Fundamentals Of Structured Programming

Lubia Vinhas

March 16, 2016

Preface

This booklet contains the notes for the course CAP-390 Fundamentals of Structured Programming. It is mainly based on the two books by Bjarne Stroustroup [2] [3]. I

Contents

Chapter 1

Introduction

This course is about good programming, using the C++ programming language, for people who want to become professionals (i.e. people who can produce systems that others will use), who are assumed to be bright and willing to work hard. The students will learn fundamental programming concepts, some key useful techniques that exist in several structured languages and the basics of modern C++. So that after the course they will be able to write small coloquial C++ programs, read and adapt much larger programs, learn the basics of many other languages by themselves and proceed with advanced C++ courses.

Our civilization runs on software. Most engineering activities involve software, but most programs do not run on things that look like a PC with a screen, a keyboard, a box under the table, but in aircrafts, ships, communication, phones, energy systems... and there is a lot more to computing than games, word processing, browsing, and spreadsheets.

What is programming? Literally, is the action or process of writing computer programs. To me, is *problem solving using algorithms*. A few terms related to programming:

- **Programming paradigm**: a pattern that serves as a school of thoughts for programming of computers
- **Programming technique**: related to an algorithmic idea for solving a particular class of problems. For example, "divide and conquer" or "program development by stepwise refinement"
- **Programming style**: the way we express ourselves in a computer program. Relates to elegance (or lack of elegance)
- **Programming culture**: The totality of programming behavior, which often is tightly related to a family of programming languages

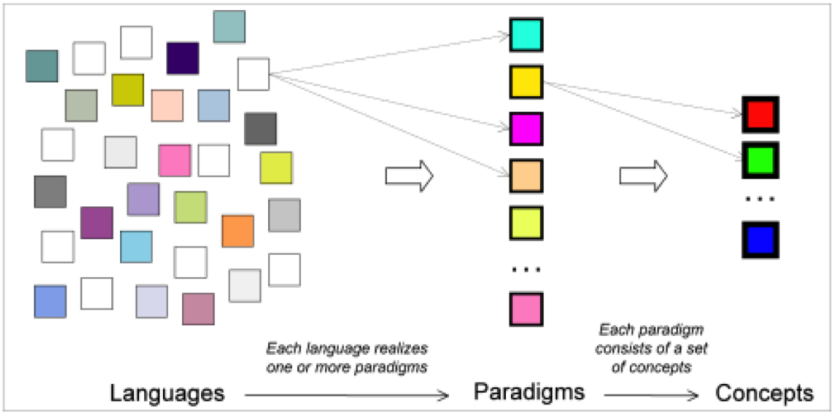


Figure 1.1: Languages, paradigms and concepts [4].

1.1 Programming paradigms

Solving a programming problem requires choosing the right concepts. All but the smallest toy problems require different sets of concepts for different parts of the program.

A programming paradigm, or programming model, is an approach to programming a computer based on a mathematical theory or a coherent set of principles. It is a way of conceptualizing what it means to perform computation and how tasks to be carried out on the computer should be structured and organized.

Programming languages realize programming paradigms. There are many fewer programming paradigms that programming languages. Examples of programming paradigms: imperative, funcional, logical, object-oriented.

Most popular languages are imperative and use structured programming techniques. Structured programming techniques involve giving the code you write structures, these often involve writing code in blocks such as **sequence** (code executed line by line), **selection** (branching statements such as *if..then..else*, or *case*) and **repetition** (iterative statements such as *for*, *while*, *repeat*, *loop*).

1.2 Imperative paradigm

Is based on the ideas of a Von Neummann architecture. A command has a measurable effect on the program and the order of commands is important. ***First do this and next do that***. Its main characteristics are incremental change of the program state (variables) as a function of time; execution of commands in an order governed by control structures; and the use of procedures,

abstractions of one or more actions, which can be called as a single command. Languages representatives of Imperative paradigm: *Fortran*, *Algol*, *Basic*, *C*, *Pascal*.

