation versus (b) mental image

As Figure 1 shows, we have to come up with some convention to transform the position of characters in the linear order to a position on the grid, and vice-versa. For instance, **grid[6]** is the bottom left corner. While such a convention is definitely possible, it may not be intuitive. Moreover, it may require some extra work; for instance, what does a row of X's mean in the linear order?

The problem here is that the linear array does not provide an appropriate *mapping* between the representation of a grid and the notion of a grid. It would be nice if the representation allows us to say: "the first element of the last row". But such a statement is meaningful only because we are viewing the grid as a two dimensional object. Fortunately, we can declare an array to capture this dimensionality. We simply have to declare an array of three elements, with each element being itself an array of three characters. Such a two dimensional array can be declared as follows:

```
int main() {
  int grid[3][3]; //grid is an array of 3 elements
                  //each element is an array of 3 characters


  . . .
}
```

The first element of the last row is now simply **grid[2][0]**. This is because **grid[2]** is the last element of the array declared above. Since each element is itself an array, **grid[2][0]** is the first element of the last row. In general, **grid[i][j]** refers to the j^{th} element of the i^{th} row (both rows and columns start with zero). Note that we can switch the notion of row and column in our mind without affecting the representation (and this is not simply because we have three rows and three columns).

The two dimensional array provides now a better mapping between the representation of a grid and the notion of a grid. In fact, the use of the term *mapping* here is essential. The representation itself did not change tremendously because memory inside the machine is still linear. We cannot make the memory representation live up to our mental image. Consider the following "fancy" tic-tac-toe mental representation.

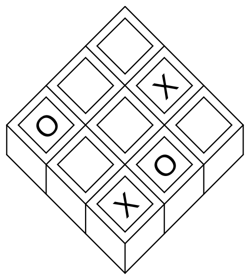


Figure 13.2: A fancier mental image of tic-tac-toe

Obviously nothing can change about the memory representation to reflect such "fanciness". So how is the grid really represented in memory? The following figure illustrates how.

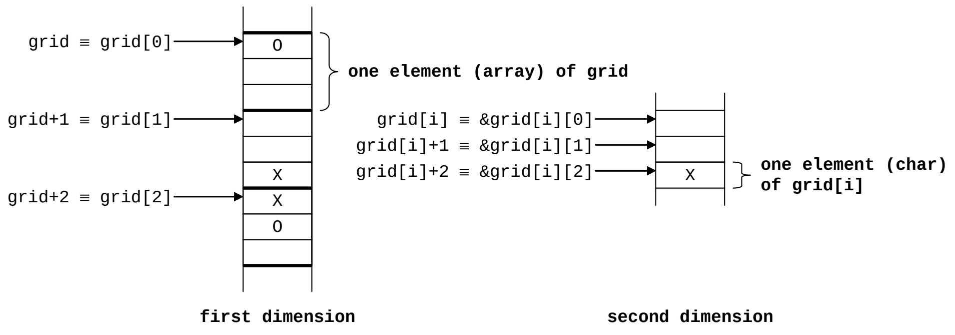


Figure 13.3: Actual (linear) representation of a two dimensional array

As shown in Figure 3, the best way to interpret a two dimensional array is as an array of arrays. Similarly, a three dimensional array is an array of arrays of arrays, and so on. In Figure 3, **grid+i** and **grid[i]** are pictured to be equivalent (but we learned that ***(grid+i)**≡ **grid[i]**). Here's an explanation: **grid[i]** is the i^{th} element of **grid**. Therefore, **grid[i]** is an array, and hence a pointer. As a result, **grid[i]** points to the same memory location as **grid+i**.

While **grid+i** and **grid[i]** are equivalent as pointer values, they are not equivalent in types (recall ***(grid+i)**≡ **grid[i]**). The type of **grid+i** (the same as the type of **grid**) is a pointer to an array of three characters; however, **grid[i]** has the type pointer to a character, and it is the address of **grid[i][0]**.

13.3 Multidimensional arrays as arguments

In Section 1, we revisited the fact that the compiler treats an array as a pointer to its first element, irrespective of the size of the array. Therefore, when passing an array as argument, the type for the corresponding parameter must be **T ***, where **T** is the type of the individual elements of the array. The type **T** must be fully known by the compiler in order to perform correct pointer arithmetics as those shown in Figure 3, e.g. **grid[1]** is actually **grid+1**. For example, when passing **grid** as an argument to a function, the corresponding parameter must tell the compiler that every element of **grid** is an array of three characters. This is essential for the increment operation mentioned above. Therefore, while the size of the first dimension is irrelevant, the size of the second dimension becomes very important. The problem we are facing now is a syntactic one: How do we say "a pointer to an array of three characters"? In other words, how can we specify that the type **T** is an array of three characters?

```
void f(T * a, int n) {
  //do something
}
```

The most obvious way is to write:

```
void f(char[3] * a, int n) {
  //do something
}
```

Unfortunately, such a syntax is not allowed. Another attempt is to write the following:

```
void f(char * a[3], int n) {
  //do something
}
```

But this makes the parameter an array of pointers to characters, i.e. each element of the array is a pointer to a character (this would be appropriate for an array of strings). The fact that the array has a size of three is irrelevant.

It turns out that the correct way to specify the type is the following:

```
void f(char a[][3], int n) {
  //a is an array
  //the size of the first dimension is irrelevant
  //the size of the second dimension is 3
  //do something
}
```

Let's use our tic-tac-toe grid as argument in a meaningful way:

```
void init(char a[][3], int n) {
  for (int i=0; i<n; i++)
    for (int j=0; j<3; j++)
      a[i][j]=' ';
}

int main() {
  char grid[3][3];
  init(grid,3);

  . . .
}
```

Another way is to give the type "array of three characters" a name and use that name as a built in type. This can be achieved using the typedef keyword (a meaningful example follows).

```
typedef char symbol;   //symbol is simply another name for char
typedef symbol row[3]; //row is simply another name
                       //for ``array of 3 symbols''

const symbol X='X';
const symbol O='O';
const symbol EMPTY=' ';

void init(row * a, int n) { //or void f(row a[], int n) {
  for (int i=0; i<n; i++)
    for (int j=0; j<3; j++)
      a[i][j]=EMPTY;
}

int main() {
  row grid[3]; //array of 3 rows
  init(grid,3);

  . . .
}
```

In general, all dimensions except the first must be specified when the array is passed as argument. Here's a example:

```
void f(int a[][3][4], int n) {
  //do something;
}

int main() {
  int cube[2][3][4];

  . . .

  f(cube,2);
}
```

13.4 Initializing multidimensional arrays

Multidimensional arrays can be initialized in the same way one dimensional arrays are initialized. Here's an example:

```
int main() {
  int a[2][3][4]={{{1,2,3,4},{5,6,7,8},{9,10,11,12}},
                 {{13,14,15,16},{17,18,19,20},{21,22,23,24}}};

  . . .
}
```

In the above, **a** is an array that contains two elements, each of the two is itself an array that contains three elements, each of the three is itself an array that contains four integers. Since the representation of a multidimensional array in memory is actually linear, with elements stored in the same order listed above (see Figure 4), the compiler does not generally require the explicit braces in the initialization.

```
int main() {
  int a[2][3][4]={1,2,3,4,5,6,7,8,9,10,11,12,
                  13,14,15,16,17,18,19,20,21,22,23,24};

  . . .
}
```

13.5 Multidimensional arrays of unknown size

Similar to a one dimensional array, when the sizes of a multidimensional array are not known, we allocate the array dynamically. Since a two dimensional array is actually an array of arrays, for an $m \times n$ array of integers for instance, we declare the array as a pointer to a pointer to **int**. Each element of the two dimensional array is an array of integers and, therefore, has type **int ***. The two dimensional array itself is a pointer to the first element and, therefore, has type **int ****.

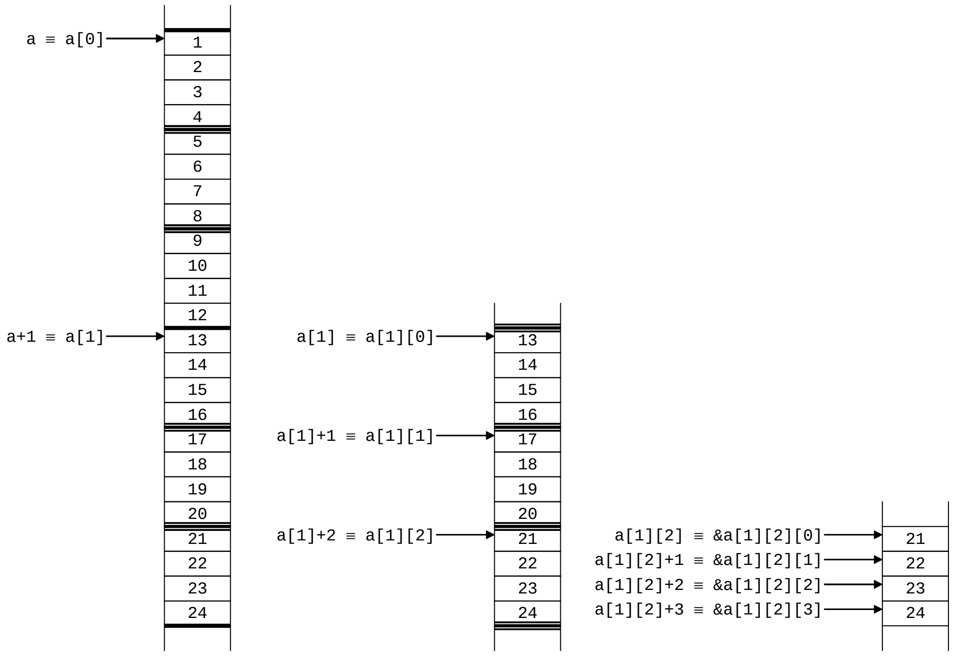


Figure 13.4: Memory representation of a[2][3][4]

The following code illustrates how to dynamically allocate an $m \times n$ array of integers.

```
int main() {
  int m,n; //to be determined

  . . .

  //allocate memory
  int ** grid=new int *[m];
  for (int i=0; i<m; i++)
    grid[i]=new int[n];

  . . .

  //free memory
  for (int i=0; i<m; i++)
    delete[] grid[i];
  delete[] grid;
}
```

Essentially, we first allocate an array of m pointers. For this we need a pointer (array) of type **int ****. Then, each one of the m pointers is dynamically allocated as an array of n integers (of type **int ***). Note that the m arrays of size n are not necessarily consecutive in memory as before. The following figure illustrates the memory representation of the $m \times n$ array thus obtained.

