deep se
dependable evolvable pervasive software engineering group

C++ Introduction

Danilo Ardagna

Politecnico di Milano
danilo.ardagna@polimi.it

POLITECNICO DI MILANO

Content

- Why C++ and Object Oriented Programming (OOP)
- Hello world
- Namespaces
- Built-in types
- Control structures
- Arrays and Structs
- Declarations and Definitions
- Header files
- Classes
- Vectors
- String (readings)

Why C++ ?

- 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, especially engineering
- Design goals:
  - Start from C, add start-of-the-art features
  - Achieve abstraction **without loosing performance**
- C++ is the most widely used language in engineering areas
  - http://www.stroustrup.com/applications.html

C++ father

- Bjarne Stroustrup
  - AT&T Bell labs
  - Texas A&M University
  - Making abstraction techniques affordable and manageable for mainstream projects
  - Pioneered the use of object-oriented and generic programming techniques in application areas where **efficiency** is a premium

Why C++ ?

- C++ is precisely and comprehensively defined by an ISO standard
  - And that standard is almost universally accepted
  - The most recent standard in ISO C++ v 17, Dec 2017
  - Working on C++ 20, at final approval step released by the end of this year
- C++ is available on almost all kinds of computers
- Programming concepts that you learn using C++ can be used fairly directly in other languages
  - Including C, Java, C#, Python
- **Faster** than other OO languages!!!

Why C++ ?

- C++ is precisely and comprehensively defined by an ISO standard
  - And that standard is almost universally accepted
  - The most recent standard in ISO C++ v 17, Dec 2017
  - Working on C++ 20, still in draft version
- C++ is available on almost all kinds of computers
- Programmi
  used fairly
  - Including
- **Faster** tha

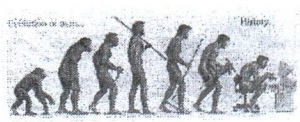
- In banking and trading systems latency is very important
- In scientific applications speed is very important
- In tiny (embedded applications where resources are limited) or very large systems performance (speed and energy!) are **extremely** important !!!

C++ in the context of Programming Paradigms

- Low-level vs. high-level programming languages
  - Machine vs. human
- Styles of computer programming
  - Procedural programming
  - Object-oriented programming
  - Functional programming
  - …

Low-level vs. High-level Programming Languages

- Low-level:
  - Assembly
  - Machine code
- High-level: (abstraction from the computer details)
  - Basic, C, Java, Pascal, **C++**, Perl, Python, …



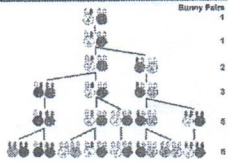
The history of computer programming is a steady **move away from machine-oriented** views of programming towards **concepts** and **metaphors** that more closely reflect the way in which we **ourselves understand the world**

Styles of Computer Programming

- Procedural programming
  - Imperative: procedures, routines, subroutines or functions
- Object-oriented programming
  - **Objects** and **Classes**
- Functional programming
  - Mathematical functions
  - E.g. Lisp, Erlang, Haskell, **Scala**, **Python**, …
- …

Examples (1/4)

- Fibonacci numbers
  - $F_n = F_{n-1} + F_{n-2}$, $n>=2$
    $F_0 = 0$, $F_1 = 1$



- How to program?
  - The following examples are adapted from Wikipedia

Examples (2/4)

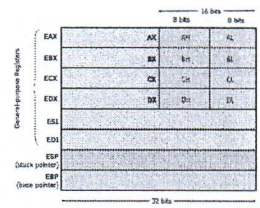
- Procedural: (C)

```
int fib(int n)
{
  int first = 0, second = 1;
  for (int i=0; i<n; ++i)
  {
    int sum = first + second;
    first = second;
    second = sum;
  }
  return first;
}
```


Examples (3/4)

- Assembly: (in x86 using MASM syntax)

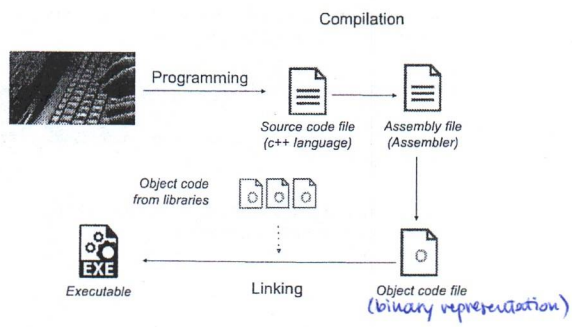
```
mov edx, [esp+8]
cmp edx, 0
ja @f
mov eax, 0
ret
@@: cmp edx, 2
ja @f
mov eax, 1
ret
@@: push ebx
mov ebx, 1
mov ecx, 1
@@: lea eax, [ebx+ecx]
cmp edx, 3
jbe @f
mov ebx, ecx
mov ecx, eax
dec edx
jmp @b
@@: pop ebx
ret
```



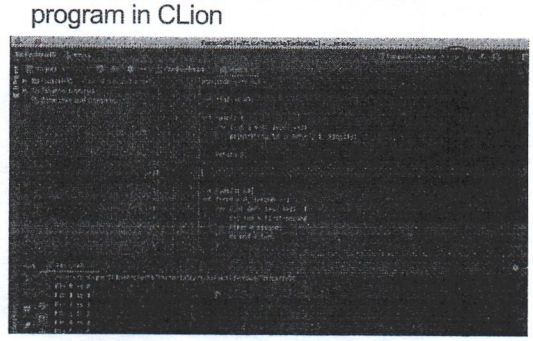
Examples (4/4)

- Machine code: (a function in 32-bit x86)
  - 8B542408 83FA0077 06B80000 0000C383 FA027706 B8010000 00C353BB 01000000 B9010000 008D0419 83FA0376 078BD98B C84AEBF1 5BC3
  - 10001011010101000010010000001000100000111111101000000000111011110000011010111000000000000000000000000000000000000110000111000001111111010000000100111011100000111011000 00000001000000000000000 00000000 1100 001101010011101110110000000100000000000000000000000010111001000000010000000000000000000000010001101000001000001100110000011111111010000000110111011000000111100010111101100110001011110010000100101011101000111100010101101111100 0011

Compilation and Linking



Compiling and Linking our first (C) program in CLion



Procedural Programming (Procedure oriented)

- Top down approach
- Big program is split into small pieces
  - Procedures, also known as functions or methods simply contains a series of computational (Algorithmic) steps to be carried out
- Procedural programming specify the syntax and procedure to write a program
- **Functions are more important than data**
- Input-arguments, output-return values
- Ex. are C, Algol etc.

Object Oriented Programming

- Bottom up approach
- Programs are built from classes and objects
  - **"Class"** refers to a blueprint. It defines the **data** members and the **operations** (mechanisms) the objects support
    - It is a collection of similar objects
    - You **put together data with functions** to work on that data
  - **"Object"** is an instance (properties) of a class. Each object has a class which defines its data and behavior
- Ex. are C++, Java, Python, etc.

Concept of Class and Object

- **"Class"**

Dog
name
run
stop
eat

which data we have to store for each dog

what are the operations that a dog can do

- **"Object"** = instances

Dido

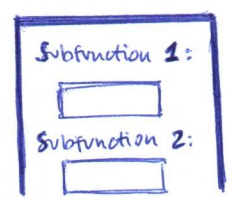
Giotto

Penny

Top down approach

- A complex program divides into smaller pieces, makes efficient and easy to understand a program
- Begins from the top level
- Emphasize the planning and complete understanding of a program
- **No coding can begins until a sufficient level of module details has been reached**

Problem:



we have to visualize the whole problem to begin coding

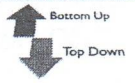
Advantages of the Top-Down Design Method

- It is **easier to comprehend** the solution of a smaller and less complicated problem than to grasp the solution of a large and complex problem
- It is **easier to test segments** of solutions, rather than the entire solution at once. This method allows one to test the solution of each sub-problem separately until the entire solution has been tested
- It is often possible to **simplify the logical steps** of each sub-problem, so that when taken as a whole, the **entire solution** has **less complex logic** and hence **easier to develop**
- A simplified solution takes **less time** to develop and will be **more readable**
- The program will be **easier to maintain**

OK for algorithms implementation but not to implement large systems

Object Oriented Programming

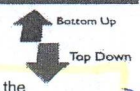
OOP: Bottom up approach



- Reverse top down approach:
  - **Lower level tasks are first carried out and are then integrated to provide the solution of a single program**
- Lower level structures of the program are evolved first, then higher-level structures are created
- Promotes code **reuse**
- Favors software **modularization**

It starts from the basic concepts, not from the whole problem.
We want every piece of code to be independent from the rest
→ MODULARIZATION

OOP: Bottom up approach

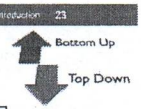


- Identify the **main abstractions** that characterize the application domain and represents them in your project as **classes**
  - Eg. CAD: geometric shapes, triangles, rectangles, lines, points, colors ...
  - Eg. e-mail: message, person, address book, protocol ...
  - Eg. ERP application: person, employee, manager, consultant, project, salary, reimbursement ...
- Assemble the various components by **identifying** the **mechanisms** that allow different objects to **work together** to implement application features
- In this way, applications are easier to understand and manipulate

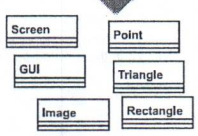
We don't know the "problem"/"task", we start to define useful concepts

Once we have concepts we can focus on applications. Applications can be separated, parallelized and they result independent among each other.

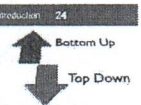
OOP: Bottom up approach



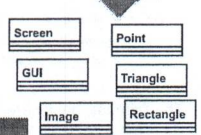
An OO program:
- Code
  - Set of **Classes** definition
  - No a single huge "main()"

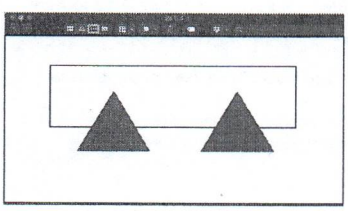


OOP: Bottom up approach

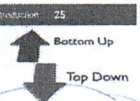


An OO program:
- Code
  - Set of **Classes** definition
  - No a single huge "main()"

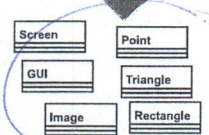




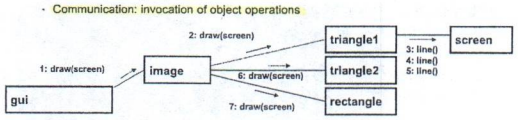
OOP: Bottom up approach



An OO program instance:
- Code
  - Set of **Classes** definition
  - No a single huge "main()"



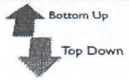
- Run-time configuration
  - Set of cooperating **objects**
  - Communication: invocation of object operations



these are the objects (concepts)

they have to collaborate:
"image" will ask "draw" to "triangle". This is a method invocation.
"triangle" will ask "line" 3 times to "screen", ... etc.