Program 1 Example of Pascal code

```
Program Example1;
Var
Num1, Num2, Sum: Integer;
Begin
Write ('Input number 1:');
Readln(Num1);
Write ('Input number 2:');
Readln(Num2);
Sum := Num1 + Num2;
Writeln(Sum);
Readln;
End
```

1.3 Functional paradigm

Based on mathematics and theory of functions. The values produced are non-mutable and time plays a minor role compared to imperative program. All computations are done by applying functions with no side effects.Functions are firsts class citizens. ***Evaluate an expression and use the resulting value for something***. Representatives: *Haskell*, *Clojure*. Example in Haskell:

Program 2 Example of Haskel code

```
-- Compute the sum of integers from 1 to n.
sumtorial :: Integer -> Integer
sumtorial 0 = 0
sumtorial n = n + sumtorial(n - 1)
main = print(sumtorial 10)
```

1.4 Logical paradigm

The logic paradigm fits well when applied in problem domains that deal with the extraction of knowledge from basic facts and relations. Is based on axioms, inference rules, and queries. Program execution becomes a systematic search in a set of facts, making use of a set of inference rules. ***Answer a question via* search *for a solution***.

Program 3 Example of PROLOG code

```
mortal(X) :- human(X)
human(socrates)
human(pitagoras)
?- mortal(socrates)
yes
?- mortal(chico)
no
```

1.5 Object-oriented paradigm

Data as well as operations are encapsulated in objects. Information hiding is used to protect internal properties of an object. Objects interact by means of message passing. In most object-oriented languages objects are grouped in classes and classes are organized in inheritance hierarchies. ***Send messages between objects to simulate the temporal evolution of a set of real world phenomena***. Representatives: *C++*, *Java*.

Program 4 Example of C++ code

```
class Employee
{
public:
string name;
int age;
double salary;
};

Employee e1;
e1.name = "socrates";
e1.age = 26;
e1.salary = 1000;

Employee e2;
e2.name = "chico";
e2.age = 30;
e2.salary = 5000;

double cost = e1.salary+e2.salary;
```

There are other paradigms, such as Visual paradigm, Constraint based paradigm, Aspect-oriented paradigm and Event-oriented paradigm.