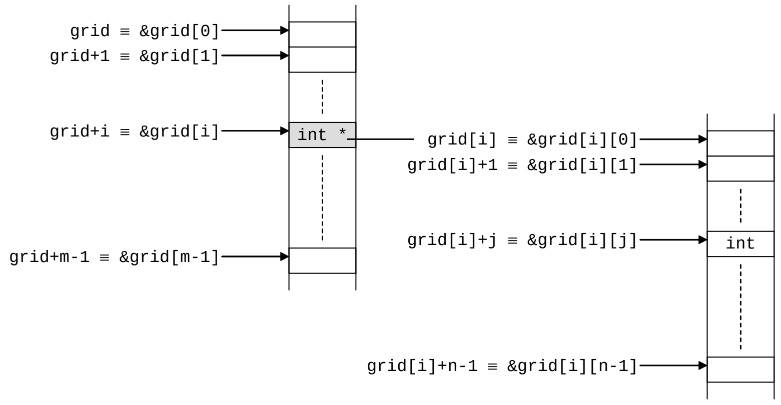


Figure 13.5: Dynamic allocation of an $m \times n$ array. Note that **grid+i** and **grid[i]** are no more equivalent as pointer values, **grid[i]** points to a memory location different from that of **grid+i**.

Chapter 14

C++ vectors, strings, and templates

14.1 The array decays into a pointer

So far, we have been treating an array as a pointer to its first element. As a direct result of this convention, there is no difference between the two following function declarations:

```
void f(int a[10]) {

  . . .
}

void f(int a[5]) {

  . . .
}
```

At the surface, it seems that the above two functions differ in their parameter list (namely the type of the parameter). However, the fact that the array is simply a pointer to its first element makes the size given between brackets irrelevant (see previous chapter). The compiler treats both of these functions as (thus a violation to the overloading rules):

```
void f(int a[]) {

  . . .
}
```

When passed as argument, we say that the array *decays* into a pointer. The pointer carries no information about the size of the array and, therefore, one should pass the size as an extra argument as follows:

```
void f(int a[], int n) {

  . . .
}
```

But what if we want the compiler to fully enforce the type on the argument? Say for instance that we want to define a function that takes as parameter an **array of integers of size 10**, i.e. the whole thing as one entity, not just the pointer. We will look at two ways for achieving this task: passing the array by reference and wrapping the array inside an object.

14.2 Passing arrays by reference

Because arrays are pointers, one might argue that arrays are always passed by reference. Well, yes and no: it is a pointer to the first element of the array that is being passed. But passing something by reference is to pass a constant pointer to it (even if itself is a pointer e.g. an array) while maintaining non-pointer syntax. It may be confusing to think about, but arrays can be passed by reference in that sense. Let's not worry about what this actually means. For our interest, the effect of passing an array by reference is to fully preserve the information about the type of the array. Here's an example of how to pass an array of ten integers by reference:

```
void f(int (&a)[10]) {

  . . .
}

int main() {
  int a[10];
  int b[5];
  f(a); //ok, correct type
  f(b); //error, wrong type
}
```

We can use the **sizeof** operator to observe the difference in the type of the argument when the array is passed as usual or by reference:

```
void f(int (&a)[10]) {
  //array does not decay into a pointer
  cout<<sizeof(a); //we will see 40 (each int is 4 bytes)

  . . .
}

void g(int a[]) {
  //array decays into a pointer
  cout<<sizeof(a); //we will see 4 (a pointer is 4 bytes)

  . . .
}

int main() {
  int a[10];
  cout<<sizeof(a); //we will see 40 (each int is 4 bytes)
  f(a);
  g(a);
}
```

14.3 A first look at templates

While we figured out a way to enforce the correct types on array arguments, we lost some flexibility. We now need a function for every possible array size!

```
void f(int (&a)[1]) {

  //process array of size 1
}

void f(int (&a)[2]) {

  //process array of size 2
}

    .
    .
    .

void f(int (&a)[100]) {

  //process array of size 100
}
```

It is of course impossible to cover all possibilities. But what if we can generate the appropriate function only when we need it? C++ provides such ability using what is known as a *template*.

```
template <int n>
void f(int (&a)[n]) {

  //process array of size n
}
```

The function defined above as a template represents an infinite number of functions, one for each possible value of n. The compiler however does not generate all of them (and it would be impossible to do so). When needed, the compiler performs the appropriate *specialization* of the template, i.e. determines a specific value for n (which we call the parameter of the template), and generates the corresponding function. As a result, we can now pass to the function an array of integers of any length.

```
int main() {
  int a[10];
  int b[5];
  f(a); //ok, specialize template with 10, equivalent to f<10>(a);
  f(b); //ok, specialize template with 5, equivalent to f<5>(b);
}
```

Using a template, we can go even one step further and make the function independent of the type of elements of the array.

```
template<class T, int n>
void f(T (&a)[n]) {

  //precess an array of Ts of size n
}
```

The above template has two parameters, a class **T** and an int **n**. The function thus defined takes an array of size **n**, each element of which has type **T**. The array is passed by reference as before.

```
int main() {
  int a[10];
  char b[5];
  f(a); //ok, equivalent to f<int,10>(a);
  f(b); //ok, equivalent to f<char,5>(b);
}
```

Note that for the compiler to specialize the template, **all the parameters of the template must be known at compile time**. Moreover, when arrays are declared dynamically, they are declared as pointers and, therefore, cannot be passed as arguments to a function expecting a reference to an array.

```
int main() {
  int n;

  . . .

  int * a=new int[n];
  f<int,n>(a); //error, n is unknown at compile time
  int * b=new int[10];
  f<int,10>(b); //error, wrong type, b has type int *
}
```

A more general solution for handling arrays is to wrap the array inside an object as explained in the following section.

14.4 Wrapping the array inside an object

A general solution for passing an array as argument while ensuring that:

- the array does not decay into a pointer
- no need to worry about passing the correct size as extra argument
- the technique works with dynamically allocated arrays

is to wrap the array inside an object and passing the object around while it provides the appropriate public interface to access the array.

Here's an example of a wrapper for an array of integers:

```
class Array {
  int * a; //the real array
  int s;   //the size of it

 public:
  Array() {
    s=0;
    a=0; //NULL pointer
  }

  Array(int n) {
    s=n;
    if (s==0)
      a=0; //NULL pointer
    else
      a=new int[s];
  }

  Array(const Array& arr) {
    s=arr.s;
    if (s==0)
      a=0; //NULL pointer
    else {
      a=new int[s];
      for(int i=0; i<s; i++)
        a[i]=arr.a[i];
    }
  }

  Array& operator=(const Array& arr) {
    if (this==&arr)
      return *this;
    delete[] a;
    s=arr.s;
    if (s==0)
      a=0; //NULL pointer
    else {
      a=new int[s];
      for(int i=0; i<s; i++)
        a[i]=arr.a[i];
    }
    return *this;
  }

  ~Array() {
    delete[] a;
  }
  int size()const {
    return s;
  }

  int& operator[](int i) {
    return a[i];
  }

  //for const objects
  const int& operator[](int i)const {
    return a[i];
  }
};
```

The Array object can be used as follows (a small example):

```
void f(Array& a) {
  int n=a.size();

  //process an array of n integers

}

int main() {
  Array a=Array(10); //array of 10 integers

  //initialize
  for(int i=0; i<10; i++)
    a[i]=0;

  . . .

  f(a);
}
```

We can use a template to convert the above class to a general array object of any type.

```
template<class T>
class Array {
  T * a; //the real array
  int s; //the size of it

 public:
  Array() {
    s=0;
    a=0; //NULL pointer
  }
  Array(int n) {
    s=n;
    if (s==0)
      a=0; //NULL pointer
    else
      a=new T[s];
  }

  Array(const Array& arr) {
    s=arr.s;
    if (s==0)
      a=0; //NULL pointer
    else {
      a=new T[s];
      for(int i=0; i<s; i++)
        a[i]=arr.a[i];
    }
  }

  Array& operator=(const Array& arr) {
    if (this==&arr)
      return *this;
    delete[] a;
    s=arr.s;
    if (s==0)
      a=0; //NULL pointer
    else {
      a=new T[s];
      for(int i=0; i<s; i++)
        a[i]=arr.a[i];
    }
    return *this;
  }

  ~Array() {
    delete[] a;
  }

  int size()const {
    return s;
  }

  T& operator[](int i) {
    return a[i];
  }

  //for const objects
  const T& operator[](int i)const {
    return a[i];
  }
};
```

```
template<class T>
void f(Array<T>& a) {
  int n=a.size();

  //process an array of Ts of size n

}

int main() {
  Array<int> a=Array<int>(10); //array of 10 integers
  Array<char> b=Array<char>(5); //array of 5 characters

  . . .

  f(a);
  f(b);
}
```

C++ provides a **vector** class as part of the standard library, which is similar to the class defined above except that it is more detailed.