OOP: Bottom up approach



- What is difficult if you come from a procedural top down world:
  - **change the way of thinking**
  - designing reusable objects is a complex activity
  - the main activity of an OO developer is to **reuse objects** (libraries) made by others

How is C++ structured?

- Three parts in C++
  - Low-level language: largely inherited from C
    - Data types, flow of control, functions, arrays, pointers, ...
  - Advanced language features: to define our own data types
    - Class, inheritance, polymorphism, template, ...
  - Standard library: some useful data structures and algorithms
    - Containers, iterators, ...

C++ is difficult: Why?!

- The worst parts of C++ are those that were inherited from C:
  - Implicit conversions, raw arrays, raw pointers, and the interactions between those features and some new C++ features

272 pages

1399 pages

C++ is difficult: Why?!

- The worst parts of C++ are those that were inherited from C:
  - Implicit conversions, raw arrays, raw pointers, and the interactions between those features and some new C++ features
- The (almost) backwards compatibility is probably the reason C++ is successfull
- At the time C++ was developed, there were two options for the designers:
  - Create a nice, consistent, ideal language that no-one will use
  - Build on an existing languages, and very very carefully add new features step by step

C++ is difficult: Why?!

- The worst parts of C++ are those that were inherited from C:
  - Implicit conversions, raw arrays, raw pointers, and the interactions between those features and some new C++ features
- The (almost) backwards compatibility is probably the reason C++ is successfull
- At the time C++
  - Create a nice,
  - Build on an exi
  features step b

Final result:

- We have power, performance, and control
- We need to know what's going on under the hood!

A first program – just the guts…

```
// ...

int main()                  // main() is where a C++ program starts
{
 cout << "Hello, world!\n";// output the 13 characters Hello, world!
                            // followed by a new line
 return 0;                  // return a value indicating success
}

// quotes delimit a string literal
// NOTE: "smart" quotes “ ” will cause compiler problems.
// so make sure your quotes are of the style " "
// \n is a notation for a new line
```


A first program – complete

```
// a first program:

#include <iostream>        // get the library facilities needed for now
using namespace std;       // This allows to use cout instead
                           // of std::cout
int main()                 // main() is where a C++ program starts
{
 cout << "Hello, world!\n";// output the 13 characters Hello, world!
                           // followed by a new line
 return 0;                 // return a value indicating success
}

// note the semicolons; they terminate statements
// braces { ... } group statements into a block
// main( ) is a function that takes no arguments ( )
// and returns an int (integer value) to indicate success or failure
```

DEMO

Hello, world!

- “Hello, world!” is a very important program
  - Its purpose is to help you get used to your tools
    - Compiler
    - Program development environment
    - Program execution environment
  - Type in the program carefully
    - After you get it to work, please make a few mistakes to see how the tools respond; for example
      - Forget the header
      - Forget to terminate the string
      - Misspell return (e.g., retrun)
      - Forget a semicolon
      - Forget { or }
      - ...

Hello, world!

- It's almost all “boiler plate”
  - Sections of code that have to be included in many places with little or no alteration
  - Only cout << "Hello, world!\n" directly does something “useful”
- *Boiler plate*, that is, notation, libraries, and other support is what makes our code simple, comprehensible, trustworthy, and efficient
  - Would you rather write 1,000,000 lines of machine code?
- That's normal
  - Most of our code, and most of the systems we use simply exist to make some other code elegant and/or efficient
- This implies that we should not just “get things done”; we should take great care that things are done elegantly, correctly, and in ways that ease the creation of more/other software

Style Matters!

Input and output

```
#include <iostream>
using namespace std;   //Simplify cin, cout and endl use
int main()
{
 cout << "Please enter your first name (followed "
 << "by 'enter'):"<<endl;
 string first_name;
 // read first name:
 cin >> first_name;
 cout << "Hello, " << first_name << endl;
}

// note how several values can be output by a single statement
// a statement that introduces a variable is called a declaration
// a variable holds a value of a specified type
// the final return 0; is optional in main()
// but you may need to include it to pacify your compiler
```

DEMO

C vs. C++

C Program

```
#include<stdio.h>


void main()
{
/*Not completely equivalent!!! */
char first_name[100];
printf ("Please enter
 your first name
 (followed by
 'enter'):\n");
scanf("%s", first_name);
printf("Hello, %s\n",
 first_name);
}
```

C++ Program

```
#include <iostream>
using namespace std;

int main()
{

string first_name;
cout << "Please enter your
 first name (followed " <<
 "by 'enter'):\n";
cin >> first_name;
cout << "Hello, " <<
 first_name << endl;
}
```

Strings are variable-length

Standard I/O objects

- C++ does not define any statements to do input or output
  - Includes an extensive **standard library** that provides I/O (and many other facilities)
- **iostream** library:
  - **istream** and **ostream**, representing input and output streams
  - A stream is a sequence of characters read/written to an I/O device
  - Characters are generated, or consumed, sequentially over time
- Four I/O objects are defined:
  - **cin:** handles input
  - **cout**: handles output
  - **cerr**: used for warning and error messages
  - **clog**: used for general information about the execution of the program
- The system associates each of these objects with the window in which the program is executed

Standard I/O objects

- **endl**
  - A **manipulator**
  - Writing endl has the effect of ending the current line and flushing the buffer associated with that device
- Flushing the buffer ensures that all the output the program has generated so far is actually written to the output stream, rather than sitting in memory waiting to be written

endl is better than "\n"

Input and type

- We read into a variable
  - Here, first_name
- A variable has a type
  - Here, string
- The type of a variable determines **what operations** we can do on it
  - Here, `cin>>first_name;` reads characters until a whitespace character is seen
    - Returns only after you hit enter
  - Whitespace: space, tab, newline, ...

String input

```
// read first and second name:
int main()
{
 cout << "please enter your first and second names\n";
 string first;
 string second;
 cin >> first >> second;              // read two strings
 string name = first + ' ' + second;// concatenate strings
                                      // separated by a space
 cout << "Hello, "<< name << '\n';
}
```

Integers

// read name and age:

```
int main()
{
 cout << "please enter your first name and age\n";
 string first_name;            // string variable
 int age;                      // integer variable
 cin >> first_name >> age;     // read
 cout << "Hello, " << first_name << " age " << age
 << '\n';
}
```


Integers and Strings

Strings	Integers and floating-point numbers
cin >> reads a word	cin >> reads a number
cout << writes	cout << writes
+ concatenates	+ adds
+= s adds the string s at end	+= n increments by the int n
++ is an error	++ increments by 1
- is an error	- subtracts
...	...

- The type of a variable determines:
  - The values (domain) for the variables of that type
  - Which operations are valid and what their meanings are for that type
- Same operators over different types: that's called "**overloading**" or "**operator overloading**"

Names

- A name in a C++ program
  - Starts with a letter, contains letters, digits, and underscores (only)
    - x, number_of_elements, Fourier_transform, z2
    - Not names:
      - 12x
      - timetomarket
      - main line
    - **Do not start** names with **underscores**: _foo
      - those are reserved for implementation and systems entities
  - Users **can't define names** that are taken as **keywords**
    - E.g.:
      - int
      - if
      - while
      - double
      - new

Names

- Choose meaningful names
  - Abbreviations and acronyms can confuse people
    - mtbf, TLA, myw, nbv
  - Short names can be meaningful
    - (only) when used conventionally:
      - x is a local variable
      - i is a loop index
  - Don't use overly long names
    - Ok:
      - partial_sum
        element_count
        staple_partition
    - Too long:
      - the_number_of_elements
        remaining_free_slots_in_the_symbol_table

Simple arithmetic

// do a bit of very simple arithmetic:

```
int main()
{
cout << "please enter a floating-point number: ";// prompt for a 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
       << '\n';
}
```

A simple computation

```
int main() // inch to cm conversion
{
  const double cm_per_inch = 2.54; // number of centimeters
                                   // per inch
  int length = 1;   // length in inches
  while (length != 0) // length == 0 is used to exit the program
  {     // a compound statement (a block)
        cout << "Please enter a length in inches: ";.
        cin >> length;
        cout << length << "in.  = "
             << cm_per_inch*length << "cm.\n";
  }
}
```

A while-statement repeatedly executes until its condition becomes false

More on istream

```
#include <iostream>
int main() {
      int sum = 0, value = 0;

      // read until end-of-file, calculating a running total of all
      // values read
      while (std::cin >> value)
            sum += value;

      std::cout << "Sum is: " << sum
                << std::endl;
      return 0;
}
```

"cin" return True/false. If we take as input what we expect (value is "int" so an "int") then it's "True". If value is "int" and we put a "string" then it'll stop. (→ false)
or whatever which is not an integer

More on istream

- When we use an istream as a condition, the effect is to test the state of the stream
- If the stream is valid (i.e., the stream hasn't encountered an error), the test succeeds
- An istream becomes invalid when we hit **end-of-file** or encounter an **invalid input** (any not integer type is read)
- An istream that is in an invalid state will cause the condition to yield **false**

Namespaces

Namespaces

- A Namespace is a named scope (piece of code)
- All the names defined by the standard library are in the **std** namespace
- Namespaces allow to **avoid inadvertent collisions** between the names we define and uses of those same names inside a library
  - We will be back to this concept later
  - Here we focus on the use of I/O facilities from the standard lib

Example

```
#include <iostream>

int main()
{
      std::cout << "Enter two numbers:"
                << std::endl;
      int v1, v2;
      std::cin >> v1 >> v2;
      std::cout << "The sum of " << v1
                << " and " << v2
                << " is " << v1 + v2 << std::endl;
      return 0;
}
```


Namespaces

- One side effect of the library use of a namespace is that when we use a name from the library, we must say explicitly that we want to use the name from the std namespace
  - Writing std::cout or std::endl uses the scope operator
  - The prefix std:: indicates that the names cout and endl are defined inside the **namespace** named **std**
- Referring to library names with this notation can be wordy