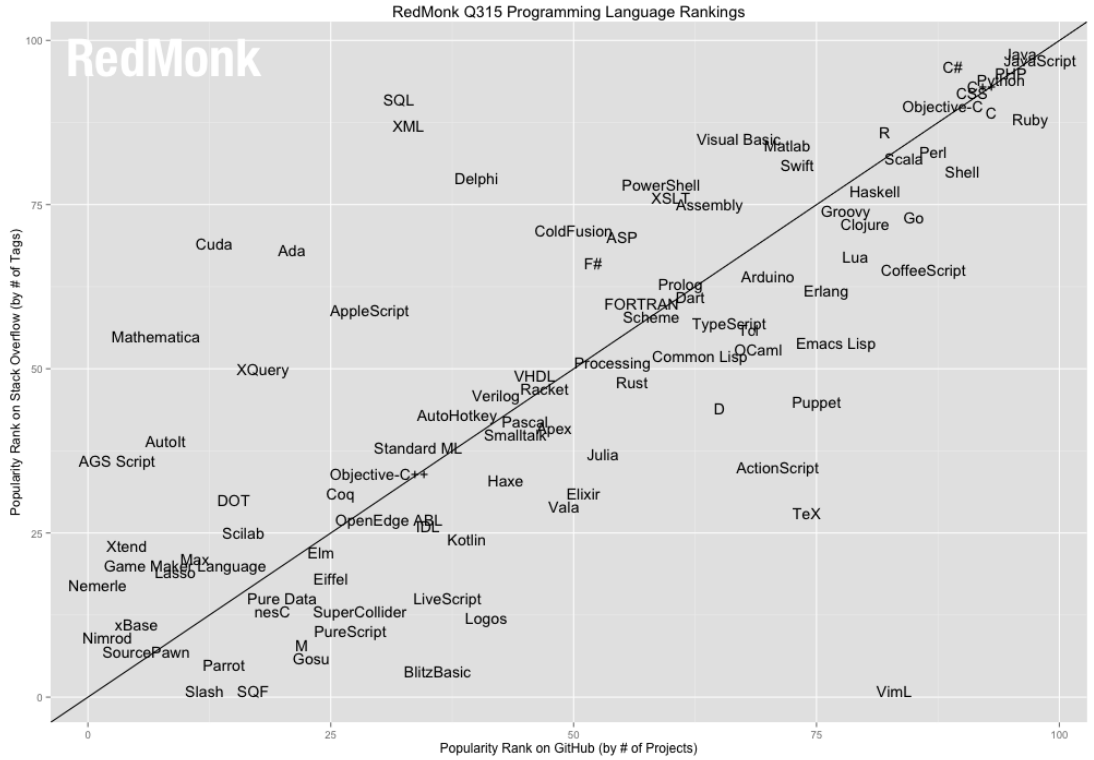


Figure 1.2: Ranking of languages. Source: http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/

1.6 Programming languages

What we have to learn to study a programming language? Syntax, semantics and how to best use the languages features to implement the programming paradigm more adequate to solve your problem. How many languages are out there? Which languages should I know?

Any ranking is influenced by communities of the development, investments from third parties and ubiquitousness of projects and statistics. But this course is not about advocating the use of this or that language. I first learnt how to program with algorithms, than I learnt PASCAL. I work with C++, SQL and I can do a few things in PHP. I believe I can learn other languages if I have to. I am not a radical champion for C++ or any other language. I believe that there is no silver bullet [1]. **This course will teach C++**.

Chapter 2

Introduction to C++

Why C++? You can not learn to program without a programming language, the purpose of a programming language is to allow you to express your ideas in code. C++ is the language that most directly allows you to express ideas from the largest number of application areas. Programming concepts that you learn using C++ can be used fairly directly in other languages, including C, Java, C sharp and (less directly) Fortran.

Bjarne Stroustrup is the creator of the C++ language:(*http://www.stroustrup.com/*) you can have an overview of C++ history in
http://www.cplusplus.com/info/history/.

Bjarne has written several books about programming and C++ such as the ones shown bellow. *Programming - Principles and Practices Using C++* is the one used in this course.

Figure 2.1: Bjarne Stroustroup, creator of C++ language.

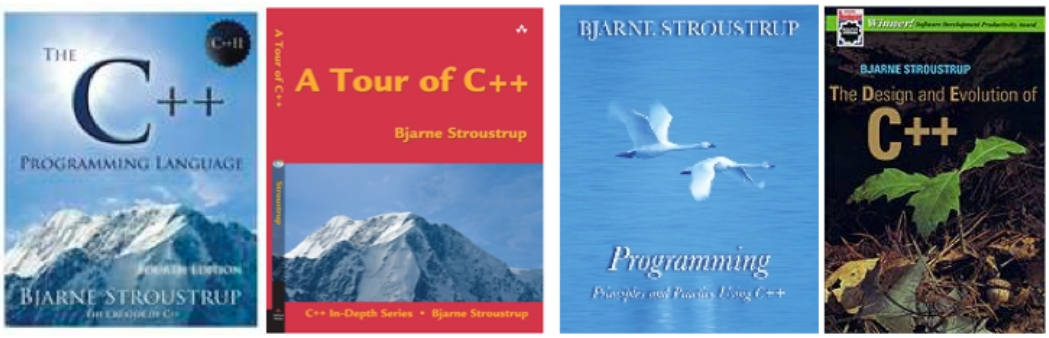


Figure 2.2: C++ books.