14.5 C++ vectors

C++ provides the following class:

```
template<class T>
class vector {

  . . .

 public:

  //constructors
  vector(); //constructs an empty vector
  vector(const vector& c); //copy constructor
  vector(int num, const T& val=T()); //constructs a vector of num
                                     //elements of type T each being
                                     //a copy of val or the default
                                     //for class T if val is not given

  . . .

  //operators
  T& operator[](int i); //returns a reference to the ith element
  const T& operator[](int i)const; //same but constant reference

  //operators =, ==, !=, <, >, <=, >= are also overloaded

  . . .
  //member functions
  int size()const; //returns the number of items in the vector
  bool empty()const; //returns true if the vector has no elements
  void push_back(const T& val); //adds an element to the end of the vector
  void pop_back(); //removes the last element of the vector

  . . .
};
```

We will illustrate how to use the vector class through the tic-tac-toe example:

```
int main() {
  vector<char> row=vector<char>(3,' ');
  vector<vector<char> > tictactoe=vector<vector<char> >(3, row);
}
```

The above example creates a vector of three characters called **row** and initializes its characters to the space character. Therefore, it uses the third constructor of the class with **T** being **char**, **num** being 3, and **val** being ' '. Then it creates a vector of three vectors of characters (two dimensional) and initializes each vector to **row**. This also used the third constructor but with **T** being **vector<char>**, **num** being 3, and **val** being **row**. Note the space between the two '>' in **vector<vector<char> >**. This is needed because othewise the compiler will think of >> as the shift operator (like the one used with an input stream, e.g. **cin**).

Another variation is to declare **tictactoe** as an empty vector and add **row** three times to it.

```
int main() {
  vector<char> row=vector<char>(3,' ');
  vector<vector<char> > tictactoe;
  for (int i=o; i<3; i++)
    tictactoe.push_back(row);
}
```

Because operator[] is overloaded for vectors, vectors can be used in the same way as arrays. Here's an example:

```
int main() {

  //yet another variation
  vector<vector<char> > tictactoe=vector<vector<char> >(3,vector<char>(3,' '));

  . . .

  for (int i=0; i<3; i++) {
    for (int j=0; j<3; j++)
      cout<<tictactoe[i][j]<<'' '';
    cout<<''\n'';
  }

  . . .
}
```

14.6 Arrays vs. vectors

One of the legitimate questions now is when to use arrays and when to use vectors. A vector has the added functionality to change its size (using **push_back()** and manage dynamic memory. Therefore, here's a rule of thumb:

- If the size of the array is known at compile time, and does not change at run time, use an array
- If the size of the array is not known at compile time, but does not change at run time, use either an array or a vector
- If the size of the array changes at run time, use a vector

14.7 Sorting with a template

In this example, we reimplement sorting using templates to sort elements of any type provided that operator < is properly overloaded for that type. We will also use vectors instead of arrays. Note that for functions that change the vector, we must pass the vector by reference; otherwise, passing the vector as a constant reference is a good idea for efficiency.

```
#include <vector>

using std::vector;

template<class T>
int minimum (const vector<T>& a, int start, int end) {
  int index=start;
  for (int i=start; i<=end; i++)
    if (a[i]<a[index]) //assumes type T overloads operator <
      index=i;
  return index;
}

template<class T>
void swap(vector<T>& a, int i, int j) {
  T temp=a[i];
  a[i]=a[j];
  a[j]=temp;
}

template<class T>
void sort(vector<T>& a) {
  int n=a.size();
  for (int i=0;i<n;i++)
    swap(a, i, minimum(a,i,n-1));
}
```

```
int main() {
  vector<int> a=vector<int>(10);
  a[0]=1;
  a[1]=2;
  a[2]=5;
  a[3]=4;
  a[4]=3;
  a[5]=8;
  a[6]=7;
  a[7]=6;
  a[8]=10;
  a[9]=9;
  sort(a);
}
```

To illustrate the power of templates, we can sort rational numbers using the same code, provided that the class Rat has overloaded operator <.

```
class Rat {

  . . .

 public:
  bool operator<(const Rat& r) {
    return (n*r.d<d*r.n);
  }

  . . .
};

int main() {
  vector<Rat> a=vector<Rat>(3); //assumes Rat has a default constructor
  a[0]=Rat(2,3);
  a[1]=Rat(1,2);
  a[2]=Rat(1,4);
  sort(a); //voila!
}
```

14.8 Person revisited

Recall our Person class from previous chapters and suppose that we would like to add a member function to return the person's name:

```
class Person {
  char * name;

 public:
  Person() {...}
  Person(const char * s) {...}
  Person(const Person& p) {...}
  Person& operator=(const Person& p) {...}
  ~Person() {...};
  char * getName()const {
    return name;
  }
};
```

The newly added function returns a pointer to private data (the string). This means that a code can now change the person's name through that pointer as shown below:

```
int main() {
  Person saad=Person(''saad'');
  char * s=saad.getName();
  s[0]=...;
  cout<<saad.getName(); //now changed
}
```

How can we avoid this problem of exposing private data while still being able to obtain a person's name? One may think of a number of solutions.

14.8.1 Make the private data constant

```
class Person {
  const char * name;

  . . .
};
```

In this case however, the characters of name cannot be assigned, not even in the constructor. For instance, the following code will fail to compile.

```
class Person {
  const char * name;

 public:
  Person() {
    name=new char[5];
    strcpy(name, ''john''); //error, name is pointer to const
  }

  . . .
};
```

14.8.2 Provide an external pointer

```
class Person {
  char * name;

 public:

  . . .
  void getName(char * s)const {
    strcpy(s, name);
  }
};

int main() {
  Person saad=Person(''saad'');
  char s[5];
  saad.getName(s);
  cout<<s;
}
```

The **getName()** function is changed to accept one parameter of type **char *** (a string). Then the person's name is copied into that string. The problem with this approach is that one needs to provide a large enough string and, therefore, knowing the length of the person's name becomes crucial. While this is possible to determine by adding another member function for the Person class, it is inconvenient and prone to error.

```
class Person {
  char * name;

 public:

  . . .

  int length()const {
    return strlen(name);
  }

  void getName(char * s)const {
    strcpy(s, name);
  }
};

int main() {
  Person saad=Person(''saad'');
  char * s=new char[saad.length()+1]; //make sure to do this step
  saad.getName(s);
  cout<<s;
}
```

14.8.3 Return another name

```
class Person {
  char * name;

 public:

  . . .
  char * getName()const {
    char * s=new char[strlen(name)+1];
    strcpy(s, name);
    return s;
  }
};
```

The person's name is copied into another **char *** variable which is dynamically allocated and returned. Such approach however will make it impossible to predict when this newly allocated variable should be freed. It becomes the responsibility of the class user to free the memory.

```
int main() {
  Person saad=Person(''saad'');
  char * s=saad.getName();
  cout<<s;

  . . .

  delete[] s; //weird, a delete for no obvious new
}
```

14.8.4 Return a constant

```
class Person {
  char * name;

 public:

  . . .

  const char * getName()const {
    return name;
  }
};
```

The **getName()** function returns the person's name as a **const char ***. Therefore, the user cannot attempt to change the characters through the obtained pointer.

```
int main() {
  Person saad=Person(''saad'');
  char * s=saad.getName(); //error, must be const
  const char * t=saad.getName(); //ok
  t[0]=...; //error, t is pointer to const
}
```

This might seem to be the perfect solution for our problem. However, not if someone knows how to "cast away" const:

```
int main() {
  Person saad=Person(''saad'');
  char * s=(char *)saad.getName(); //ok, cast into non-const
  s[0]=...;
  cout<<saad.getName(); //now changed
}
```

14.8.5 Define a string class

All of the proposed solutions above have their drawbacks. It seems that the best way is for a Person not to return its name at all. Why would anyone want to retrieve the name? To print it? Then let Person provide a member function to print its name. To compare it to another name? Then let Person provide a member function to do it. That member function would takes another Person as a parameter and compare the names. Whatever functionality we want, we can make Person do it. However, soon Person will become a complicated class for handling strings. But that's the key to the solution: Provide such a class and let the **name** member datum of Person be an object of that class (instead of a **char ***). The string class would wrap the **char *** pointer in a similar way to an array class.

```
class String {
  friend ostream& operator<<(ostream&, const String&);

  char * str; //the real string
  int len;  //the length of it

 public:
  String() {
    len=0;
    str=0; //NULL pointer
  }

  String(const char * str) {
    len=strlen(t);
    if (len==0)
      str=0; //NULL pointer
    else {
      this->str=new char[len+1];
      for(int i=0; i<len+1; i++)
        this->str[i]=str[i];
    }
  }
  String(const String& s) {
    len=s.len;
    if (len==0)
      str=0; //NULL pointer
    else {
      str=new char[len+1];
      for(int i=0; i<len+1; i++)
        str[i]=s.str[i];
    }
  }

  String& operator=(const String& s) {
    if (this==&s)
      return *this;
    delete[] str;
    len=s.len;
    if (len==0)
      str=0; //NULL pointer
    else {
      str=new char[len+1];
      for(int i=0; i<len+1; i++)
        str[i]=s.str[i];
    }
    return *this;
  }

  ~String() {
    delete[] str;
  }

  int size()const {
    return len;
  }

  char& operator[](int i) {
    return str[i];
  }

  //for const objects
  const char& operator[](int i)const {
    return str[i];
  }
};

ostream& operator<<(ostream& os, const String& str) {
  os<<str.s;
  return os;
}
```

C++ provides a **string** class as part of the standard library, which is similar to the class defined above except that it is more detailed.

