DM560

Introduction to Programming in C++

Types, Computations

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Bjarne Stroustrup*]

Outline

Most programming tasks involve manipulating data. Today, we will:

- describe how to input and output data
- present the notion of a variable for holding data
- introduce the central notions of “Type” and “Type Safety”

Outline

1. Data Types

2. Type safety

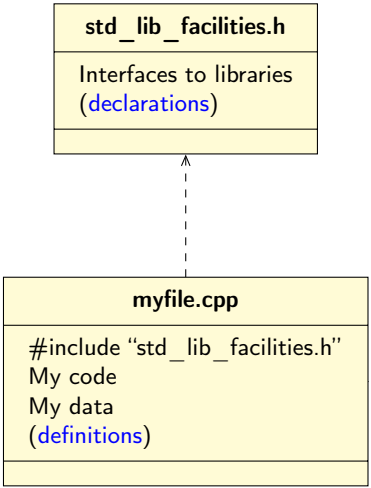
3. Computation

Outline

Input and Output

```
// read first name:
#include "std_lib_facilities.h"            // our course header
int main()
{
  cout << "Please enter your first name (followed " << "by 'enter'):\n";
  string first_name;
  cin >> first_name;
  cout << "Hello, " << first_name << '\n';
}
// - 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)
```

Source Files



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 (“a word”)
  – White space: 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';
}
// We left out here the line #include "std_lib_facilities.h" to save space and
// reduce distraction
// Don't forget it in real code!
// Similarly, we leave out the Windows-specific keep_window_open();
```

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

- `cin >>` reads a word
- `cout <<` writes
- `+` concatenates
- `+= s` adds the string `s` at end
- `++` is an error
- `-` is an error
- ...

Integers and floating-point numbers

- `cin >>` reads a number
- `cout <<` writes
- `+` adds
- `+= n` increments by the int `n`
- `++` increments by 1
- `-` subtracts
- ...

The type of a variable determines which operations are valid and what their meanings are for that type (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`
  - `time$to$market`
  - `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`

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

Types and Literals

Built-in types	Types	Literals
Boolean	`bool`	`true false`
Character	`char`	`'a', 'x', '4', 'n', '$'`
Integer	`int`, `short`, `long`	`0, 1, 123, -6, 034, 0xa3`
Floating-point	`double` and `float`	`1.2, 13.345, .3, -0.54, 1.2e3, .3F`

Standard-library types	Types	Literals
String	`string`	``` ``asdf'', ``Howdy, all y'all!'' ```
Complex Numbers	`complex<Scalar>`	`complex<double>(12.3,99)`
		`complex<float>(1.3F)`

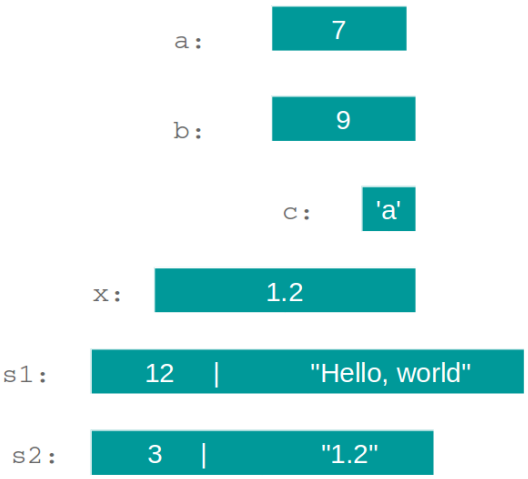
If (and only if) you need more details, see the book!

Types

- C++ provides a set of types called built-in types
  E.g. `bool, char, int, double`
- 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

Declaration and Initialization

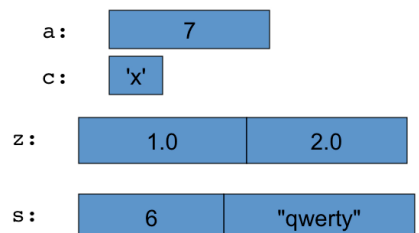
```
int a = 7;

int b = 9;

char c = 'a';

double x = 1.2;

string s1 = "Hello,
  world";
string s2 = "1.2";
```



Objects

- An object is some memory that can hold a value of a given type
- A variable is a named object
- A declaration names an object