Example

```
#include <iostream>
// using declarations for names from the standard library
using std::cin;
using std::cout; using std::endl;
int main()
{
      cout << "Enter two numbers:" << endl;
      int v1, v2;
      cin >> v1 >> v2;
      cout << "The sum of " << v1 << " and "
           << v2 << " is " << v1 + v2 << endl;
      return 0;
}
```


`using` Declarations and Directives

- To avoid the tedium of
  std::cout << "Please enter stuff... \n";
- you could write a "using declaration"

```
using std::cout;        // when I say cout, I mean std::cout
cout << "Please enter stuff... \n";      // ok: std::cout
cin >> x;               // error: cin not in scope
```

- or you could write a "using directive"

```
using namespace std;  /* "make all names from namespace std
                         available"*/
cout << "Please enter stuff... \n";      // ok: std::cout
cin >> x;                                // ok: std::cin
```

Be very careful with this. Including *using namespace* in a header file can create a large set of conflicts! That's a frequent

Built-in Types

Types

- The type of a variable determines:
  - The values (domain) for the variables of that type
  - Which operations are valid and what their meanings are for that type
  - In C++ (and actually C) all variables need to be declared before they are used and their type will be fixed

Types

- C++ provides a set of types
  - E.g. bool, char, int, double
  - Called "built-in types"
- C++ programmers can define new types
  - Called "user-defined types"
  - We'll get to that eventually
- The C++ standard library provides a set of types
  - E.g. string, vector, complex
  - Technically, these are user-defined types
    - They are built using only facilities available to every user

Types and literals

literals = what can we assign

- Built-in types
  - Boolean type
    - bool
  - Character type
    - char
  - Integer types
    - int
    - and short, long and unsigned
  - Floating-point types
    - double and float
- Standard-library types
  - string
  - complex<Scalar>

- Boolean literals
  - true false
- Character literals
  - 'a', 'x', '4', '\n', '$'
- Integer literals
  - 0, 1, 123, -6, 034, 0xa3
- Floating point literals
  - 1.2, 13.345, .3, -0.54, 1.2e3, .3F
- String literals "asdf", "Howdy, all y'all!"
- Complex
  - complex<double>(12.3,99)
  - complex<float>(1.3)

If (and only if) you need more details, see the book!

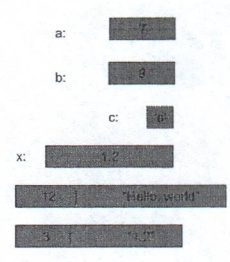
C++ Data types

S. No	DATA TYPE	Size (in bytes)	RANGE
1	Short int	2	-32768 to +32767
2	Unsigned short int	2	0 to 65535
3	long int	4	-2147483648 to 2147483647
4	Float	4	3.4e-38 to 3.4e+38
5	Char	1	-128 to 127
6	Unsigned char	1	0 to 255
7	Unsigned long int	4	0 to 4294967295
8	Double	8	1.7e-308 to 1.7e+308
9	Long double	10	1.7e-308 to 1.7e+308

Size is architecture dependent. The standard only defines ordering

Declaration and initialization

```
int a = 7;
int b = 9;
char c = 'a';
double x = 1.2;
string s1 = "Hello, world";
string s2 = "1.2";
```



Assignment and increment

check the example!

```
// changing the value of a variable
int a = 7; // a variable of type int called a
           // initialized to the integer value 7
a = 9;     // assignment: now change a's value to 9

a = a+a;   // assignment: now double a's value

a += 2;    // increment a's value by 2

++a;       // increment a's value (by 1)
```

a:

DEMO

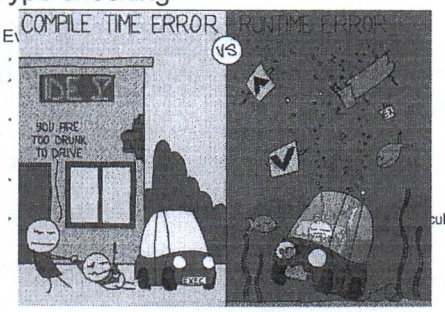
int i=0;
a=i++;
b=++i;
⇒ a=0, b=2, i=2

Type checking

- Every variable is used only according to its type
  - A variable **must** be used only after it has been **initialized** !
  - Only operations defined for the variable declared type will be applied
  - Every operation defined for a variable leaves the variable with a valid value
  - It is not always possible to detect issues at compile time (**the compiler is your friend!**)
  - Some errors are raised at runtime and they are much more difficult to correct

Type checking

- Ev
  -
  -
  -
  -
  - cult



Some examples: "implicit narrowing"

```
// Beware: C++ does not prevent you from trying to put a large value
// into a small variable (though a compiler may warn)
int main()
{
 int a = 20000;
 char c = a;
 int b = c;
 if (a != b) // != means "not equal"
      cout << "oops!: " << a << "!=" << b << '\n';
 else
      cout << "Wow! We have large characters\n";
}
```

a: 20000

c: ??

it won't give errors

DEMO

- Try it to see what value **b** gets on your machine

Some examples: uninitialized variables

```
// Beware: C++ does not prevent you from trying to use a variable
// before you have initialized it (though a compiler typically warns)
int main()
{
 int x;
 char c;
 double d;

 double dd = d;

 cout << " x: " << x << " c: " << c << " d: "
      << d << '\n';
}
```

it won't give errors

- Always initialize your variables

Some examples: uninitialized variables

```
// Beware: C++ does not prevent you from trying to use a variable
// before you have initialized it  (though a compiler typically warns)
int main()
{
 int x;          // x gets a "random" initial value
 char c;         // c gets a "random" initial value
 double d;       // d gets a "random" initial value
                 // not every bit pattern is a valid floating-point
 double dd = d;  // potential error: some implementations
                 // can't copy invalid floating-point values
 cout << " x: " << x << " c: " << c << " d: "
      << d << '\n';
}
```

- Always initialize your variables

Type conversions

- The type of a variable **defines the data** that the variable might contain and **what operations can be performed**
- Among the operations that many types support is the ability to **convert** a variable of a given type to other, related, types
- Type conversions happen automatically when we use a variable of one type where a variable of another type is expected
- Sometimes **dangerous things** might happen

Type conversions

since 42 ≠ 0 then b = 1 (True). Then, i = 1

bool are True/false: when we assign something ≠ 0 to a boolean variable we get **True**. We get false only if we assign 0 to the boolean variable.

```
bool b=42;
int i=b;
i=3.14;
double pi = i;
unsigned char c = -1;

signed char c2 = 256;
```

now i=3 because i is an integer

with this we get pi = 3.0

- If we assign an out-of-range value to a variable of signed type, the result is **undefined**
- Undefined behavior: the program **might appear to work, it might crash, or it might produce garbage values**

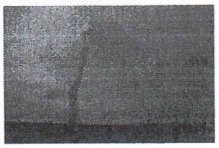
Type conversions

```
bool b=42;              // b is true
int i=b;                // i has value 1
i=3.14;                 // i has value 3
double pi = i;          // pi has value 3.0
unsigned char c = -1;   // assuming 8-bit chars,
                        // c has value 255
signed char c2 = 256;   // assuming 8-bit chars, the
                        // value of c2 is undefined
```

- If we assign an out-of-range value to a variable of signed type, the result is **undefined**
- Undefined behavior: the program **might appear to work, it might crash, or it might produce garbage values**

What can happen with type conversions

- **Very bad things do happen**, be very careful!
- **Overflows** and/or **implicit narrowing** are around the corner
- One of the most famous overflow/out of range:



Ariane 5, June 1996:
Start. 37 seconds of flight. KaBOOM!
10 years (4 satellites) and 7 billion dollars are turning into dust!!!

https://www.viva64.com/en/b/0426/?fbclid=IwAR2xwWa94pE_-9-xij-NlNb9p5BdC6t6tndNp58dGxjunAaSVJqNK3CPvrY

Expression involving unsigned types

- Although we are unlikely to intentionally assign a negative value to an object of unsigned type, we can (all too easily) write code that does so implicitly
- For example, if we use **both unsigned** and **int** values in an arithmetic expression, the **int** value ordinarily is **converted to unsigned**
- Converting an int to unsigned executes the same way as if we assigned the int to an unsigned

Expression involving unsigned types

```
unsigned u = 10;
int i = -42;
long int l = -42;
std::cout << i + i << std::endl;    // prints -84
std::cout << u + i << std::endl; // if 32-bit ints,
                                 // prints 4294967264
std::cout << i + u << std::endl; // if 32-bit ints,
                                 // prints 4294967264
std::cout << u + l << std::endl; // prints -32
```

← here i is converted to an unsigned

we get it by chance, we can't rely on it

- Converting a negative number to unsigned behaves exactly as if we had attempted to assign that negative value to an unsigned object. The value "wraps around" as before

Expression involving unsigned types

```
for (int i = 10; i >= 0; --i)
      std::cout << i << std::endl;
```

same with "i--"

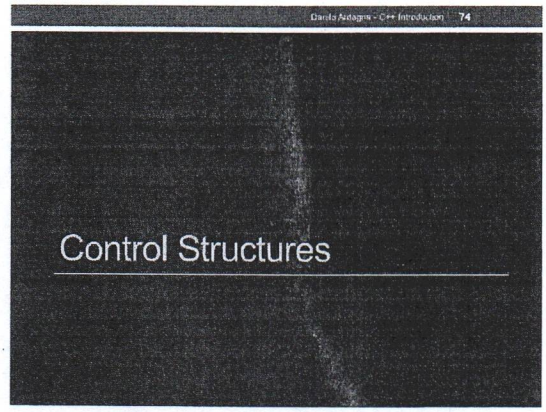
```
// WRONG: u can never be less than 0; the condition will
// always succeed
for (unsigned u = 10; u >= 0; --u)
      std::cout << u << std::endl;
```

✗ — when u=0 then u becomes the larger value that an unsigned can represents (4 bilion and something) and the loop restarts from there

```
unsigned u = 11; // start the loop one past the first element
                 // we want to print
while (u > 0) {
      --u; // decrement first, so that the last iteration will print 0
      std::cout << u << std::endl;
}
```

✓

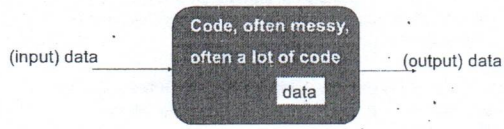
DEMO



You already know most of this

- Note:
  - You know how to do arithmetic
    - d = a+b*c
  - You know how to select
    - "if this is true, do that; otherwise do something else"
  - You know how to "iterate"
    - "do this until you are finished"
    - "do that 100 times"
  - You know how to do functions
    - "go ask Joe and bring back the answer"
    - "hey Joe, calculate this for me and send me the answer"
- What I will show you today is mostly just vocabulary and syntax for what you already know