14.9 C++ strings

C++ provides the following class:

```
class string {

  . . .

 public:

  //constructors
  string(); //constructs an empty string
  string(const string& s); //copy constructor
  string(int len, const char& c); //len copies of c
  string(const char * str); //constructs a string from str
  string(const char * str, int len); //from str up to len characters
  string(const char * str, int index, int len); //same starting at index

  . . .

  //operators
  char& operator[](int i); //returns a reference to the ith character
  const char& operator[](int i)const; //same but constant reference
  string& operator=(const string& s);
  string& operator=(const char * str);
  string& operator=(char c);

  . . .

  //member functions
  int size()const; //returns the length of the string
  bool empty()const; //returns true if the string is empty
  string substr(int index, int num); //returns a substring with num
                                     //characters starting at index

  . . .
};

//concatenation operators
string operator+(const string& s1, const string& s2);
string operator+(const string& s, const char * str);
string operator+(const char * str, const string& s);
string operator+(const string& s, char c);
string operator+(char c, const string& s);

//relational operators (alphabetical comparison)
bool operator==(const string& s1, const string& s2);
bool operator!=(const string& s1, const string& s2);
bool operator<(const string& s1, const string& s2);
bool operator>(const string& s1, const string& s2);
bool operator<=(const string& s1, const string& s2);
bool operator>=(const string& s1, const string& s2);
```

```
//stream operators
ostream& operator<<(ostream& os, const string& s);
istream& operator>>(istream& is, string& s);
```

As an example of using C++ strings, let us now rewrite our Person class. For one thing, we do not have to manage the dynamic memory allocation of a **char *** string. This will be handled internally by the C++ string class. This means that we no longer need to declare a destructor, a copy constructor, and an assignment operator.

```
class Person {
  string name;

 public:
  Person() {
    name=''john'';
  }

  Person(const string& s) {
    name=s;
  }

  Person(const char * str) {
    name=str;
  }

  string getName()const {
    return name;
  }
};
```

Note that **getName()** now returns a string object. This means the copy constructor of class string is used to return **name** and, therefore, the user has no control over this private member datum.

```
int main() {
  Person saad=Person(''saad'');
  string s=saad.getName();
  s[0]=...; //changes s not saad.name;
  cout<<saad.getName(); //unchanged
}
```

14.10 Initialization vs. assignment

Consider our new Person class:

```
class Person {
  string name;

 public:
  Person() {
    name=''john'';
  }

  Person(const string& s) {
    name=s;
  }

  Person(const char * str) {
    name=str;
  }

  string getName()const {
    return name;
  }
};
```

This is the first time we have a user defined class in which one of the member data is itself an object of a user defined class. This raises an issue about construction: To construct a Person, one must construct a string. A Person object cannot exist without a string object as part of it. So how is that done? Here's the rule: **Before an object is constructed, each of its member data objects is constructed using the default constructor of the corresponding class** [1]. For our particular example, **name** is contructed first using the default constructor of class **string**, then the Person object is constructed using whatever constructor was invoked. This means that **name=...** in the above three constructors is actually an assignment of an empty string.

This seems a bit inefficient. The data member **name** is first constructed as an empty string, then assigned using the assignment operator of class **string**. This is not our idea of initialization. We would like to directly initialize **name** to the desired string. Such initialization (not assignment) can be achieved by overriding the rule stated above. In each constructor for Person, we can instruct the compiler what constructor to use for **name**, hence avoiding the default constructor and the need for the extra assignment. Here's the syntax:

```
class Person {
  string name;

 public:
  Person(): name(''john'') { //call constructor with const char *
  }

  Person(const string& s): name(s) { //call copy constructor
  }

  Person(const char * str): name(str) { //call constructor with const char *
  }
```

[1]Similarly, after the object is desctucted, the corresponding destructor for each of its member data objects is called.

```
  string getName()const {
    return name;
  }
};
```

Whenever possible, always prefer initialization over assignment.

Chapter 15

Recursive functions

15.1 A recursive process

Consider the factorial of an integer n. This is defined as

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

There are many ways to compute the factorial function. One way is by making the observation that $n! = n \cdot (n-1)!$:

$$\begin{aligned} n! &= n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 \\ &= n \cdot [(n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1] \\ &= n \cdot (n-1)! \end{aligned}$$

Therefore, to compute $n!$, we compute $(n-1)!$ and multiply the result by n. This gives rise to a recursive definition: we can compute $(n-1)!$ recursively in the same way. If we add the fact that $1! = 1$, then:

$$n! = \begin{cases} n \cdot (n-1)! & n > 1 \\ 1 & otherwise \end{cases}$$

Note that without $1! = 1$ (we call it the base case) the recursive definition is not complete. Every recursive definition must have a base case.

Recursive definitions can translate directly into recursive functions in any programming language. All we need is to declare a function **fact** that takes and integer n, and returns (an integer) n multiplied by the result of **fact**$(n-1)$. Therefore, the function calls itself recursively.

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

int main() {
  cout<<fact(6);
}
```

Using the substitution model described earlier (the parameter is replaced with the value of the argument in the body of the function), we can observe the recursive process generated by **fact(6)**.

```
fact(6)
6*fact(5)
  5*fact(4)
    4*fact(3)
      3*fact(2)
        2*fact(1)
          1
        2
      6
    24
  120
720
```

Figure 15.1: Recursive process

15.2 An iterative process

Another way to compute the factorial of a number n is by specifying that we first multiply 1 by 2, then multiply the result by 3, then by 4, and so on until we reach n. For this, we can maintain a running product and a counter that counts from 1 to n. Therefore, we update the product and the counter as follows:

$$product \leftarrow product \cdot counter$$

$$counter \leftarrow counter + 1$$

Both the product and the counter can start at 1. Finally, $n!$ will be the value of the product when the counter reaches $n+1$. While most of you are familiar with this iterative process, it can also be expressed recursively. The following is a recursive function implementation of this iterative process.

```
int fact_iter(int product, int counter, int n) {
  if (counter<=n)
    return fact_iter(product*counter,counter+1,n);
  else
    return product;
}

int fact_iter(int n) {
  return fact_iter(1,1,n);
}

int main() {
  cout<<fact_iter(6);
}
```

The idea is basically to carry forward the new values for all parameters as arguments for the recursive function call. Let us observe the process generated by **fact_iter(6)** as we did in the previous section.

```
fact_iter(6)
fact_iter(1,1,6)
fact_iter(1,2,6)
fact_iter(2,3,6)
fact_iter(6,4,6)
fact_iter(24,5,6)
fact_iter(120,6,6)
fact_iter(720,7,6)
720
```

Figure 15.2: Iterative process

15.3 Recursive vs. iterative (tail recursive)

Let us compare the two processes described in the previous sections. They both compute the same mathematical function using a recursive implementation. [1] Moreover, they both require a number of steps proportional to n. We use the notation $\Theta(n)$ ("theta" notation) to express that the running time of both processes is proportional to n. In fact, they even perform the same sequence of n multiplications obtaining the same sequence of partial products: 1, 2, 6, 24, 120, and 720. However, they evolve in different "shapes".

The recursive process reveals a shape that grows then shrinks. By contrast, the iterative process does not grow and shrink. We will see shortly that the latter can be transformed into a standard loop iteration.

With the recursive process, the shape grows while the process builds up a chain of deferred operations (multiplications in this case). More precisely, a multiplication cannot be performed until the second operand is determined, which does not happen until the recursive call returns. The shape then shrinks while the multiplications are actually performed. The implication of such a process is that the compiler must keep track of the operations to be performed later on. In particular, the compiler must keep track of values of parameters for every recursive call. For instance, in the example of Figure 1, when **fact(5)** returns, the compiler must recall that 6 was the value for n prior to that point in time. Similarly, when **fact(4)** returns, the compiler must recall that 5 was the value for n prior to that point in time, and so on. As a result, the compiler must store all values of n until the base case is reached. This means that the amount of memory needed is proportional to n, or $\Theta(n)$.

In general, compilers use a stack to store this information. When the function is called, the values of its parameters are pushed onto the stack. When the function returns, the values of its parameters are popped off the stack. The compiler uses the values stored at the top of the stack to access the parameters at any point in time. This stack is usually referred to as the "call stack".

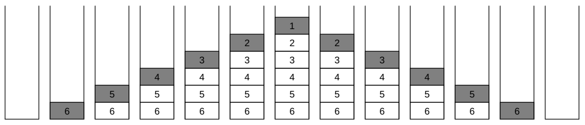


Figure 15.3: The call stack for 6!

[1]It is important to distinguish between a recursive process (semantics) and a recursive function (syntax). In fact, a recursive function may represent an iterative process (Section 2).

With the iterative process, the compiler needs to remember only the current values of the parameters: product, counter, and n. This is due to the fact that **the recursive call is the last thing** performed by the process. This type of recursion is called *tail recursion*: there is nothing left to do after the recursive call returns. Therefore, the compiler does not really need to recall anything. In fact, the current values of product, counter, and n completely determine the state of the computation. If we stop the computation at any point in time, and forget all the history, we can simply resume by starting from those values. This is not the case for the recursive process (we must remember where we stopped). While in general compilers still use the call stack, a "smart enough" compiler will detect tail recursion. In this case, the amount of memory needed is a constant independent of n (only store the current values of the parameters), or $\Theta(1)$.

Compilers with the above property are said to use a tail recursive implementation. With the tail recursive implementation, the compiler transforms the recursive function into a standard loop iteration (thus eliminating the stack). To transform a tail recursion into a loop, the following is done:

- change the **if** to a **while** and drop the **else** if any
- eliminate the **recursive call** by using **assignment** to update the parameters instead of passing them as arguments

Here's a tail recursive implementation of fact_iter.