2.1 Standardization

The C++ Language **is an open ISO-standardized language**. For a time, C++ had no official standard and was maintained by a de-facto standard, however since 1998, C++ is standardized by a committee of the ISO. **Is a compiled language**, C++ compiles directly to a machine's native code, allowing it to be one of the fastest languages in the world, if optimized. **Is a strongly-typed unsafe language**, C++ is a language that expects the programmer to know what he or she is doing, but allows for incredible amounts of control as a result.

C++ is standardized. The C++ standard was finalized and adopted by ISO (International Organization for Standardization) as well as several national standards organizations. The ISO standard was finalized and adopted by unanimous vote in November 1997. The latest (and current) standard version was ratified and published by ISO in December 2014 as ISO/IEC 14882:2014 (informally known as C++14), as can be seen in Figure 2.3.

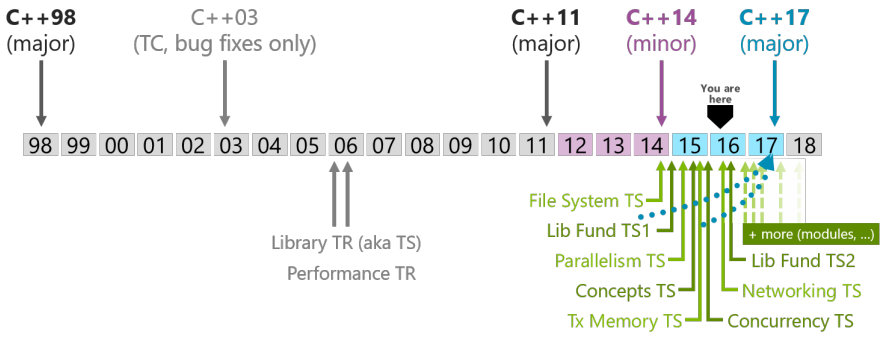


Figure 2.3: C++ standards timeline. Source: https://isocpp.org.

As one of the most frequently used languages in the world and as an open language, C++ has a wide range of compilers that run on many different platforms that support it. Sometimes the latest standard is not 100% supported by all compilers. Check http://en.cppreference.com/w/cpp/compiler

to see features, versions and several compilers compliance.

2.2 Programming environment

C++ is a compiled language. That means you will need some tools to work with C++:

- **Editor**: to write your code
- **Compiler**: translate the source code to machine code to be executed
- **Interpreter**: reads a little source code, translates it to machine code, and executes it, than reads a little more, etc.
- **Debugger**: helps you step through code, shows you variables and flow of execution
- **Linker**: connects code from libraries with your code to make one executable

There are **Integrated Development Environments** (IDE) that provides editors, compilers and linker in as a single package. Examples of IDEs for C++:

1. **Windows**:
   - Microsoft Visual C++ → *http://www.visualstudio.com/*
   - MingW → *http://www.mingw.org/*
2. **Linux**:
   - Eclipse → *http://www.eclipse.org/cdt/*
   - QtCreator → *http://www.qt.io/ide/*
3. **Web** based:
   - C++ Shell → *http://cpp.sh/*
   - CodeChef → *https://www.codechef.com/ide*

2.3 Basic facilities

The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones. The most important thing to do is to focus on concepts and not get lost in language-technical details. This section introduces some C++ program features necessary to the most basic programs.

2.3.1 Types

In **typed** languages, such as C++, every operation defines types of data to which an operation is applicable, with the implication that it is not applicable to other types. For example, "this text between the quotes" is a **string**, and 10 is a **number**. In most languages the division of a number by a string (or vice-versa) has no meaning and the compiler will reject it. This is static type checking. In C++ you can have:

- **built-in** types: `bool`, `char`, `float`, `int` (`short` and `long`), etc.
- **Standard Library** types: `string`, `complex`, input/output streams, etc.
- **user-defined** types (more about this later).