```
int a = 7;
char c = 'x';
complex<double>
  z(1.0,2.0);
string s = "qwerty";
```



Outline

Type Safety

- Language rule: type safety
  Every object will be used only according to its type
  - A variable will be used only after it has been initialized
  - Only operations defined for the variable's declared type will be applied
  - Every operation defined for a variable leaves the variable with a valid value

- Ideal: static type safety
  A program that violates type safety will not compile
  The compiler reports every violation (in an ideal system)

- Ideal: dynamic type safety
  If you write a program that violates type safety it will be detected at run time
  $\rightsquigarrow$ Some code (typically "the run-time system") detects every violation not found by the compiler (in an ideal system)

Type Safety

- Type safety is a very big deal
  Try very hard not to violate it
  "when you program, the compiler is your best friend"
  But it won't feel like that when it rejects code you're sure is correct

- C++ is not (completely) statically type safe
  - No widely-used language is (completely) statically type safe
  - Being completely statically type safe may interfere with your ability to express ideas

- C++ is not (completely) dynamically type safe
  - Many languages are dynamically type safe
  - Being completely dynamically type safe may interfere with the ability to express ideas and often makes generated code bigger and/or slower

- Almost all of what you'll be taught here is type safe
  We'll specifically mention anything that is not

Assignment and Increment


```
// 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)
```

7

9

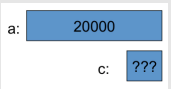
18

20

21

A type-safety violation
(“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";
}
```



⇝ Try it to see what value b gets on your machine

A Type-safety Violation

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 value
  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 – beware: 'debug mode' may initialize (valid exception to this rule: input variable)

A Technical Detail

- In memory, everything is just bits; type is what gives meaning to the bits

  (bits/binary) 01100001 is the int 97 is the char 'a'
  (bits/binary) 01000001 is the int 65 is the char 'A'
  (bits/binary) 00110000 is the int 48 is the char '0'

```
char c = 'a';
cout << c;       // print the value of character c, which is a
int i = c;
cout << i;       // print the integer value of the character c, which is 97
```

- This is just as in "the real world":
  What does "42" mean?
  You don't know until you know the unit used
  Meters? Feet? Degrees Celsius? $s? a street number? Height in inches? ...

About Efficiency

- For now, don't worry about efficiency
  Concentrate on correctness and simplicity of code

- C++ is derived from C, which is a systems programming language
  - C++'s built-in types map directly to computer main memory
    a `char` is stored in a byte
    an `int` is stored in a word
    a double fits in a floating-point register
  - C++'s built-in operations map directly to machine instructions
    an integer `+` is implemented by an integer add operation
    an integer `=` is implemented by a simple copy operation
  - C++ provides direct access to most of the facilities provided by modern hardware

- C++ help users build safer, more elegant, and efficient new types and operations using built-in types and operations.
  E.g., `string`
  Eventually, we'll show some of how that's done

Another Simple Computation


```
// inch to cm and cm to inch conversion:

int main()
{
  const double cm_per_inch = 2.54;
  int val;
  char unit;
  while (cin >> val >> unit) {  // keep reading
    if (unit == 'i')            // 'i' for inch
      cout << val << "in == " << val*cm_per_inch << "cm\n";
    else if (unit == 'c')       // 'c' for cm
      cout << val << "cm == " << val/cm_per_inch << "in\n";
    else
      return 0; // terminate on a 'bad unit', e.g. 'q'
  }
}
```

C++11 Hint

- All language standards are updated occasionally
  Often every 5 or 10 years
- The latest standard has the most and the nicest features
  Currently C++17
- The latest standard is not 100% supported by all compilers
  GCC (Linux) and Clang (Mac) are fine
  Microsoft C++ is OK
  Other implementations (many) vary

C++14 Hint

You can use the type of an initializer as the type of a variable

```
// 'auto' means 'the type of the initializer'
auto x = 1;      // 1 is an int, so x is an int
auto y = 'c';    // 'c' is a char, so y is a char
auto d = 1.2;    // 1.2 is a double, so d is a double

auto s = ``Howdy'';  // ``Howdy'' is a string literal of type const char[]
                     // so don't do that until you know what it means!

auto sq = sqrt(2);       // sq is the right type for the result of sqrt(2)
                         // and you don't have to remember what that is
auto duh;                // error: no initializer for auto
```

Outline

Overview

In this unit, we learn the basics of computation:

- Computation
  - What is computable? How best to compute it?
  - Abstractions, algorithms, heuristics, data structures
- Language constructs and ideas
  - Sequential order of execution
  - Expressions and Statements
  - Iteration: how to iterate over a series of values
  - Selection: how to select between alternative actions
  - Function: how a particular sub-computation can be named and specified separately
  - To be able to perform more realistic computations, we will introduce the `vector` type to hold sequences of values.