```
int fact_iter(int product, int counter, int n) {
  while (counter<=n) {
    product=product*counter;
    counter=counter+1;
  }
  return product;
}

int fact_iter(int n) {
  return fact_iter(1,1,n);
}
```

This should now look like a familiar implementation. It is nothing but the iterative version of a tail recursive function. Here's a comparison among the alternative implementations of factorials:

	recursive	iterative	tail recursive
time	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
space	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

So why use recursion at all? The answer to this question is that recursion is sometimes the only natural way to think about a solution for the problem, and gives a simple and elegant code.

15.4 Bring on the rabbits

In 1200 AD, the italian mathematician Leonardo De Piza, also known as Leonardo Fibonacci (don't confuse him with Leonardo Da Vinci who is more recent), formulated a problem on rabbits. His formulation lead to the conclusion that the number of rabbits grows according the the following sequence, which has become to be known as the Fibonacci sequence.

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Therefore, the Fibonacci of an integer n is defined as:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n > 1 \\ 1 & otherwise \end{cases}$$

To illustrate the power of recursion, here's a recursive implementation of Fibonaccis which is a direct translation from the mathematical definition:

```
int fib(int n) {
  if (n>1)
    return fib(n-1)+fib(n-2);
  else
    return 1;
}
```

This code is simple, elegant, clear, and self explanatory. But it is a terrible way of computing Fibonaccis. Let's look at the recursive process generated by, say **fib(5)**. The **fib** function calls itself twice each time it is invoked, so we will end up with a tree like "shape" for the recursive process.

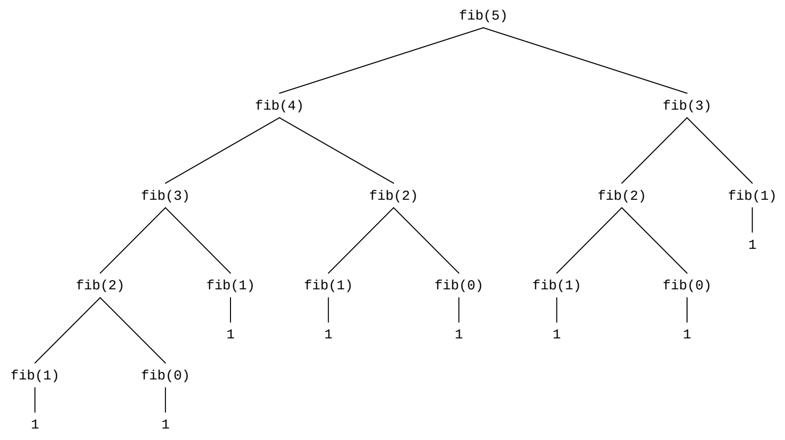


Figure 15.4: Tree recursive process of Fibonacci

Note that the Fibonacci of three is computed redundantly twice, which is almost half the work as seen by the above process. In general, the amount of work in a tree recursive process is proportional to the number of leaves in the tree. Similarly, the height of the call stack (amount of memory) is proportional to the height of the tree because we need to keep track of only the information above us in the tree at any point in the computation.

It can be shown that the number of leaves in this case is always equal to $Fib(n)$. It can be also shown that $Fib(n)$ is proportional to ϕ^n, where $\phi = 1.61803...$ is the golden number. Therefore, the running time of this implementation of Fibonacci is $\Theta(\phi^n)$, which is exponential in n (bad). By contrast, the height of the tree, and hence the required memory, is only $\Theta(n)$ (not that bad).

A better way is to think about an iterative process for Fibonacci which will lead to an implementation with $\Theta(n)$ running time and $\Theta(1)$ space. The idea is to come up with a number of variables that will capture the state of the computation at any point in time. If we keep track of the last two Fibonaccis in the sequence, we can always compute the next one. Therefore, a possible iterative process is to maintain two numbers a and b, and updates them as follows:

$$a \leftarrow b$$

$$b \leftarrow a + b$$

Starting with both $a = 1$ and $b = 1$, b will be $Fib(n)$ after repeating this process $n - 1$ times.

```
int fib_iter(int a, int b, int n) {
  if (n>1)
    return fib_iter(b,a+b,n-1);
  else
    return b;
}

int fib_iter(int n) {
  return fib_iter(1,1,n);
}
```

Let's look at the process generated by this implementation of the function for **fact(5)** (compare this to Figure 4).

```
fib_iter(5)
fib_iter(1,1,5)
fib_iter(1,2,4)
fib_iter(2,3,3)
fib_iter(3,5,2)
fib_iter(5,8,1)
8
```

Figure 15.5: Iterative process of Fibonacci

The number of steps required by this implementation is proportional to n. Therefore, this implementation has a $\Theta(n)$ running time and a $\Theta(n)$ space requirement (the height of the call stack). Since the recursive call is the last thing to be done, with a tail recursive implementation, only the current values of the parameters are stored and the space requirement reduces to $\Theta(1)$. Here's a tail recursive implementation following the guidelines listed in Section 3.

```
int fib_iter(int a, int b, int n) {
  while (n>1) {
    b=a+b;
    a=b-a;
    n=n-1;
  }
  return b;
}

int fib_iter(int n) {
  return fib_iter(1,1,n);
}
```

Here's a comparison among the alternative implementations of Fibonaccis.

	recursive	iterative	tail recursive
time	$\Theta(\phi^n)$	$\Theta(n)$	$\Theta(n)$
space	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

15.5 Exponentiation

Consider the problem of computing b^n for an integer base b and a non-negative integer exponent n. Again, using recursion, we first seek a simple direct translation from a mathematical definition to C++ code. Mathematically,

$$b^n = \begin{cases} b \cdot b^{n-1} & n > 0 \\ 1 & \textit{otherwise} \end{cases}$$

Here's a direct translation:

```
int exp(int b, int n) {
  if (n>0)
    return b*exp(b,n-1);
  else
    return 1;
}
```

This recursive process runs in $\Theta(n)$ time and $\Theta(n)$ space. Let us look now for an iterative process. We maintain a running product and a counter, and we update them according to the following rule until the counter reaches n:

$$\textit{product} \leftarrow \textit{product} \cdot b$$

$$\textit{counter} \leftarrow \textit{counter} + 1$$

Alternatively, we can use n itself as a counter and decrement it until it reaches zero. Starting with product$= 1$, and after n repetitions, product will be equal to b^n.

```
int exp_iter(int product, int b, int n) {
  if (n>0)
    return exp_iter(product*b,b,n-1);
  else
    return product;
}

int exp_iter(int b, int n) {
  return exp_iter(1,b,n);
}
```

This process runs in $\Theta(n)$ time and $\Theta(n)$ space. But it leads to a tail recursive implementation that uses only $\Theta(1)$ space (see Section 3 for the transformation).

```
int exp_iter(int product, int b, int n) {
  while (n>0) {
    product=product*b;
    n=n-1;
  }
  return product;
}

int exp_iter(int b, int n) {
  return exp_iter(1,b,n);
}
```

Here's a comparison among the alternative implementations of exponentiation:

	recursive	iterative	tail recursive
time	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
space	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

15.6 Tower of Hanoi

In 1833, the french mathematician Edouards Luca suggested the following problem inspired by an old legend about the tower of Hanoi. We are given a tower of n disks, initially stacked in decreasing size on one of three pegs, say A.

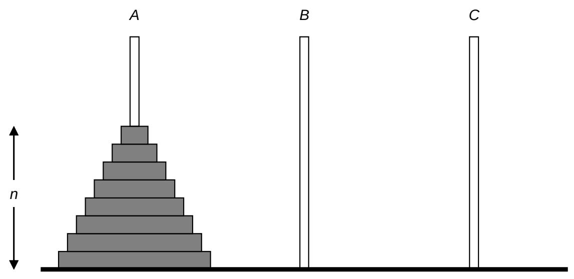


Figure 15.6: The tower of Hanoi

The objective is to transfer the entire tower to one of the other pegs, say C, using the other, say B, as a temporary peg, and obeying the following two rules of the game:

- only one disk at a time can be moved
- a larger disk can never be moved onto a smaller one

The original legend (most likely also invented by Luca) says that $n = 64$ and the disks are golden, and when all the disks are moved, the tower will collapse and the world will end. But not to worry. It can be shown that at least $2^n - 1$ moves are needed. For $n = 64$, that's $2^{64} - 1$ moves. If each move takes 1 second, that's 584.9 billion years (the universe began around 13.7 billion years ago).

The importance of this problem lies in the fact that the solution is naturally recursive. Therefore, being able to write a recursive function is almost the only practical way for solving this problem (of course every recursive implementation can be made non-recursive, just simulate the call stack with arrays or vectors).

While the solution to this problem seems to be very hard at first, we may assume that we know how to move a tower of $n - 1$ disks. Once we have done that, here's a solution for moving a tower of n disks.

- Move the first $n - 1$ disks from A to B using C as temporary
- Move the n^{th} disk from A to C
- Move the first $n - 1$ disks from B to C using A as temporary

Therefore, to move the n disks, move the first $n - 1$ disks first, then move the n^{th} disk, then move the first $n-1$ disks again. This suggests a tree recursive process in which the function calls itself twice each time it is invoked.

Let's start with a basic function to move a disk from one beg to another.

```
void move(char a, char c) {
  cout<<a<<''->''c<<''\n'';
}
```

Now onto the recursive implementation.