For each type an operand have a particular semantics. The type of a variable determines which operations are valid and what their meanings are for that type. For example:

Strings (STD)		**Integers** (built-in)	
`cin >>`	reads a word	`cin >>`	reads a number
`cout <<`	writes a word	`cout <<`	writes a word
+	concatenates	+	adds
+= s	adds the string s at end	+= n	increments the int by n
++	is an error	++	is n increments by 1
−	is an error	−	subtracts

2.3.2 Objects, declaration and initialization

In the computer memory, everything is just bits; type is what gives meaning to the bits. Some **types** and **literals** in C++ can be seen in Table 2.1. An **object** is some memory that can hold

Table 2.1: C++ types and literals

Type	Literal
`bool`	`true`, `false`
`int`	1 2
`float`	10.2 11.3
`char`	`'c'`
`string`	`"abcd"`

a value of a given type. A **variable** is a named object. A **declaration** names an object.

2.3.3 C++ Standard Library

The C++ Standard Library is a collection types and functions, which are written in the core language and part of the C++ ISO Standard itself. It provides a ready-made set of common and

Table 2.2: Types and memory space

Declaration	Memory	
`int i;`	i:	␣␣␣␣␣␣␣␣
`int a = 7;`	a:	␣␣␣␣␣␣␣7
`int b = a;`	b:	␣␣␣␣␣␣␣7
`char c = 'a';`	c:	'c'
`double x = 1.2;`	x:	␣␣␣␣␣␣␣␣␣␣␣␣␣1.2
`string s2 = "lubia";`	s2:	5 | lubia

highly used features for C++. Program 5 shows a very simple using the Standard Library.

Program 5 STD input and output

```
#include <iostream>
#include <string>
int main() // read first and second name
{
  std::cout << "please enter your first and second names\n";
  std::string first, second;
  std::cin >> first >> second; // read two strings
  std::string name = first + ' ' + second;  // concatenate
  std::cout << "Hello, "<< name << '\n';
  return 0;
}

```

- the macro `#include` give access external packages and libraries
- `<iostream>`: is the STL package for inputing and outputing
- `<string>`: is the STL package to handle strings of characters
- comments are identified "`//`" (single line) or "`/* ... */`" (multiple lines).
- the `main` scope defines the piece of code that will be the final program.
- a C++ statement end with "`;`"
- `::std` : indicates that a name is defined in the standard library namespace
- operator `cin` >> : reads from streams
- operator `cout` << : puts objects in streams.
- return a value indicating success (usually 0).

Only lines 6 to 9 directly do anything. That's normal. Much of the code in our programs come from someone else (libraries). Would you rather write 1,000,000 lines of machine code? Code that exclusively uses C++'s standard library will run on many platforms with few to no changes and is upwards compatible with C.

2.3.4 Separate compilation

C++ suports separate compilation, that can be used to organize a program into a set of semi-independent fragments. An executable (final application) is the result of linking and compiling some piece of code that has one `main` block of statements. Pieces of code that do not have a `main` (libraries), when compiled produce object code to be linked with other code to produce a final application. Figure 2.4 shows how is the building process happens to generate an executable program.

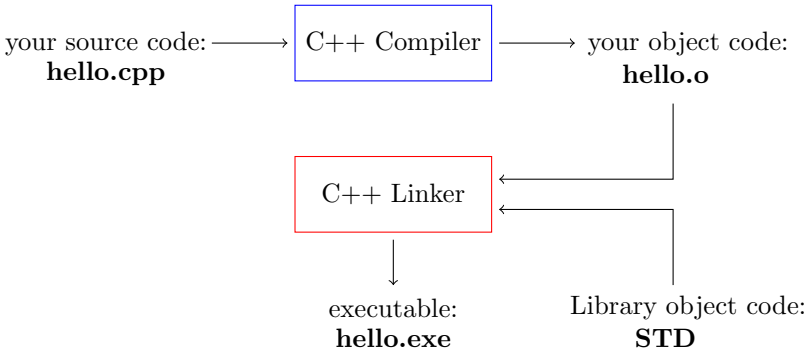


Figure 2.4: Example of the building process

The **interface** is placed in a header file that is included by the users, whereas its **implementation** is defined in another file. To use a library you have to have access to its interface files at compilation time and to its object code at linking time. Open source libraries also make its implementation files available.