Computation



- Input: 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: to screen, files, other output devices, other programs, other parts of a program

Computation

- Our job is to express computations
  - Correctly
  - Simply
  - Efficiently
- One tool is called Divide and Conquer
  - To break up big computations into many little ones
- Another tool is Abstraction
  - Provide a higher-level concept that hides detail
- Organization of data is often the key to good code
  - Some algorithms can be faster than others thanks to data organization
  - E.g., search in sorted and unsorted arrays

we just have to know what to provide and what we obtain

Expressions

```
// compute area:
int length = 20;    // the simplest expression: a literal (here, 20)
                    // (here used to initialize a variable)
int width = 40;
int area = length*width;        // a multiplication
int average = (length+width)/2; // addition and division
```

- The usual rules of precedence apply:
  `a*b+c/d` means `(a*b)+(c/d)` and not `a*(b+c)/d`
- If in doubt, parenthesize. If complicated, parenthesize
- Don't write "absurdly complicated" expressions:
  `a*b+c/d*(e-f/g)/h+7` *// too complicated*
- Choose meaningful names

Expressions

- Expressions are made out of operators and operands
  - Operators specify what is to be done
  - Operands specify the data for the operators to work with
- Boolean type: **bool** (true and false)
  - Equality operators: = = (equal), != (not equal)
  - Logical operators: && (and), || (or), ! (not)
  - Relational operators: < (less than), > (greater than), <=, >=
- Character type: char (e.g., 'a', '7', and '@')
- Integer types: short, int, long
  - arithmetic operators: +, -, *, /, % (remainder)
- Floating-point types: e.g., float, **double** (e.g., 12.45 and 1.234e3)
  - arithmetic operators: +, -, *, /

C++ Operators

S. No	OPERATORS	SYMBOLS
1.	Arithmetic	+,-,*,/,%
2.	Logical	&&,||,!
3.	Relational	<,>,<=,>=,==,!=
4.	Assignment	=
5.	Increment	++
6.	Decrement	--
7.	Comma	,
8.	Conditional (Ternary)	?:
9.	Bitwise	&,|,^,~,>>,<<
10.	Special Operator	sizeof
11.	Extraction	>>
12.	Insertion	<<
13.	Dynamic Memory Allocator	new
14.	Dynamic memory De-allocator	delete

Statements

- A statement is
  - a declaration, or
  - an expression followed by a semicolon, or
  - a "control statement" that determines the flow of control
- For example
  - a = b;
  - double d2 = 2.5;
  - if (x == 2)
    y = 4;
  - while (cin >> number)
    numbers.push_back(number);
  - int average = (length+width)/2;
  - return x;
- You may not understand all of these just now, but you will …

Selection

- Statements are executed sequentially
- Sometimes we must select between alternatives
- For example, suppose we want to identify the larger of two values. We can do this with an if statement

```
if (a<b)      // Note: No semicolon here
  max = b;
else          // Note: No semicolon here
  max = a;
```

- The syntax is

```
if (condition)
  statement-1   // if the condition is true, do statement-1
else
  statement-2   // if not, do statement-2
```

DEMO

Selection

- Statements are executed sequentially
- Sometimes we must select between alternatives
- For example, suppose we want to identify the larger of two values. We can do this with an if statement

```
if (a<b)      // Note: No semicolon here
  max = b;
else          // Note: No semicolon here
  max = a;
```

- The syntax is

Be careful == is the comparison operator!

```
int a=3;
int b=4;
if(a=b) is always true unless you initialize b to 0!
```

ue, do statement-1

t-2

DEMO

Iteration (while loop)

- The world's first "real program" running on a stored-program computer
  (David Wheeler, Cambridge, May 6, 1949)

```
// calculate and print a table of squares 0-99:
int main()
{
    int i = 0;
    while (i<100) {
        cout << i << "  " << square(i) << '\n';
        ++i ; // increment i
    }
}
// (No, it wasn't actually written in C++ ☺.)
```

! Remember the increment!

Iteration (while loop)

- What it takes

A loop variable (control variable);	here: i
Initialize the control variable;	here: int i = 0
A termination criterion;	here: if i<100 is false, terminate
Modify the control variable;	here: ++i
Something to do for each iteration;	here: cout << …

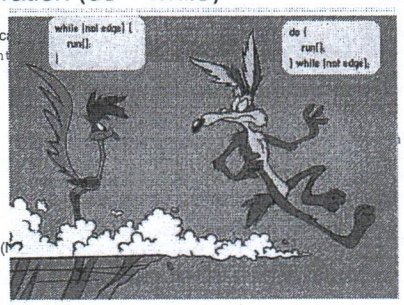
```
int i = 0;
while (i<100)  {
        cout << i << '\t' << square(i) << '\n';
        ++i ;  // increment i
}
```

Iteration (do – while)

```
// calculate and print a table of squares 0-99:
int main()
{
    int i = 0;
    do {
        cout << i << "  " << square(i) << '\n';
        ++i ; // increment i
    }
    while (i<100);
}
// (No, it wasn't actually written in C++ ☺.)
```

The difference is that the other version (only "while") tested the condition at the beginning, not at the end. Here we always enter the loop at least once.

Iteration (do – while)



Sometimes it can really make a difference

Iteration (for loop)

- Another iteration form: the for loop
- You can collect all the control information in one place, at the top, where it's easy to see

```
for (int i = 0; i<100; ++i) {
    cout << i << '\t' << square(i) << '\n';
}
```

this represents a TAB

- That is,
  - for (initialize; condition ; increment)
  - controlled statement
- Note: what is square(i)?

Functions

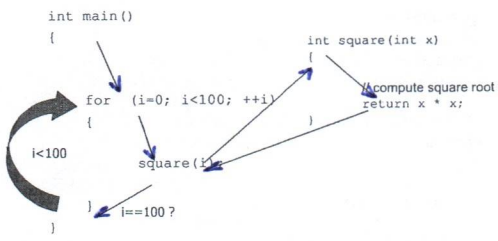
- But what was square(i)?
  - A call of the function square()

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

```
#include <iostream>
using namespace std;
int square (int x);          def.
int main() {
}
int square (int x) {         function
    return x*x;
}
```

- We define a function when we want to separate a computation because it
  - is logically separate
  - makes the program text clearer (by naming the computation)
  - is useful in more than one place in our program
  - eases testing, distribution of labor, and maintenance

Control Flow



x = formal parameter
i = actual parameter (parameter that has a value)

Functions

- Our function

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

- is an example of

```
Return_type function_name ( Parameter list )
                                        // (type name, etc.)
{
   // use each parameter in code
   return some_value;                   // of Return_type
}
```

- More technicalities are needed on parameters if we need to return more values

Arrays and Structs

Built-in arrays

- To do just about anything of interest, we need a collection of data to work on. We can store this data in an array
- For example:
  - A collection of grades that range from 0 through 100
  - Count how many grades fall into various clusters of 10

```
int main(){
const unsigned sz = 11;
// count the number of grades by clusters of ten: 0-9, 10-19,
// ... 90-99, 100

unsigned scores[sz] = {}; // 11 buckets, all value initialized to 0

unsigned grade;
while (cin >> grade)
    if (grade <= 100)
        ++scores[grade/10]; // increment the counter for the current cluster

for (i=0; i< sz; ++i ) // for each counter in scores
    cout << scores[i] << " "; // print the value of that counter
```

Goal: count the number of grades in each class (0-9, 10-19, 20-29, ...)

we only need to check if grade ≤ 100, not if 0 ≤ grade ≤ 100, because grade is an "unsigned" if someone puts a negative value then the negative will be automatically converted (**note: not** in the absolute value)

if we don't say anything the array will be initialized randomly ("unsigned scores[sz];"). If we do like this ("unsigned scores[sz] = {};") then the array will be initialized with zeros.

Built-in arrays

- A built-in array is a data structure that stores values of a single type (are **homogeneous**) that we access by position
- User-defined type
- Have **fixed size**:
  - The **dimension** must be known at **compile time**, which means that the dimension must be a ***constant expression***
  - We cannot add elements to an array
  - If you don't know exactly how many elements you need, you usually over-estimate the dimension
- Declaration

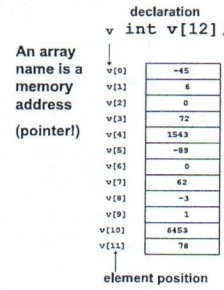
```
base_type array_name[array_size];
```

- Index range: [0, `array_size` -1]

!! CONSTANT EXPRESSION !!

Arrays and Memory allocation

- The memory space is allocated contiguously in the physical address space
- The total memory space is given by the space required to store each individual element times the size of the array



Defining and initializing built-in arrays

```
unsigned cnt = 42;  // not a constant expression
constexpr unsigned sz = 42;  // constant expression

int arr[10];  // array of ten ints

string bad[cnt];  // error: cnt is not a constant expression
string strs[get_size()];// ok if get_size is constexpr, error otherwise
```

Here we're defining an array by calling a function. If the function returns a constant expression is

- A **constant expression** is an expression whose value cannot change and that can be evaluated at compile time
  - A literal is a constant expression
  - A const variable that is initialized from a constant expression is also a constant expression
  - Functions (with some restrictions) can be used to define `constexpr`
- **The elements in an array are default initialized**

Defining and initializing built-in arrays

```
const unsigned sz = 3;

int ia1[sz] = {0, 1, 2};// array of three ints with values 0, 1, 2

int a2[] = {0, 1, 2};// an array of dimension 3

int a3[5] = {0, 1, 2};// equivalent to a3[] = {0, 1, 2, 0, 0}

string a4[3] = {"hi", "bye"};// same as a4[] = {"hi", "bye", ""}

int a5[2] = {0,1,2};// error: too many initializers
```


Defining and initializing built-in arrays

- We can **list initialize** the elements in an array
  - When we do so, we can omit the dimension
  - If we omit the dimension, the compiler infers it from the number of initializers
- If we specify a dimension, the number of initializers must not exceed the specified size
  - If the dimension is greater than the number of initializers, the initializers are used for the first elements and any remaining elements are value initialized