```
void hanoi(char a, char b, char c, int n) {
  if (n>1) {
    hanoi(a,c,b,n-1);
    move(a,c);
    hanoi(b,a,c,n-1);
  }
  else
    move(a,c);
}
```

The tower of Hanoi is one of the best classical problems to illustrate the power of recursion. One cannot hope for a better example of such a simple solution for such a hard problem. Nevertheless, we can make some enhancements. We can convert the tree recursive process into a linear recursive one by noting that the second recursive call is always the last thing to be done. Therefore, the second recursive call can be eliminated with a tail recursive implementation. Here's a tail recursive implementation following the guidelines listed in Section 3.

```
void hanoi(char a, char b, char c, int n) {
  while (n>1) {
    hanoi(a,c,b,n-1);
    move(a,c);
    char temp=a;
    a=b;
    b=temp;
    n=n-1;
  }
  move(a,c);
}
```

This runs in $\Theta(2^n)$ time ($2^n - 1$ moves are needed) and $\Theta(n)$ space (height of the call stack).

Chapter 16

Efficiency, doing it the *crazy* way

16.1 Introduction

In this chapter we are going to focus on "crazy" ways for doing things. Why say crazy? Because there exist ways that are much more direct and intuitive. So why do things the crazy way? Because it's going to be more efficient in terms of time. We are going to look at four crazy examples of doing things: exponentiation, testing primes, sorting, and searching (this introduction does not suggest that crazy always means efficient).

16.2 Exponentiation

We have seen before how to compute b^n for an integer base b and a non-negative integer exponent n. This was done in a straight forward way using the following recursive definition:

$$b^n = \begin{cases} b \cdot b^{n-1} & n > 0 \\ 1 & otherwise \end{cases}$$

Here's a direct translation of the mathematical definition:

```
int exp(int b, int n) {
  if (n>0)
    return b*exp(b,n-1);
  else
    return 1;
}
```

The "crazy" idea: Instead of performing n multiplications, why don't we use repeated squaring? Therefore, square b to compute b^2, then square the result to compute b^4, then square that to compute b^8, and so until we reach b^n. The number of multiplications (squares) we perform in this case is at most $\log_2 n$, because that's how many times we can divide n by 2 before reaching 1. Of course this works exactly only if n is a power of 2, e.g. 0, 2, 4, 8, 16, etc... But we can fix that by using the old rule for odd n and squaring for even n.

$$b^n = \begin{cases} [b^{n/2}]^2 & n\ even\ \neq 0 \\ b \cdot b^{n-1} & n\ odd \\ 1 & n = 0 \end{cases}$$

However, this at most doubles the number of multiplications from $\log_2 n$ to $2\log_2 n$, because if n is odd then $n-1$ is even. But this is still proportional to $\log_2 n$ (only a constant factor of 2). Therefore, we have $\Theta(\log_2 n)$ multiplication operations. The running time for finding b^n in this way is therefore $\Theta(\log_2 n)$ instead of $\Theta(n)$ for the standard way. Here's a direct translation of the new recursive definition:

```
int square(int x) {
  return x*x;
}

int fast_exp(int b, int n) {
  if (n>0)
    if (n%2==0)
      return square(fast_exp(b,n/2));
    else
      return b*fast_exp(b,n-1);
  else
    return 1;
}
```

16.3 Testing primes

To test whether a number n is prime, simply see if it admits any divisors in $\{2, \ldots \lfloor\sqrt{n}\rfloor\}$. This can be easily done in a loop:

```
bool prime(int n) {
  for (int i=2; i<=sqrt(n); i++)
    if (n%i==0)
      return false;
  return true;
}
```

The "crazy" idea: Instead of checking $\sqrt{n}$ numbers, and hence spending $\Theta(\sqrt{n})$ time, perform Fermat's test on few numbers. Huh? Of course this crazy idea seems to demand from us a little number theory. So here it is:

Fermat's little theorem: If n is prime, then any positive integer $a < n$ satisfies $a^{n-1}\%n = 1$.

So let's pick an integer $a < n$. If $a^{n-1}\%n \neq 1$, then Fermat tells us that n cannot be prime. That's good. If $a^{n-1}\%n = 1$, then Fermat does not really tell us that n is prime [1], but then we have a reasonable doubt that it is. In fact, if n were not prime, then this would be unlikely to happen. But to strengthen our belief, we can pick another integer $a < n$ and test whether $a^{n-1}\%n = 1$. We can repeat this Fermat test three or four times. If $a^{n-1}\%n = 1$ in every time, then we conclude that n is prime.

[1] If A implies B, then B does not necessarily imply A.

Of course there is still a tiny probability that we are wrong, but this idea gives a much faster way for testing primality because we can use **fast_exp()** to compute a^{n-1}. For illustration, here's a code to perform one Fermat test.

```
bool fast_prime(int n) {
  int a=rand()%(n-1)+1; //a in {1..n-1}
  if (fast_exp(a,n-1)%n==1)
    return true;
  else return false;
}
```

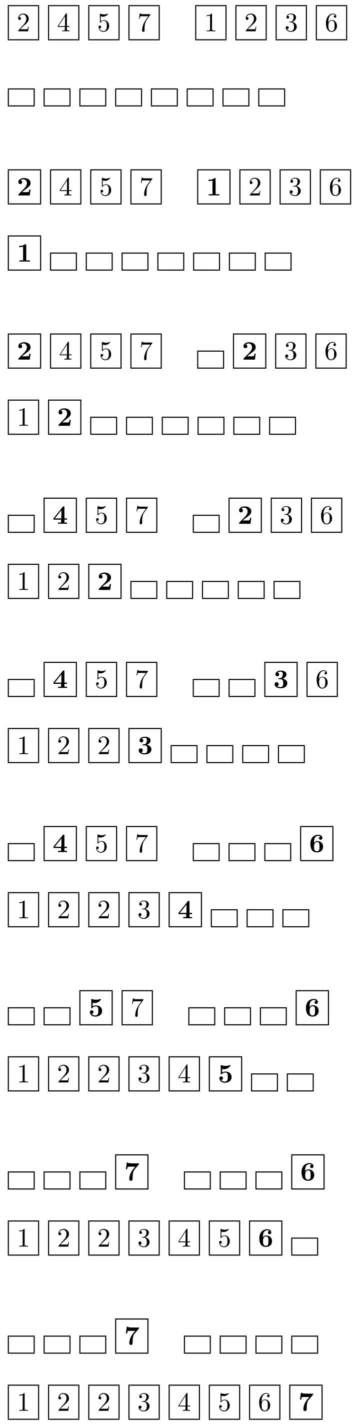
The function **fast_prime()** makes a one simple call to **fast_exp**$(a, n-1)$. Since **fast_exp()** runs in $\Theta(\log_2 n)$ time, then **fast_prime()** also runs in $\Theta(\log_2 n)$ time. While this is a much better running time compared to $\Theta(\sqrt{n})$, we have introduced a problem. Namely, a^{n-1} can be very large (a can be close to n) and, therefore, we risk the problem of overflow. Since we are interested in computing $a^{n-1}\%n$ and not a^{n-1}, we can solve this problem by making **fast_exp**$(a, n-1)$ compute all its intermediate results modulo n. Therefore, we change **fast_exp()** to take one additional parameter m and compute everything modulo m.

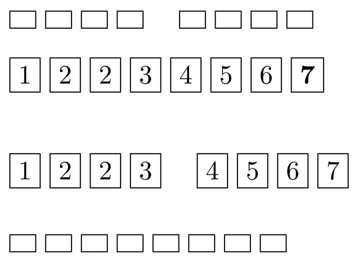
```
int square(int x) {
  return x*x;
}

int fast_exp(int b, int n, int m) {
  if (n>0)
    if (n%2==0)
      return square(fast_exp(b,n/2,m))%m;
    else
      return (b*fast_exp(b,n-1,m))%m;
  else
    return 1;
}

bool fast_prime(int n) {
  int a=rand()%(n-1)+1; //a in {1,...,n-1}
  if (fast_exp(a,n-1,n)==1)
    return true;
  else return false;
}
```

16.4 Sorting

Sorting an array of elements is a problem that we have seen before and solved using the idea of repeatedly finding the smallest element and swapping it with the first in successive parts of the array (see Note 6). The running time of such an algorithm is proportional to the square of the number of elements to be sorted. Therefore, we spend $\Theta(n^2)$ time to sort n elements. The reason for this is that we need to check all n elements to find the smallest. Having done that, we then need to check $n-1$ elements to find the second smallest, then $n-2$ elements for the third smallest, and so on until the elements are sorted. The total number of checks is $1 + 2 + \ldots + (n-1) + n = \frac{n(n+1)}{2} = \Theta(n^2)$.

The “crazy” idea: Assume that the array is such that the left half and the right half are sorted. Then sort the elements in $\Theta(n)$ time by merging the two halves of the array: Use an auxiliary array and repeatedly copy the smallest element of the two halves into successive positions. Then copy the auxiliary array back into the original one. Here’s an example:





The running time for this merging operation is $\Theta(n)$ because every time we make a comparison we copy one element into the auxiliary array. Therefore, we do this at most n times. Without worrying about the detail, we will assume the existence of a function to merge two sorted halves of the array (assume also an array of integers):

```
void merge(int * a, int p, int q, int r) {
  //merge a[p]...a[q] and a[q+1]...a[r]
}
```

This "crazy" idea assumed that each half of the array is sorted. But what if that's not the case? In fact, that's why the idea is "crazy", because now we are going to ensure that the two halves are sorted by performing the following:

- split each in two halves
- assume the two halves are sorted
- merge the two halves

The argument is carried recursively until each half is only one element and is, therefore, safe to be assumed sorted. This method for sorting is known as merge sort. To some extent, merge sort refrains from actually sorting. It only perform merges. But to ensure that the two halves are sorted and ready to be merged, it recursively uses merge sort on each one of them.