2.4 Computation

Computation is what we do to manipulate objects. To program, we have to think *what is computable* and *how best to compute it.* To do this we think about abstractions, algorithms, heuristics, data structures. We use language to construct these ideas in terms of sequential order of execution, expressions and statements, selection, interation, functions and data structures (for example vectors).

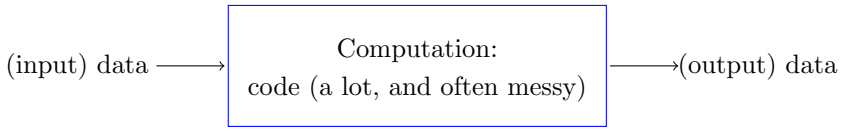


Figure 2.5: A typical program

- **Input**: data from keyboard, files, other input devices, other programs, other parts of a program
- **Computation**: what our program will do with the input to produce the output
- **Output**: data sent to screen, files, other output devices, other programs, other parts of a program

Computations should be expressed, correctly, simply and efficiently. To achieve that, different strategies can be adopted, e.g., **divide and conquer** (breaking up big computations into many little ones), or **abstraction** (provide a higher-level concept that hides details not relevant to the problem solution). **Organization of data** is often the key to good code: input/output formats; protocols; data structures.

Expressions are the most basic building blocks of a program. An expression computes a value from a number of operands. An expression produces a single value. Table 2.3 shows some examples of expressions.

Table 2.3: Expressions

Expression	**Example**
Literals	`10`, `'a'`, `3.14`, `"Norah"`
Names of variables	`int lenght;`
Combinations	`perimeter = (length+width)*2;`
Constant expressions	constexpr double pi=3.141516;

Most **Operators** are conventional and you can look up details if and when you find a need. But a list of the most common operators include:

- **mathematical operators**: `+`, `*`, `%` (reminder) ... :
- **logical operators**: `==` (equal), **`!=`** (not equal), `&&` (logical AND), || (logical OR)
- **increment/decrement operators**: `++lval`, `--lval`, ... (lval is short for value that can appear on the left-hand side os an assignment)

Statements are language features used to produce several values, or to do something many times, or choose among alternatives:

- **selection**: `if`-statements; `switch`-statements.
- **iteration**: `while`-statements; `for`-statements

Program 6 shows some computation using mathematical operators, such as +, ∗, / and a mathematical function (`sqrt`). Program 7 also shows some computation using iteration statements.

Program 6 A very simple computation

```
// do a bit of very simple arithmetic:
int main()
{
  cout << "please enter a floating-point number: ";
  double n;              // floating-point variable
  cin >> n;
  cout << "n == " << n
  << "\nn+1 == " << n+1          // '\n' means a newline
  << "\nthree times n == " << 3*n
  << "\ntwice n == " << n+n
  << "\nn squared == " << n*n
  << "\nhalf of n == " << n/2
  << "\nsquare root of n == " << sqrt(n) // library function
  << endl;                       // another name for newline
}
```

Program 7 Other computing features

```
if (a<b)     //selection
  max = b;
else
  max = a

for (short i=0; i<10; ++i)  // repetition
  cout << i << ' => ' << square(i) << '\n';

int i = 0;
while (i<100)
{
  cout << i << ' => ' << square(i) << '\n';
  ++i ; // increment i;
}
```

2.4.1 Functions

What is the name `square` in Program 7? It is a call to a function that was defined somewhere. We define a function when we want to separate a computation because it is logically separate, it makes the program text clearer (by naming the computation), is useful in more than one place in our program. It eases testing, distribution of labor, and maintenance of our code.

A function can not be called unless it has been previously declared. A function declaration gives the name of the function, the type of the value returned (if any) by the function, and the number and types of arguments that must be supplied in a call of the function(e.g. Program 8). Program 9 shows an example of using a STL vector.