No Copy or Assignment

- We cannot initialize an array as a copy of another array, nor it is legal to assign one array to another
- We cannot compare two arrays through ==

```
int a[] = {0, 1, 2};  // array of three ints
int a2[] = a;  // error: cannot initialize one array with another
a2 = a; // error: cannot assign one array to another

// copy needs to be performed element by element
int a2[3];
for (unsigned i = 0; i < 3; ++i)
      a2[i] = a[i];
```


Multiple dimensional arrays: Matrices

- An array element can be another array
- Declaration

```
int matrix [SZ2][SZ1];
```

matrix [rows] [columns];

- Define a new variable matrix as an array of `SZ2` elements, and each of them is an array of `SZ1` elements
  - I.e., a matrix of `SZ2` rows and `SZ1` colums
- Allocate memory (in physical contigous locations) to store `SZ1xSZ2` elements
  - Elements are stored by rows
- Access to individual elements:
  - `matrix[i][j]` is the j-th element of i-th array, i.e. element at row i and column j

Multiple dimensional arrays

```
int ia[3][4];  // array of size 3; each element is an array of
               // ints of size 4
int ia[3][4] = {// three elements; each element is an array of size 4
{0, 1, 2, 3},// initializers for the row indexed by 0
{4, 5, 6, 7},// initializers for the row indexed by 1
{8, 9, 10, 11} // initializers for the row indexed by 2
};

// equivalent initialization without the optional nested braces for each row
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

equivalently:

Accessing multiple dimensional arrays

"size_t" is equivalent to: "unsigned"

```
constexpr size_t rowCnt = 3, colCnt = 4;

int ia[rowCnt][colCnt];  // 12 uninitialized elements
                         // for each row

for (size_t i = 0; i != rowCnt; ++i) {
      // for each column within the row
      for (size_t j = 0; j != colCnt; ++j) {
            // assign the element's positional index as its value
            ia[i][j] = i * colCnt + j; }
}
```

fancy way to obtain the above initialization

Structs

- Structs allow to declare variables able to store heterogeneous data
- Eg.: we want to store students data (name, last_name, id and 29 exam grades)

Structs

```
const unsigned grades_sz = 29;
struct Student{
                string last_name;
                string name;
                unsigned id;
                unsigned grades[grades_sz];
          };

Student s;

s.name = "Danilo";
s.last_name = "Ardagna";
s.grades[0]=30;
s.grades[1]=30;
s.grades[2]=27;
```

with this we're declaring a variable s whose type is "student"

- Individual fields can be accessed through the **dot notation**: `var_name.field_name`

Structs operations

- Operations on individual fields: depends on the base data type
- Global operator: **assignment** `student2 = student1;`
- We **cannot compare** structs through **==** (unless we provide this operator! We will see how in the next classes)
- Comparison, at this stage of the course, is implemented by performing **comparison field by field**

An example: a library

```
struct Book{
    unsigned year;
    unsigned pages;
    string author;
    string title;
};
```

Example of use of a book:

```
Book my_book;
my_book.year = 1998;
cout << "Year :" << my_book.year << endl;
cout << "Number of pages :" << my_book.pages
 << endl;
```


An example: a library

```
constexpr unsigned sz =100;

Book library[sz];
```

← we create an array of books (array of type "Book")

Finding the oldest book:

Here the goal is to get the index of the oldest book in the library

```
size_t oldest = 0;
...
for (size_t i=1; i< sz; ++i)
  if(library[i].year<library[oldest].year)
    oldest = i;
```

Then we refer to a book by positions

Declarations and Definitions

Declarations

- A declaration introduces a name into a scope
- A declaration also specifies a type for the named object
- Sometimes a declaration includes an initializer
- A name must be declared before it can be used in a C++ program
- Examples:
  - `int a = 7;` *// an int variable named 'a' is declared*
  - `const double cd = 8.7;` *// a double-precision floating-point // constant*
  - `double sqrt(double);` *// a function taking a double argument and // returning a double result*

For example

- At least three errors: (3)

```
int main()
{
    cout << f(i) << '\n';
}
```

- Add standard library declarations:

1.
```
#include<iostream>
using std::cout;

int main()
{
    cout << f(i) << '\n';
}
```

For example

```
#include<iostream>
using std::coul;

int f(int x) // declaration of f

int main()
{
    int i = 7;      // declaration of i
    cout << f(i) << '\n';
}
int f(int x) { /* … */ } // definition of f
```

2.

3.

Declare and define your own functions and variables

Definitions

- A declaration that (also) fully specifies the entity declared is called a definition
  - Examples

```
int a = 7;
int b;                          // an (uninitialized) int
double sqrt(double x) { … };    // a function with a body
struct Point { int x; int y; };
```

this works even if we don't say what is "b" at the beginning ⇒ DEFINITION (it's not like declaring functions (functions won't work till we don't define them. here "b" is already compile with "int b"))

  - Examples of declarations that are not definitions

```
double sqrt(double);  // function body missing
struct Point;         // members specified elsewhere
```


Declarations and definitions

- You cannot define something twice
  - A definition says what something is

```
int a;                          // definition
int a;                          // error: double definition
double sqrt(double d) { … }     // definition
double sqrt(double d) { … }     // error: double definition
```

- You can declare something twice
  - A declaration says how something can be used

```
double sqrt(double);            // declaration
double sqrt(double d) { … }     // definition (also a declaration)
```


Why both declarations and definitions?

- To refer to something, we need (only) its declaration
- Often we want the definition "elsewhere"
  - Later in a file
  - In another file
    - Preferably written by someone else
- Declarations are used to specify interfaces
  - To your own code
  - To libraries
    - Libraries are key: we can't write all ourselves, and wouldn't want to
- In larger programs
  - Place all declarations in **header files** to ease sharing

Header files

where do we place the declarations?

Source and header files

- As programs grow larger you make use of more code files
- C++ code files (with a .cpp extension) are not the only files commonly seen in C++ programs
- The other type of file is called a **header file**
  - Header files usually have a .h extension, but you will occasionally see them with a .hpp extension
  - Typically, header files are simply text (source code) files
- The primary purpose of a header file is to propagate declarations to code files

Source and header files

- To use an external code file you need to include its header
- At compile time, when the compiler finds an include it **copies** inside a source file the header file content
- The construct

  `#include "MyFriendLibrary.h"`

  is a "preprocessor directive" that adds declarations to your program

we use " " instead of < > to underly that MyFriendLibrary.h is not something inside the language (<iostream> is)

When the compiler reads this, the compiler goes to the file .h and copy everything. then it pastes it where the "#include "file.h" " is.

Header Files and the Preprocessor

- A header is a file that holds declarations of functions, types, constants, and other program components
- A header gives you access to functions, types, etc. that you want to use in your programs
  - Usually, you don't really care about how they are written
  - The actual functions, types, etc. are defined in other source code files
    - Often as part of libraries

Example:

MyFriendLibrary.h

```
#ifndef MY_FRIEND_LIBRARY_H
#define MY_FRIEND_LIBRARY_H

void bar(); //declaration

#endif // MY_FRIEND_LIBRARY_H
```

Compiler directives, guarantee that header files will be included in your code **only** once

MyFriendLibrary.cpp

```
#include <iostream>
#include "MyFriendLibrary.h"

void bar(){//definition
  // Do something usefull
  std::cout << "MyFriendLibrary bar"
            << std::endl;
  return;

}
```

(we can even omitt it)

MyFriendLibrary.cpp – Compile time

```
#include <iostream>
#ifndef MY_FRIEND_LIBRARY_H
#define MY_FRIEND_LIBRARY_H

void bar(); //declaration

#endif // MY_FRIEND_LIBRARY_H


void bar(){//definition
  // Do something usefull
   std::cout << "MyLibrary bar" << std::endl;
   return;
```

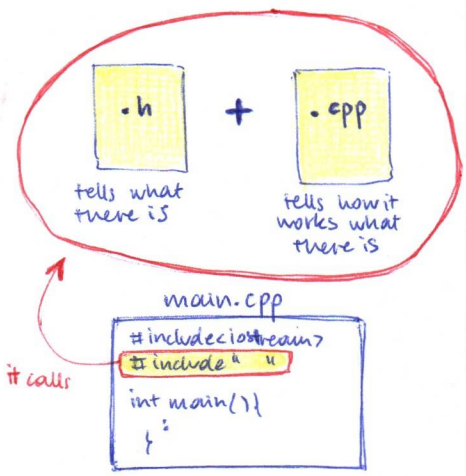
These two are saying: "if my_friend_library_h" is not yet defined then ... since "my_friend_library_h" is not defined then the compiler sees this

MyFriendLibrary.cpp – Compile time

```
#include <iostream>

#define MY_FRIEND_LIBRARY_H

void bar(); //declaration



void bar(){//definition
  // Do something usefull
   std::cout << "MyLibrary bar" << std::endl;
   return;
```


Header files and namespaces in practice

```
#include <iostream>

#include "MyFriendLibrary.h"


int main() {
    std::cout << "main function" <<'\n';
    bar(); // here we use the library version
    return 0;
}
```



Header files and namespaces in practice

```
void bar(); // declaration

int main(){
     bar(); // use
}

bar(){// definition
// your code goes here
}
```

Imagine this is your initial code and you need to extend your program relying on MyFriendLibrary where the function **bar** is available and the two functions **need to coexist** in your code

What if we want to create a function which name is "bar" but we also have the function "bar" from "MyFriendLibrary.h"?
NAMESPACES!

Header files and namespaces in practice

```
#include <iostream>

#include "MyFriendLibrary.h"

void bar(); // declaration

int main() {
    bar(); // here I would like to use my bar version
    bar(); // here I would like to use the library version
    return 0;
}

void bar(){ // definition
    std::cout << "Running main bar" << std::endl;
}
```

Header files and namespaces in practice

```
#include <iostream>

#ifndef MY_FRIEND_LIBRARY_H
#define MY_FRIEND_LIBRARY_H
void bar(); // declaration
#endif //MY_FRIEND_LIBRARY_H

void bar(); // declaration

int main() {
    bar(); // here I would like to use my bar version
    bar(); // here I would like to use the library version
    return 0;
}

void bar(){ // definition
    std::cout << "Running main bar" << std::endl;
}
```


Header files and namespaces in practice

```
#include <iostream>


#define MY_FRIEND_LIBRARY_H
void bar(); // declaration


void bar(); // declaration

int main() {
    bar(); // here I would like to use my bar version
    bar(); // here I would like to use the library version
    return 0;
}

void bar(){ // definition
    std::cout << "Running main bar" << std::endl;
}
```

What happens? Is this OK?

Header files and namespaces in practice

```
#include <iostream>

#include "MyFriendLibrary.h"

namespace foo{
    void bar(); //declaration
}

int main() {
    foo::bar(); // here I use my bar version
    bar(); // here I use the library version
    return 0;
}

void foo::bar(){ //definition

    std::cout << "Running main bar" << std::endl;

}
```


Header files and namespaces in practice

```
#include <iostream>

#include "MyFriendLibrary.h"

namespace foo{
    void bar(); //declaration
}

int main() {
    foo::bar(); // here I use my bar version
    bar(); // here I use the library version
    return 0;
}

void foo::bar(){ //definition

    std::cout << "R

}
```

Please remind to avoid to add clauses like
`using namespace std;`
in header files!

in doing so we're including all the things in the library std (we may call things that we don't want)

Why Header guards?