```
void merge_sort(int * a, int p, int r) {
  if (p<r) { //more than one element
    int q=(p+r)/2;
    merge_sort(a,p,q);
    merge_sort(a,q+1,r);
    merge(a,p,q,r);
  }
}

int main() {

  int a[n];

  . . .

  merge_sort(a,0,n-1);

  . . .
};
```

So what is the running time of merge sort? As we argued before, to merge the first two halves we need $\Theta(n)$ time. But before doing so, each half is merge sorted. This means each half produces two other halves (quarters) to be merged first. To merge these, we need $\Theta(n/2) + \Theta(n/2)$ time. That's $\Theta(n)$ time again. We can observe that at every level of the recursion tree, we need at most $\Theta(n)$ time. Therefore, we need a total time of $\Theta(n)$ times the number of levels. The number of levels is guaranteed not to be more than $\log_2 n$ because that's how many times we can divide n by 2 before reaching 1. We conclude that the running time is $\Theta(n \log_2 n)$ (compare this to the previous $\Theta(n^2)$ time).

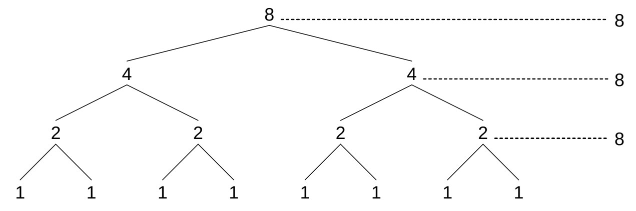


Figure 16.1: Number of elements merged at each level of the recursion tree starting with 8 elements

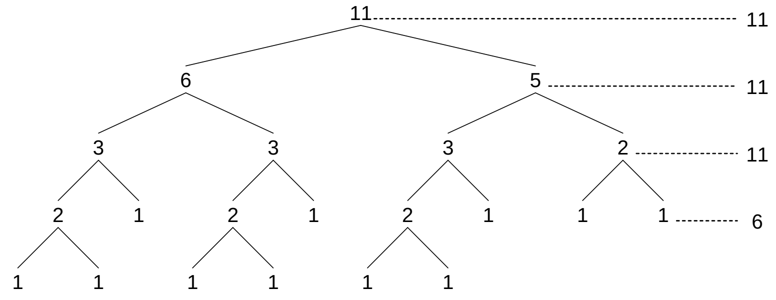


Figure 16.2: Number of elements merged at each level of the recursion tree starting with 11 elements

16.5 Searching

Finally, consider the problem of searching an array for a particular element, called the key. A simple way to search is to check every element against the key. If found, we return its position in the array; otherwise, we return -1.

```
int search(int * a, int n, int key) {
  for (int i=0; i<n; i++)
    if (a[i]==key)
      return i;
  return -1;
}
```

This algorithm runs in $\Theta(n)$ time because it will possibly check every element in the array. For instance, if the key is not an element of the array, the loop will be completely exhausted before we return.

The “crazy” idea: Assume the array is sorted and split it in two halves, determine which half may potentially contain the key, and only search that half. By now we have hopefully learned that such a crazy idea is not really crazy because we can apply it recursively. But what if the array is not sorted? Then we can sort it, say using merge sort above in $\Theta(n \log_2 n)$ time. But this alone is more than the trivial $\Theta(n)$ time presented above. So it is not justifiable. However, it we are performing multiple searches, we only need to sort the array once. Then every search will be efficient. Therefore, sorting will pay off. So let's assume the array is sorted and see how we can proceed from here. Assume the following strategy:

- check the key against the middle element
- if found, return that position
- if the key is smaller, then we only need to search the first half
- if the key is larger, then we only need to search the second half

We can recursively apply the same strategy on the successive halves. Therefore, for every check that we perform, we reduce the size of the array by half. Of course, we can do this at most $\log_2 n$ times before we reach an array of size 1. We conclude that in doing so, we only perform $\log_2 n$ checks and, therefore, our algorithm runs in $\Theta(\log_2 n)$ time. This algorithm is called binary search.

We need to make our search function now take extra parameters to determine which part of the array we need to recursively search (in a similar way to merge sort).

```
int binary_search(int * a, int p, int r, int key) {
  if (p<=r) { //at least one element
    int q=(p+r)/2;
    if (a[q]==key)
      return q;
    if (a[q]>key)
      return search(a,p,q-1,key);
    if (a[q]<key)
      return search(a,q+1,r,key);
  }
  else
    return -1;
}

int main() {
  int a[n];

  . . .

  merge_sort(a,0,n-1);

  . . .

  int pos=binary_search(a,0,n-1,key);

  . . .
};
```

This recursive search function is tail recursive because the recursive call always comes last. Therefore, we can eliminate recursion using the standard tail recursive implementation (change **if** to **while**, drop **else**, and replace recursive call with assignment).

```
int binary_search(int * a, int p, int r, int key) {
  while (p<=r) { //at least one element
    int q=(p+r)/2;
    if (a[q]==key)
      return q;
    if (a[q]>key)
      r=q-1;
    if (a[q]<key)
      p=q+1;
  }
  return -1;
}
```

Chapter 17

Happy Feet

17.1 Introduction

Consider our Person class as a motivating example:

```
class Person {
  string name;

 public:
  Person(): name(''john'') {}
  Person(const string& s): name(s) {}
  Person(const char * str): name(str) {}

  string getName()const {
    return name;
  }
};
```

In this chapter we address two problems:

- How can we keep track of the number of persons alive at any time in the program (static members)?
- How can we create a new class Student without having to reinvent a Person (inheritance)?

17.2 Static member data

Consider the problem of counting the number of persons that are alive at any time in the program. The reason why this may be needed is irrelevant at this point but, for instance, we can use it to discover potential memory leaks. This counting task may seen trivial at a first glance (after all counting is a basic thing): every time we declare a Person object, we increment a counter by 1.

```
int main() {
  int counter=0;
  Person p=...;
  ++counter;

  . . .
}
```

However, a deeper understanding of the problem will reveal some real difficulty. What if Person objects are allocated dynamically at run time?

```
int main() {
  int counter=0;
  int n;
  cin>>n;

  . . .

  Person * p=new Person[n];
  counter=counter+n;

  . . .
}
```

What about destruction? We need to also decrement the counter by 1 every time a Person object is destructed.

```
int main() {
  int counter=0;
  int n;
  cin>>n;

  . . .

  Person * p=new Person[n];
  counter=counter+n;

  . . .

  delete[] p;
  counter=counter-n;
}
```

But what if the destruction is handled automatically by the compiler, e.g. local non-dynamic allocation:

```
void f() {
  Person p=...;
  ++counter; //which counter?!

  . . .

  //must decrement counter here
  //because p is destroyed
}
```

And what about constructions that are automatically done by the compiler?

```
void f(Person p) {
  //p is constructed locally
  //using the copy constructor
  //number of persons here changed

  . . .

  //p is destructed after
  //function returns
  //number of persons here changed again
}

int main() {
  counter=0;
  Person p=...;
  ++counter;
  f(p);

  . . .
}
```

Obviously, there is plenty of room to make mistakes. It is hard to detect when the compiler is constructing temporary Person objects and when it is destructing them. Not to mention the messy code that will result from all the counter updates. Finally, counter will have to be declared as a global variable to be accessed anywhere in the program (bad, what if another piece of code declared the same global variable?).

Therefore, the best way to keep track of the number of persons in the program is to use a counter as part of class Person itself and maintain that counter inside the class. Here's a first attempt:

```
class Person {
  int counter;
  string name;

 public:

  . . .
};
```

But soon we realize that by making the counter one of the member data of class Person, ever person object will have its own counter. This definitely will not achieve the desired effect. Fortunately, we can tell the compiler that all Person objects must share the same counter. This is done by declaring counter as a *static member.*

```
class Person {
  static int counter;
  string name;

 public:

  . . .
};
```

A person object cannot be created without using a constructor. Similarly, a Person object cannot be "killed" without using a destructor. Therefore, all we need to do is maintain the counter appropriately in every constructor, and in the destructor. This means that we have to declare the copy constructor and the destructor in class Person (although we do not really need them except for updating the counter).

```
class Person {
  static int counter;
  string name;

 public:
  Person(): name(''john'') {
    ++counter;
  }
  Person(const string& s): name(s) {
    ++counter;
  }
  Person(const char * str): name(str) {
    ++counter;
  }

  Person(const Person& p): name(p.name) {
    ++counter;
  }

  ~Person() {
    --counter;
  }

  int count()const {
    return counter;
  }

  string getName()const {
    return name;
  }
};
```

Finally, we need to initialize the counter to 0. Static member data is initialized outside the class as follows:

```
int Person::counter=0;
```

Here's an example of using the modified Person class:

```
void f(Person p) {

  . . .

  cout<<p.count(); //will see 3

  . . .
}

int main() {
  Person saad=Person(''saad'');
  cout<<saad.count(); //will see 1
  Person john;
  cout<<saad.count(); //will see 2
  cout<<john.count(); //will see 2
  f(john);
  cout<<saad.count(); //will see 2
  cout<<john.count(); //will see 2
}
```

17.3 Static member functions

Consider the **count()** function above. This function behaves exactly the same way regardless of which object invokes it. This is because the function **count()** accesses only static member data. Therefore, it may be misleading (for a reader of the code) to invoke this function from multiple objects. For this reason, member functions of a class can also be declared static. When static, a member function looses its self reference pointer **this**. As a result, **a static member function cannot access non-static member data**. Furthermore, a static member function cannot be declared as **const**, since **const** refers to the **this** pointer (but can still be invoked on a constant object because it cannot access its non-static member data).