Program 8 Functions

```
// max.h: function declaration
int max(int, int);

// max.cpp  : function implementation
int max(int a, int b)  // this function takes 2 parameters
{
  if (a<b)
    return b;
  else
    return a;
}

// main.cpp
#include "max.h"

int x = max(7, 9);     // x becomes 9
int y = max(19, -27); // y becomes 19
int z = max(20, 20);  // z becomes 20
```

2.4.2 Data structures

To do just about anything of interest, we need a collection of data to work on. A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

Languages provide mechanisms to implement data structures, or libraries that provide common data structures ready to use. The C++ Standard Library includes most of the Standard

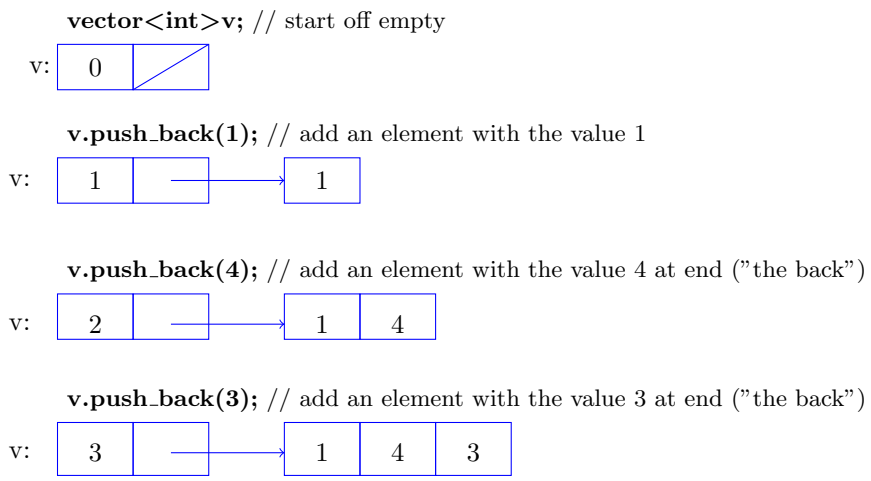


Figure 2.6: The Vector construction

Template Library (STL) [1]. The STL is a generic library, meaning that its components are heavily parameterized by a type.

Vector is the most useful STL data structure, or data container. A `vector<T>` holds a sequence of values of type `T` (e.g. a vector named `v` contains 3 elements: $1, 4, 3$). Figure 2.6 shows the structure of a vector and how it is built using `pushback` while Program 9 exemplify how to use it.

Program 9 An example using a STL vector

```
// read some temperatures into a vector:
int main()
{
  // declare a vector of type double to store temperatures
  vector<double> temps;
  double temp;     // a variable for a single temperature value

  // cin reads a value and stores it in temp
  while (cin>>temp)
    temps.push_back(temp);
  //... do something ...
}
// cin>>temp  will return true until we reach the end of file
// or encounter something that isn't a double
```

[1] https://www.sgi.com/tech/stl/

2.5 Memory, addresses and pointers

A computer's memory is a sequence of bytes. When the computer encounter a declaration such as `int var=17` it will set aside an `int`-size piece of memory for var somewhere and put the `17` into memory, this location has an *address*. Some languages, such as C++, allows you to store and manipulate addresses. An object that store an address is called a pointer. Program 10 shows the C++ syntax to manipulate pointers and addresses.