```
// lib1.h
#ifndef LIB_1_H
#define LIB_1_H

void f1(); //declaration

#endif // LIB_1_H
```

```
// lib1.cpp
#include "lib1.h"

void f1(){…}; //definition
```

Why Header guards?

```
// lib2.h
#ifndef LIB_2_H
#define LIB_2_H
#include "lib1.h"
void f2(); //declaration

#endif // LIB_2_H
```

we include "lib1.h" because we need something (f1()) from it

```
// lib2.cpp
#include "lib2.h"

void f2(){// use f1()}; //definition
```

in fact, for the definition of f2() we have to use f1() (for example)

part 2

And since LIB_2_H has never been defined:

main.cpp

```
#include <iostream>
#define LIB_1_H
void f1();
#define LIB_2_H
#include "lib1.h"
void f2();
:
```

Now it focus on the #include "lib1.h"

main.cpp

```
#include <iostream>
#define LIB_1_H
void f1();
#define LIB_2_H
#ifndef LIB_1_H
#define LIB_1_H
void f1();
#endif
void f2();
:
```

But the whole part of LIB_1_H has already been defined before, so:

main.cpp

```
#include <iostream>
#define LIB_1_H
void f1();
#define LIB_2_H
void f2();
:
```


Why Header guards?

```
#include <iostream>
#include "lib1.h"
#include "lib2.h"


int main() {
    f1();
    f2();
    return 0;
}
```


Why Header guards?

```
#include <iostream>
#ifndef LIB_1_H
#define LIB_1_H

void f1(); //declaration

#endif // LIB_1_H
#include "lib2.h"


int main() {
    f1();
    f2();
    return 0;
}
```

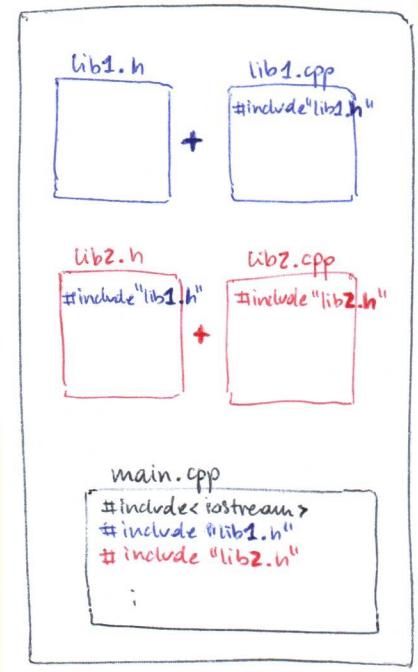

Why Header guards?

```
#include <iostream>

#define LIB_1_H

void f1(); //declaration


#include "lib2.h"


int main() {
    f1();
    f2();
    return 0;
}
```


Why Header guards?

```
#include <iostream>

#define LIB_1_H

void f1(); //declaration

#ifndef LIB_2_H
#define LIB_2_H
#include "lib1.h"
void f2(); //declaration

#endif // LIB_2_H

int main() {
    f1();
    f2();
    return 0;
}
```

part 1



As we saw, in the runtime the include becomes the header!

main.cpp

```
#include<iostream>
#ifndef LIB_1_H
#define LIB_1_H
void f1();
#endif
#include "lib2.h"
:
```

since LIB_1_H has never been defined, it becomes:

main.cpp

```
#include<iostream>
#define LIB_1_H
void f1();
#include "lib2.h"
:
```

And then:

main.cpp

```
#include<iostream>
#define LIB_1_H
void f1();
#ifndef LIB_2_H
#define LIB_2_H
#include "lib1.h"
void f2();
#endif
:
```

Why Header guards?

```
#include <iostream>

#define LIB_1_H

void f1(); //declaration


#define LIB_2_H
#include "lib1.h"
void f2(); //declaration


int main() {
    f1();
    f2();
    return 0;
}
```


Why Header guards?

```
#include <iostream>

#define LIB_1_H

void f1(); //declaration


#define LIB_2_H
#ifndef LIB_1_H
#define LIB_1_H

void f1(); //declaration

#endif // LIB_1_H
void f2(); //declaration
```

```
int main() {
    f1();
    f2();
    return 0;
}
```


Why Header guards?

```
#include <iostream>

#define LIB_1_H

void f1(); //declaration

#define LIB_2_H

void f2(); //declaration


int main() {
    f1();
    f2();
    return 0;
}
```


Classes

Very similar to "struct". Here we're allowed to introduce FUNCTIONS that work on these structures

Classes

- A class is a (user-defined) type that specifies how objects of its type can be created and used
- A class directly represents a concept in a program
  - If you can think of "it" as a separate entity, it is plausible that it could be a class or an object of a class
  - Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, picture on screen, dialog box, graph, window, temperature reading, clock
- In C++ (as in most modern languages), a class is the key building block for large programs
  - And very useful for small ones also
- Classes implements a very important concept: **Abstract Data Type**

Members and member access

- One way of looking at a class;

```
class X {// this class' name is X
    // data members (they store information)
    // function members (they do things, using the information)
};
```

- Example

```
class X {
public:
    int m;   // data member (usually a very bad idea to have public data
             // member !)
    int mf(int v) { int old = m; m=v; return old; }// function member
};

X var;              // var is a variable of type X
var.m = 7;          // access var's data member m
int x = var.mf(9);// call var's member function mf()
```

public = usable by everybody

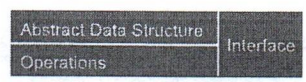
this modifies the value of m!

→ after this, var.m = 9 !!

Abstract Data Type (ADT)

- Type:
  - Domain
  - Set of operations
- Module:
  - Distinguish
    - What the software DOES:
      - Set of services provided (interface)
    - How the software is built:
      - Module internals (implementation)
  - Interface: a contract among a module and its users

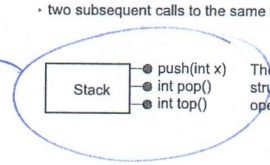
ADT



- Module that defines a new type, and all the operations that allow to manipulate instances (e.g., the type stack)
- Exports
  - one type (only the name)
  - operations for manipulating objects (i.e. data) type
- Hides
  - type structure
  - operations implementation
- The user can create objects (data) of the type specified by the module and manipulate these objects through the operations defined within the module

ADT

- Interface: set of functions / procedures (such as libraries)
- An object has a **state** = values of the hidden data structure
  - couples the data structure and procedures that manipulate it
  - two subsequent calls to the same function can give different results



The stack module hides the data structure and implementation of operations

it's like a pile of dishes:

dishes

- push(int x) create a new dish and it adds it on top:

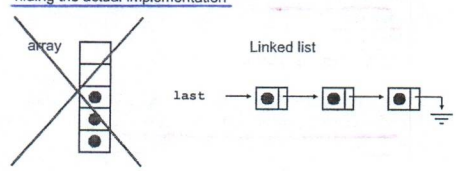
- pop() eliminate the last

- top() allows to see the top element without removing it

"  "

ADT

- **Abstract:**
  - the module provides to the users an abstract view of the object hiding the actual implementation



two possible implementations of the "stack"

- If you decide to change the implementation later, all the code using your ADT (Class!) **does not require any change!!**

Concept of Class and Object

- "**Class**" refers to a blueprint. It defines the data members and methods the objects support. It is the basic unit of **Encapsulation.** It can be considered as the collection of similar types of objects

data members + methods

- "**Object**" is an instance (properties) of a class. Each object has a class which defines its data and behavior. **Objects have states**

Concept of Class and Object

- "**Class**"

- "**Object**"


Dido


Giotto


Penny

Definition of a "Class"

- A class is an ADT characterized by:
  1. Interface (public part):
     - properties (or variables or data members, again **don't do it** in general!)
     - procedures / functions (or methods)
  2. Implementation (private part):
     - properties (or variables or data members) **- hidden data structure**
     - procedures / functions (or methods) **-** support routine implementation

Definition of an "Object"

- An object is a computational entity that:
  1. Is an instance of a Class
  2. Encapsulates some state
  3. Is able to perform actions, or methods, on this state
  4. **Communicates with other objects invoking their methods**

methods may change the internal state of an object

Classes – C++ general syntax

```
class X {// this class' name is X
public:// public members -- that' s the interface to users
      //  (accessible by all)
      // functions
      // types
      // data (often best kept private!)

private:// private members -- that' s the implementation details
      // (accessible by members of this class only)
      // functions
      // types
      // data
};
```

Classes basic example

```
class X {
private:
      int data;
      int mf1();
      // …
public:
      int mf2();

};

X x;          // variable x of type X
x.data = 4;  // error: data is private (i.e., inaccessible)
int y = x.mf1();// error: mf1 is private
int z = x.mf2(); // ok: mf2 is public
```


Stack: array based implementation

we will store up to 10 elements

we want users to have just 3 options: push(), pop(), top()

```
const int max_size = 10; // Maximum size of Stack

class Stack {

public:
    // constructor: initialize Stack data structure
    Stack() { top_index = -1; }
    void push(int x);
    int pop();
    int top() const;
    // destructor: run when Stack data goes out of scope
    ~Stack() { std::cout << "Stack deallocated";}

private:
    bool isEmpty() const;
    bool isFull() const;

    int top_index;
    int a[max_size]
};
```

index of the elem that we can access from the top

Stack: array based implementation

CONSTRUCTOR (which name is the same as the class)

DESTRUCTOR (which name always starts with a tilde (~) and follows with the name of the class)