```
class Person {
  static int counter;
  string name;

 public:

  . . .                         //equivalent to

  static int count() {          string Person::count() { //no this pointer
    return counter;               return Person::counter;
  }                             }

  string getName()const {       string Person::getName(const Person * const this) {
    return name;                  return (*this).name;
  }                             }
};
```

While it would still be possible to invoke a static member function through an object, a static member function can be invoked through the class itself (a better way):

```
int main() {
  Person saad=Person(''saad'');
  Person john;
  cout<<saad.count(); //will see 2
  cout<<john.count(); //will see 2
  cout<<Person::count(); //will see 2
}
```

Public and private rules apply to static member data and static member functions as well. For instance, if **counter** were public in class Person, one could have done the following:

```
int main() {
  Person saad=Person(''saad'');
  Person john;
  cout<<saad.counter; //will see 2
  cout<<john.counter; //will see 2
  cout<<Person::counter; //will see 2
}
```

17.4 Inheritance

Consider the problem of creating a class Student. This of course can be easily achieved by repeating what we have done for class Person, and adding the extra information specific to a student. In doing so, however, we are reinventing the Person class. Moreover, creating Student objects will not appropriately update the Person count, after all, a student is a person.

Instead of creating a Student class from scratch, C++ provides a way to build on top of an existing class. This is motivated by the fact that a student **IS A** person. Therefore, we should reuse the Person class to create the Student class. This form of reuse is known as *inheritance*. More specifically, the **IS A** relation means *public inheritance*:

```
class Student: public Person {

  . . .
};
```

With public inheritance, we are telling the compiler that:

- Student is the "derived" class, and Person is the "base" class
- Person represents a more general concept than Student, and Student represents a more specific concept than Person
- Student has access to everything public in Person, and everything public in Person is also public in Student
- Every object of type Student is also an object of type Person, but not vice-versa

- Anywhere an object of type Person can be used, an object of type Student can be used just as well

Here's an example:

```
void play(const Person& p) {

  . . .
}

void study(const Student& s) {

  . . .
}

int main() {
  Person p=...;
  Student s=...;
  play(p); //ok
  play(s); //ok, a student is a person (true only with public inheritance)
  study(s); //ok
  study(p); //error, a person is not a student
}
```

Similarly, anywhere a pointer to Person can be used, a pointer to Student can be used just as well:

```
int main() {
  int n=...;
  bool party=...;
  Person * p;

  . . .

  if (party)
    p=new Person[n];
  else
    p=new Student[n];

  . . .
}
```

To create the Student class, we have to consider the following:

- The Student class can access only the public members (data or functions) of Person (why?)

- The Student class must declare its own constructors, and each constructor must also tell the compiler how to construct the Person part (or the default constructor for Person is used)

- The Student class must declare its own member data and functions

```
class Student: public Person {
  int id;

 public:
  Student() { //default constructor for Person used
    id=0;
  }

  Student(int i) { //default constructor for Person used
    id=i;
  }

  Student(const string& s, int i): Person(s) {
    id=i;
  }

  Student(const char * str, int i): Person(str) {
    id=i;
  }

  int getId()const {
    return id;
  }
};

ostream& operator<<(ostream& os, const Student& s) {
  os<<s.getName()<<'' ''<<s.getId();
  return os;
}

int main() {
  Student john;
  Student saad=Student(''saad'',1);
  cout<<Person::count(); //we will see 2
}
```

17.5 The IS A relation

Public inheritance should only be used when we have an IS A relation (e.g. a student is a person). The equivalence of public inheritance and IS A sounds simple; however, sometimes intuition can be misleading. For example, it is a fact that a penguin is a bird, and it is a fact that birds can fly. If we express this in C++, we get:

```
class Bird {
 public:
  void fly() {...}

  . . .
};
```

```
class Penguin: public Bird {

  . . .
};
```

Suddenly we have a problem: Penguins can fly!

```
int main() {
  Penguin p=...;
  p.fly(); //ok, but it shouldn't
}
```

When we say that buirds can fly, we don't really mean that all birds can fly, only that, in general, birds have the ability to fly. Therefore, a more precise model will recognize that there are several types of non-flying birds.

```
class Bird {

  . . .
};

class FlyingBird: public Bird {
 public:
  void fly() {...}
  . . .
};

class NonFlyingBird: public Bird {

  . . .
};

class Penguin: public NonFlyingBird {

  . . .
};
```

Similarly, it is a fact that penguins can sing, and it is a fact that Happy Feet is a penguin (well, it's controversial). If we express this in C++, we get:

```
class Penguin: public NonFlyingBird {
 public:
  void sing() {...}

  . . .
};

class HappyFeet: public Penguin {

  . . .
};
```

Suddenly we have another problem: Happy Feet can sing! Obviously this is not true according to the movie. A better implementation is the following:

```
class SingingPenguin: public Penguin {
 public:
  void sing() {...}

  . . .
};

class HappyFeet: public Penguin {

  . . .
}
```

Figure 17.1: Happy Feet

Here's another example: Is a square a rectangle? Mathematically speaking, the answer is yes. A square is a rectangle whose width and height are equal. If we express this in C++, we get:

```
class Rectangle {
  int w;
  int h;
 public:

  Rectangle(int w, int h) {
    this->w=w;
    this->h=h;
  }

  void setWidth(int w) {
    this->w=w;
  }

  void setHeight(int h) {
    this->h=h;
  }
  int width()const {
    return w;
  }

  int height()const {
    return h;
  }
};

class Square: public Rectangle {
 public:
  Square(int a): Rectangle(a,a) {}
};
```

Then we have a problem: something applicable to a rectangle, its width may be modified independently of its height, is not applicable to a square. But public inheritance says that everything applicable to a base class object -*everything*- is also applicable to a derived class object.

```
int main() {
  Square s=Square(5);
  s.setWidth(3); //ok, but it shouldn't be ok
};
```

17.6 IS A vs. HAS A

It is important to distinguish whether something IS A something else or HAS A (is implemented in terms of) something else. For instance, in our Person/Student example, a student is a person, but a person has a (is implemented in terms of) string.

```
class Person {
  string name;

  . . .
};

class Student: public Person {

  . . .
};
```

It is really bad to adopt the following implementation instead (why?):

```
class Person: public string {

  . . .
};

class Student {
  Person p;

  . . .
};
```

Consider creating a class Stack with two public member functions push() and pop(). We can use the C++ vector class as a base class for Stack:

```
template<class T>
class Stack: public vector<T> {
 public:
  Stack() {}
  Stack(const T& e): vecotr<T>(1,e) {}

  void push(const T& e) {
    push_back(e);
  }

  T pop() {
    if (!empty()) {
      T e=(*this)[size()-1];
      pop_back();
      return e;
    }
    else
      return T(); //return a default element
  }
};
```

However, if we do so, then anything applicable to a vector is also applicable to a stack:

```
int main() {
  Stack s;

  . . .

  //we can access any element is s
};
```

Therefore, it is better to implement a stack in terms of a vector. The relation Stack HAS A (is implemented in terms of) vector is better than the relation Stack IS A vector in this case:

```
template<class T>
class Stack {
  vector<T> v;

 public:
  Stack() {}
  Stack(const T& e): v(1,e) {}

  void push(const T& e) {
    v.push_back(e);
  }
  T pop() {
    if (!v.empty()) {
      T e=v[v.size()-1];
      v.pop_back();
      return e;
    }
    else
      return T(); //return a default element
  }
};
```

17.7 Some concepts revisited

17.7.1 Construction

Construction is done from Base to Derived. For instance, when declaring a Student object, the Person is constructed first, then the Student.

17.7.2 Destruction

Destruction is done from Derived to Base. For instance, when destructing a Student object, the Student is destructed first, then the Person.

17.7.3 Default copy constructor

The default copy constructor performs a memberwise copy using the corresponding copy constructor for each member, at every level in the hierarchy. For instance, since we have not declared a copy constructor for Student, the default copy constructor will copy the id, and then perform a copy at the base level. Since we have a copy constructor for Person, this copy constructor will be used (it will copy the name and increment the counter). Note that if we supply our own copy constructor for Student, we must make sure to copy the base.

```
class Student: public Person {
 public:
  Student(const Student& s) {
    id=s.id;
    name=s.name; //error, name is private in Person
  }

  . . .
};

class Student: public Person {
 public:
  Student(const Student& s): Person(s) { //ok
    id=s.id;
  }

  . . .
};
```

17.7.4 Default assignment operator

Same description as above, simply replace copy constructor with assignment operator. For instance, since we have not declared an assignment operator for Student, the default assignment operator will copy the id, and then perform a copy at the base level. Since we have not declared an assignment operator for Person, the default assignment operator for Person will be used (it will copy the name). Again, if we supply our own assignment operator for Student, we must make sure to copy the base.

```
class Student: public Person {
 public:
  Student& operator=(const Student& s) {
    if (this==&s)
      return *this;
    id=s.id;
    name=s.name; //error, name is private is Person
    return *this;
  }

  . . .

};

class Student: public Person {
 public:
  Student& operator=(const Student& s) {
    if (this==&s)
      return *this;
    id=s.id;
    (Person)(*this)=s; //assignment operator for Person
    return *this;
  }

  . . .

};
```

Note: The assignment operator is not inherited, e.g. if Person declares an assignment operator and Student publicly inherits from Person, then Student would still have the default assignment operator (unless declared in Student).