Program 10 Pointer and address

```
1  int x;   // a variable called x, that can hold an integer
   x = 17;  // the value 17 is stored in variable x
3
   int* pi; // a variable called px that can hold an address of an
5  // integer variable, os simply a pointer to int

7  pi = &x; // the address of x is stored in variable pi

9  // prints the content of variable x
   cout << "x=" << x << std::endl;
11
   // prints the address of variable x
13 cout << "&x=" << &x << std::endl;

15 // prints the content of variable pi
   cout << "pi=" << pi << std::endl;
17
   // prints the content of memory pointed by pi
19 cout << "*pi=" << *pi << std::endl;
```

Important things about pointers:

- a pointer is a type (such as `int`), that provides operators suitable for addresses (whereas `int` provides operators suitable to integers)
- operator `&` retrieves the address of a variable and a pointer can hold addresses
- operator `*` can only be applied to pointers and retrieves the contents of the memory it points to

A pointer is specific to the type it points to. It is not possible to assign the address of an `int` to a `pointer to char`, because different types occupy different sizes in memory. The size of a type is not guaranteed to be the same on every implementation of C++. Program 11 shows how to check the size of types.

Program 11 Size of types

```
cout << "the size of a char is " << sizeof(char) << '\n';
cout << "the size of an int is " << sizeof(int) << '\n';
cout << "the size of a float is " << sizeof(double) << '\n';

cout << "the size of an int* is " << sizeof(int*) << '\n';
cout << "the size of an float* is " << sizeof(float*) << '\n';
```

2.5.1 Free store and pointers

When you start a C++ program, the computer set aside memory for your code (*code storage*), for the global variables (*static storage*) and to be used when you call functions and their variables (*stack storage).* The rest of the computer's memory is potentially available for other uses: it is the *free* memory. C++ makes this free storage (or *heap*) available through the operator `new`.

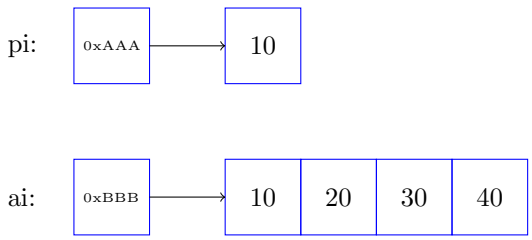
The new operator can allocate individual elements or sequences (arrays) of elements. In addition of using the dereference operator (`*`) on a pointer we can also use the substrcipt operator (`[]`). It treats memory as a sequence of objects (of the type specified) with the pointer pointing to the first one. The memory allocated can be (should be!) returned to the free store using the operator delete. (see Program 12).

Program 12 Allocation on free memory

```
int* pi = new int;       // allocate one int
int* ai = new int[4];    // allocate 4 ints (an array of 4 ints)

pi = 10;
ai[0] = 10;
ai[1] = 20;
ai[2] = 30
ai[3] = 40;

delete pi;       // if you allocate you have to deallocate
delete [] ai;
```



2.5.2 Pointers and functions

Whenever a function needs to return a value (after some computation) it is necessary to declare that the function return a value (declaration) and use the `return` statement (implementation). Function can also receives arguments.

Function `fbyvalue` in Program 13 uses the simplest way of passing an argument called called **pass-by-value**. It means that the function receives a copy of the argument, so the original variable, passed as an argument to a function is not changed by the function. Passing arguments by value have some drawbacks, mainly the cost of copying arguments. In the example is just an int, but if the argument is a very large data structure (for example, an image) then the function can be costly.

In function `fbyref` in Program 13 it was used the way **pass-by-argument**. In this case, the function receives the address of the variable passed as argument. That means that the function can in fact change the value of the variable. Somethimes you need the best of the two ways: preventing unecessary copies and not allowing the function to accedentally change the argument value. In this case you should make the argument address constant as in `fbyconstref`.

Program 13 Function arguments and returnig

```
void NoRetNoArgFunc()
{
// do something and go
}

int fbyvalue(int x)    // pass-by-value
{
   x = x+1;
   return x;
}

int fbyref(int& x)    // pass-by-reference
{
  x = x+1;
  return x;
}

int fbyconstref(const int& x)    // pass-by-const-reference
{
  return x;
}
```

Rule of thumb to decide between pass-by-value or pass-by-reference:

1. use pass-by-value to pass very small objects

2. use pass-by-`const`-reference to pass large objects that you don't neeed to modify

3. return a result rather than modifying an object through a reference argument

4. use pass-by-reference only when you have to.