```
const int max_size = 10; // Maximum size of Stack

class Stack {

public:
    // constructor: initialize Stack data structure
    Stack() { top_index = -1; }
    void push(int x);
    int pop();
    int top() const;
    // destructor: run when Stack data goes out of scope
    ~Stack() { std::cout << "Stack deallocated";}

private:
    bool isEmpty() const;
    bool isFull() const;

    int top_index;
    int a[max_size]
};
```

!! const means that the method execution doesn't change the object state

If a method is CONST then it's **not** going to change the internal state

Constructor

we run it whenever we allocate an object

- To define the initial state of the objects in each class we must define a particular method the so called ***constructor***
- In C ++ the ***constructor*** is a method that has the **same class name**
- The constructor is automatically invoked by the run-time support of the language every time an object is created
- The constructor can have parameters
- Often the language provides a **default constructor** (without any parameter!) when it has not been defined by the developer

Destructor

- The destructor operates inversely to the constructors and is automatically **invoked** every time an **object goes out of scope**
- The destructor is a method with the name of the class prefixed by a tilde (~)
- It has no return value and takes no parameters
- There is always **only one** destructor for a given class
- Destructors do whatever work is needed to free the resources used by an object
  - Extremely important when we use pointers!
- C++ provides a **default destructor which does nothing** (as our!) when it has not been defined by the developer

Stack: array based implementation

```
Stack s;
s.push(10);
s.push(20);
s.push(30);
cout << s.pop()
     << " Popped from stack\n";
```

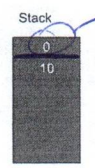

Stack: array based implementation

```
Stack s;
s.push(10);
s.push(20);
s.push(30);
cout << s.pop()
     << " Popped from stack\n";
```

this is because we wrote it in the constructor (we wrote: "Stack() {top_index = -1;}) and so, whenever we create an object, this happens.

Stack: array based implementation

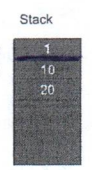
```
Stack s;
s.push(10);
s.push(20);
s.push(30);
cout << s.pop()
     << " Popped from stack\n";
```



now the index becomes valid (since we have an element in the position 0 (a[0]))

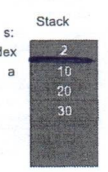
Stack: array based implementation

```
Stack s;
s.push(10);
s.push(20);
s.push(30);
cout << s.pop()
     << " Popped from stack\n";
```



Stack: array based implementation

```
Stack s;
s.push(10);
s.push(20);
s.push(30);
cout << s.pop()
     << " Popped from stack\n";
```



Stack: array based implementation

```
Stack s;
s.push(10);
s.push(20);
s.push(30);
cout << s.pop()
     << " Popped from stack\n";
```

Stack: array based implementation

the class is also a namespace

```
bool Stack::isEmpty() const
{
    return top_index < 0;
}

bool Stack::isFull() const
{
    return top_index >= (max_size - 1);
}
```

[top_index < 0] ⟹ return "True"

when is it full the stack?
When we're indexing the last element of a[], corresponding to max_size - 1

Stack: array based implementation

```
void Stack::push(int x)
{
    if (isFull()) {
        cout << "Stack Overflow";
    }
    else {
        a[++top_index] = x;
        cout << x << " pushed into stack\n";
    }
}
```

here we don't need to write Stack::isFull()

Stack: array based implementation

```
int Stack::pop()
{
    if (isEmpty()) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top_index--];
        return x;
    }
}

int Stack::top() const
{
    if (isEmpty()) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top_index];
        return x;
    }
}
```

we first return the element at "top_index" and then we do the "--" to make the element not accessible anymore

here we only return the element at the top-index, we're not decreasing top-index

Stack: array based implementation

```
Stack s;
s.push(10);
s.push(20);
s.push(30);
cout << s.pop()
     << " Popped from stack\n";
```

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 Popped from stack
Stack deallocated
```

Code organization

```
const int max_size = 10; // Maximum size of Stack
class Stack {
public:
    Stack() { top_index = -1; }
    void push(int x);
    int pop();
    int top() const;
   ~Stack() { std::cout << "Stack deallocated"}
private:
    bool isEmpty() const;
    bool isFull() const;

    int top_index;
    int a[max_size];
};
```

Stack.h (or Stack.hh or Stack.hpp)

declaration of the class

```
bool Stack::isEmpty() const
{
    return top_index < 0;
}
bool Stack::isFull() const
{
    return top_index >= tmax_size - 1);
}
```

```
void Stack::push(int x)
{
    ...
}
int Stack::pop()
{
    ...
}
int Stack::top() const
{
    ...
}
```

Stack.cpp

definition of the class' functions

Source files

Stack.h:

```
// declarations
class Stack {
int pop();
...
};
```

Stack.cpp:

```
#include "Stack.h"
//definitions:
Stack::pop()
{/* ... */}
...
```

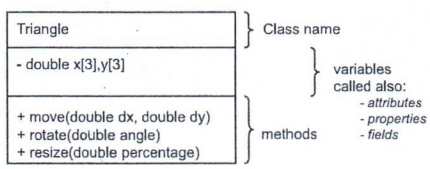
use.cpp:

```
#include " Stack.h"
...
Stack s;
...
```

- The header file **Stack.h** declares an interface between user code and implementation code (usually in a library)
- The same **#include** declarations in both **.cpp** files (definitions and uses) ease consistency checking

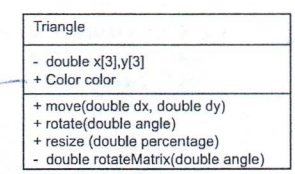
Classes

- Graphical representation
  - three distinct sections
    - To each method and each attribute a symbol that indicates the visibility (public "+" private "-") can be associated
    - The **name** of the **class** and the set of **methods** and **attributes** marked with **"+"** symbol are the class **interface**



Class

- Also attributes can be public
- Also methods can be private
- **Attributes may be of any type, also classes** (e.g., color)



The implementation code can access all the attributes and all methods of the class (including private ones)

if we implement the color as a class then: Color color

Vectors

we don't need to say the length

Vectors

- We can think to a vector as a variable sized array
- Vectors are often referred to as **containers** because they "contain" other objects

```
// read some temperatures into a vector:
int main()
{
    vector<double> temps; // declare a vector of type double to store
                          // temperatures – like 21.4
    double temp;          // a variable for a single temperature value
    while (cin>>temp)     // cin reads a value and stores it in temp
        temps.push_back(temp); // store the value of temp in the vector
    // ... do something ...
}
// cin>>temp will return true until we reach the end of file or encounter
// something that isn't a double: like the word "end"
```

whenever temp is not a double we exit the loop

we're running a method (push_back) of temps

push_back() : adding an element at the end

Vectors

- To use a vector, we must include the appropriate header. In our examples, we also assume that an appropriate using declaration is made

```
#include <vector>
using std::vector;
```

- A vector is a **class template**
- They are provided by the STL (Standard Template Library)

Vectors

blueprint type-independent

- C++ has both **class and function templates**
- Templates are not themselves functions or classes. Instead, they can be thought of as **instructions** to the **compiler** for **generating** classes or functions
- The process that the compiler uses to create classes or functions from templates is called **instantiation**
- When we use a template, we specify what kind of class or function we want the compiler to instantiate (`vector<double>` in the previous example)

Vectors

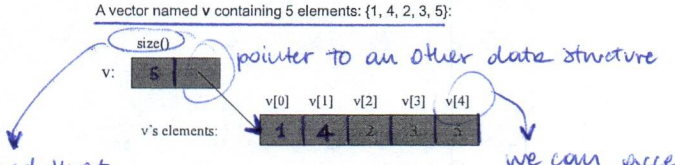
```
vector<int> ivec;  // ivec holds objects of type int
vector<Sales_item> Sales_vec;// holds Sales_items
                             // imagine Sales_item
                             // as a struct or a class
vector<vector<string>> file;// vector whose elements
                            // are vectors
```

- We can define vectors to hold objects of most any type

Vectors

- Vector is the most useful standard library data type
  - a **vector<T>** holds a sequence of values of type **T**

A vector named **v** containing 5 elements: {1, 4, 2, 3, 5}:



pointer to an other data structure

method that we can call on the vector object to know how many elements we have

we can access to every element by the index, like in the array case

Vectors

```
vector<int> v;          // start off empty
```

v: [0 | /]

```
v.push_back(1);         // add an element with the value 1
```

v: [1 | ●] → [1]

```
v.push_back(4);         // add an element with the value 4 at end
                        // ("the back")
```

v: [2 | ●] → [1 | 4]

```
v.push_back(3);         // add an element with the value 3 at end
                        // ("the back")
```

v: [3 | ●] → [1 | 4 | 3]

Vectors

- Once you get your data into a vector you can easily manipulate it

```
// compute mean (average) and median temperatures:
int main()
{
 vector<double> temps;   // temperatures, e.g. 21.4
 double temp;
 while (cin>>temp)
    temps.push_back(temp);  // read and put into vector

 double sum = 0;                    // sums temperatures
 for (size_t i = 0; i< temps.size(); ++i) sum += temps[i];

 cout << "Mean temperature: " << sum/temps.size() << '\n';
 sort(temps.begin(), temps.end());
 cout << "Median temperature: " << temps[temps.size()/2] <<
 '\n';
}
```

! it's more convenient than unsigned because it guarantees that we can reach the last element of the array/vector whatever the size is

we access to elements like we do in arrays

to use "sort" we have to write also:
#include<algorithm>

Vectors

- Once you get your data into a vector you can easily manipulate it
- Initialize with a list

```
v = { 1, 2, 3, 5, 8, 13 }; // initialize with a list
```

- Often we want to look at each element of a vector in turn:

```
                    // list all elements
for (size_t i = 0; i< v.size(); ++i) cout << v[i] << '\n';
```

!!!

```
// there is a simpler kind of loop for that (a range-for loop):
for (int x : v) cout << x << '\n'; // list all elements
// for each x in v ...
```


Range-for is available also for built-in arrays

!

```
int main(){
const unsigned array_size = 11;
// count the number of grades by clusters of ten: 0-9, 10-19,
// ... 90-99, 100
unsigned scores[array_size] = {}; // 11 buckets, all value initialized
                                  // to 0
unsigned grade;
while (cin >> grade) {
      if (grade <= 100)
            ++scores[grade/10]; // increment the counter for
                                // the current cluster
}
for (unsigned i : scores) // for each counter in scores
      cout << i << " "; // print the value of that counter
cout << endl;

}
```

!

Ways to initialize a vector

vector <T> v1	Vector that holds objects of type *T*. Default initialization, *v1* is empty
vector <T> v2(v1)	*v2* has a copy of each element in *v1*
vector <T> v2=v1	As above
vector <T> v3(n, val)	*v3* has *n* elements with value *val*
vector <T> v4(n)	*v4* has *n* copies of a value-initialized object
vector <T> v5{a,b,c,...}	*v5* has as many elements as there are initializers; elements are initialized by corresponding initializers
vector <T> v5={a,b,c,...}	As above

DEMO

Ways to initialize a vector

- A way to provide element values in C++ 11 is to provide a list of zero or more initial element values in curly braces:

```
vector<string> articles = {"a", "an", "the"};

vector<string> v5{"hi"}; // list initialization: v5 has one element

vector<string> v6(v5); // OK, v6 has its own elements, are
                       // copies of v5's!

vector<string> v7(10); // v7 has ten default-initialized elements

vector<string> v8(10, "hi"); // v8 has ten elements with value
                             // "hi"
```


Vectors

- Vectors **grow efficiently**
- Because vectors grow efficiently, it is often unnecessary (can result in poorer performance) to define a vector of a specific size
  - The exception to this rule is if all the elements actually need the same value
- If differing element values are needed, it is usually more efficient to define an empty vector and add elements as the values we need become known at run time

Vectors

- The fact that we can easily and efficiently add elements to a vector greatly simplifies many programming tasks
- This simplicity imposes a new obligation on our programs: We must **ensure** that **any loops** we write are **correct even** if the loop **changes the size** of the vector
- We will be back to this during the course

loops must be correct even when we change the length of the vector **INSIDE** the loop

Summary of Vectors operations

v.empty()	Returns true if *v* is empty; false otherwise
v.size()	Returns number of elements in *v*
v.push_back(t)	Adds an element with value *t* in *v*
v[n]	Returns a reference to the element at position *n* in *v*
v1=v2	Replaces the elements in *v1* with a copy of the elements in *v2*
v1={a, b, c, ...}	Replaces the elements in *v1* with a copy of the elements in the comma-separated list
v1==v2	*v1* and *v2* are equal if the have the same number of elements and each element in *v1* is equal to the corresponding one in *v2*
v1!=v2	
<,<=,>,>=	Dictionary order

element-wise

we couldn't do it with arrays!!

The first element that differs determines the value of the comparison:

Case I:

v1 : 0 3 5 6
v2 : 0 3 7 8 9

⟹ v1 < v2

Case II:

v1 : 0 3
v2 : 0 3 7 8 9

⟹ v1 < v2

Vectors

- We can fetch a given element using the subscript operator (i.e., indexing starting from 0)
- We can compute an index and directly fetch the element at that position

Vectors

```
// count the number of grades by clusters of ten: 0-9,
// 10-19, . .. 90-99, 100
vector<unsigned> scores(11, 0); // 11 buckets, all initially 0
unsigned grade;
while (cin >> grade) { // read the grades
      if (grade <= 100) // handle only valid grades
            ++scores[grade/10]; // increment the counter for
                                // the current cluster
}
```

We're initializing a vector with 11 zeros

C++ Vectors vs. Matlab

- Programmers new to C++ think that subscripting a vector adds elements; **it does not!!!**

```
vector<int> ivec; // empty vector
for (int ix = 0; ix != 10; ++ix)
      vec[ix] = ix; // disaster: ivec has no elements
```

```
for (int ix = 0; ix != 10; ++ix)
      ivec.push_back(ix); // ok: adds a new element with
                          // value ix
```


Vectors

```
vector<int> ivec; // empty vector
cout << ivec[0]; // error: ivec has no elements!

vector<int> ivec2(10); // vector with ten elements
cout << ivec2[10]; // error: ivec2 has elements 0 . . . 9
```

A good way to ensure that subscripts are in range is to avoid subscripting altogether by using a **range for** whenever possible

Example – Word List

```
// "boilerplate" left out

vector<string> words; AND
for (string s; cin>>s && s != "quit"; )// && means AND
      words.push_back(s);

sort(words.begin(),words.end()); // sort the words we read

for (string s : words)
      cout << s << '\n';

/*
   read a bunch of strings into a vector of strings, sort
   them into lexicographical order (alphabetical order),
   and print the strings from the vector to see what we have.
*/
```

we keep adding strings until we write: "quit"

Word List – Eliminate Duplicates

```
// Note that duplicate words were printed multiple times. For
// example "the the the". That's tedious, let's eliminate duplicates:

vector<string> words;
for (string s; cin>>s && s!= "quit"; )
      words.push_back(s);

sort(words.begin(),words.end());

for (size_t i=1; i<words.size(); ++i)
      if(words[i-1]==words[i])
            "get rid of words[i]"    // (pseudocode)
for (string s : words)
      cout << s << '\n';

//  there are many ways to "get rid of words[i]"; many of them are messy
//  (that's typical). Our job as programmers is to choose a simple clean
//  solution – given constraints – time, run-time, memory.
```

Example (cont.) Eliminate Words!

The easiest way is to rely on an other vector where we store words without replicas

```
// Eliminate the duplicate words by copying only unique words:
vector<string> words;
for (string s; cin>>s && s!= "quit"; )
     words.push_back(s);
sort(words.begin(),words.end());
vector<string>w2;
if (words.size()>0) {
     w2.push_back(words[0]);
     for (size_t i=1;i<words.size();++i)// note: not a range-for
          if(words[i-1]!=words[i])
               w2.push_back(words[i]);
}
cout<< "found " << words.size()-w2.size() <<  "
duplicates\n";
for (string s : w2)
     cout << s << "\n";
```


Readings

Strings

- A **string** is a variable-length sequence of characters
- To use the string type, we must include the string header. Because it is part of the library, string is defined in the std namespace

```
#include <string>
using std::string;
```


Defining and initializing strings

```
string s1;  // default initialization; s1 is the empty string
string s2=s1;// s2 is a copy of s1
string s3 = "hiya";// s3 is a copy of the string literal
string s4(10, 'c');// s4 is ccccccccc
```


Direct and Copy forms of initialization

- When we initialize a variable using =, we are asking the compiler to **copy initialize** the object by copying the initializer on the right-hand side into the object being created
- Otherwise, when we omit the =, we use **direct initialization**

```
string s5 = "hiya";  // copy initialization
string s6("hiya");  // direct initialization
string s7(10, 'c');  // direct initialization; s7 is ccccccccc
```

getline

```
int main()
{
string line;

while (getline(cin, line)) // read until end-of-file
     if (!line.empty() && line.size() < 80)
          cout << line<< endl; // write each line
return 0;
}
```

- `auto len = line.size();` *// len has type string::size_type*

String comparison

- The string class defines several operators that compare strings
- The comparisons are **case-sensitive**
- The equality operators (== and !=) test whether two strings are equal or unequal, respectively
  - Two strings are equal if they are the same length and contain the same characters
- The relational operators <, <=, >, >= test whether one string is less than, less than or equal to, greater than, or greater than or equal to another
  - Use the same strategy as a (case-sensitive) dictionary:
    - If two strings have different lengths and if every character in the shorter string is equal to the corresponding character of the longer string, then the shorter string is less than the longer one
    - If any characters at corresponding positions in the two strings differ, then the result of the string comparison is the result of comparing the first character at which the strings differ

Adding literals and strings

```
string s1 = "hello", s2 = "world"; // no punctuation in s1
                                   // or s2
string s3 = s1 + ", " + s2 + '\n';
string s4 = s1 + ", "; // ok: adding a string and a literal
string s5 = "hello" + ", "; // error: no string operand
string s6 = s1 + ", " + "world"; // ok: each + has a string
                                 // operand
string s7 = "hello" + ", " + s2; // error: can't add
                                 // string literals
```


Strings C++ vs. C

- There are two ways to access individual characters in a string: We can use a subscript or an iterator
- The subscript operator (the **[ ] operator**) takes a **string::size_type** value that denotes the position of the character we want to access. The operator returns a reference to the character at the given position
- Subscripts for strings start at zero; if s is a string with at least two characters, then s[0] is the first character, s[1] is the second, and the last character is in s[s.size() - 1]
- Although C++ supports C-style strings, they should not be used by C++ programs. **C-style strings are a surprisingly rich source of bugs** and are the root cause of many security problems. They're also harder to use!

String operations

os<<s	Writes *s* onto output stream *os*. Return *os*
is>>s	Reads whitespace-separated string from *is* into *s*. Return *is*
getline(is,s)	Reads a line into *s*. Return *is*
s.empty()	true if *s* is empty; false otherwise
s.size()	Returns number of characters in *s*
s[n]	Returns a reference to the char at position *n* in *s*
s1+s2	Returns a string obtained by concatenating *s1* and *s2*
s1=s2	Replaces the characters in *s1* with a copy of *s2*
s1==s2	*s1* and *s2* are equal if they contain the same characters
s1!=s2	
<,<=,>,>=	Comparisons are case sensitive and use dictionary order

Dealing with the characters in a string

isalnum(c)	true if *c* is a letter or digit
isalpha(c)	true if *c* is a letter
iscntrl(c)	true if *c* is a control character
isdigit(c)	true if *c* is a digit
isgraph(c)	true if *c* is not a space but is printable
islower(c)	true if *c* is lowercase
isprint(c)	true if *c* is a printable char (space or visible repr.)
ispunct(c)	true if *c* is a punctuation character

Dealing with the characters in a string

isspace(c)	true if *c* is whitespace (includes space, tab, return, newline, etc.)
isupper(c)	true if *c* is an upper case character
isxdigit(c)	true if *c* is a hexadecimal digit
tolower(c)	If *c* is an uppercase letter, returns its lowercase equivalent otherwise returns c unchanged
toupper(c)	If *c* is a lowercase letter, returns its uppercase equivalent otherwise returns c unchanged

Processing every character? Use range-based for

```
string str("some string");
// print the characters in str one character to a line
for (auto c : str) // for every char in str
      cout << c << endl; // print the current character
                         // followed by a newline

string s("Hello World!!!");
// punct_cnt has the same type that s.size
string::size_type punct_cnt = 0;
// count the number of punctuation characters in s
for(auto c: s) // for every char in s
      if (ispunct(c)) // if the character is punctuation
            ++punct_cnt; // increment the punctuation counter
cout << punct_cnt << " punctuation characters in "
<< s << endl;
```


References

- Lippman Chapters 1, 2 and 3

Credits

- Bjarne Stroustrup. www.stroustrup.com/Programming