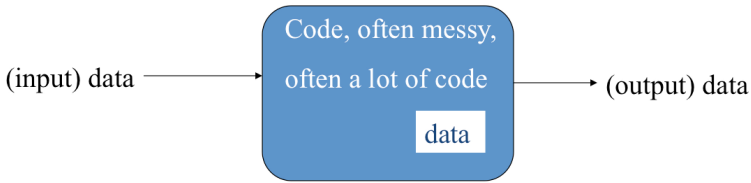
You already know most of this

Note:

- You know how to do arithmetic
  $d = a + b \cdot c$
- 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 we will see here 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
  - Input/output formats
  - Protocols
  - Data structures

Note the emphasis on structure and organization
You don't get good code just by writing a lot of statements

Language Features

- Each programming language feature exists to express a fundamental idea
  For example:
  - `+`: addition
  - `*`: multiplication
  - `if (expression) statement else statement;` selection
  - `while (expression) statement;` iteration
  - `f(x)` function/operation
  - ...
- We combine language features to create programs

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: `+, -, *, /`

Concise Operations

For many binary operators, there are (roughly) equivalent more concise operators

For example:

`a += c`	means	`a = a+c`
`a *= scale`	means	`a = a*scale`
`++a`	means	`a += 1` or `a = a+1`

Concise operators are generally better to use (clearer, express an idea more directly)

Statements

A statement is

- an expression followed by a semicolon, or
- a declaration, 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

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
```

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 << '\t' << square(i) << '\n';
    ++i ;       // increment i
  }
}
// (No, it wasn't actually written in C++.)
```

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
Increment 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 (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';
}
```

That is,

```
for (initialize; condition ; increment )
```

controlled statement

Note: what is `square(i)`?

Functions

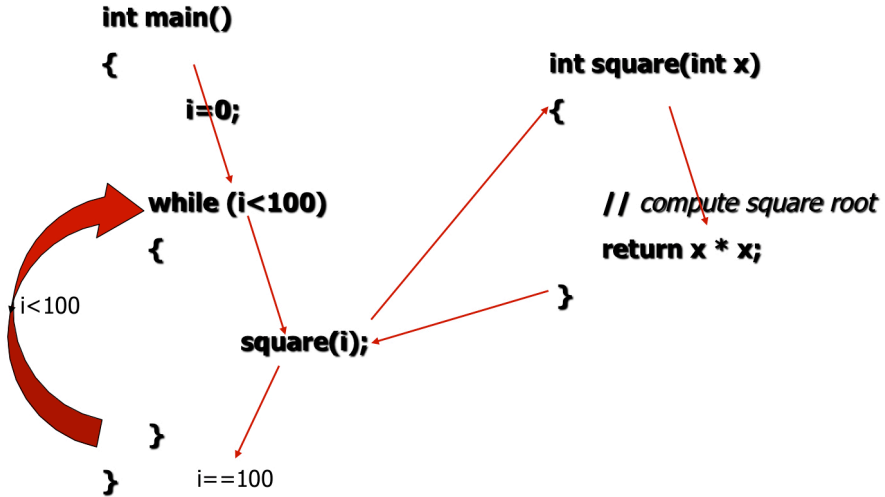
What was `square(i)`?

- A call of the function `square()`

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

- We define a function when we want to separate a computation because it
  - is logically separated
  - 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



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
}
```

Another Example

Earlier we looked at code to find the larger of two values. Here is a function that compares the two values and returns the larger value.

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

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

Data for Iteration – Vector

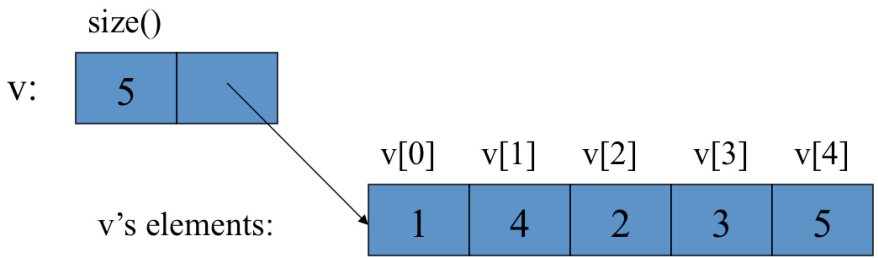
To do just about anything of interest, we need a collection of data to work on. We can store this data in a vector.
For example:

```
// read some temperatures into a vector:
int main()
{
  vector<double> temps; // declare a vector of type double to store temperatures
  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"
```

Vector

Vector is the most useful standard library data type

- a `vector<T>` holds a sequence of values of type `T`
- Think of a vector this way
  A vector named `v` contains 5 elements: `{1, 4, 2, 3, 5}`:

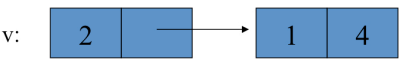


Vectors

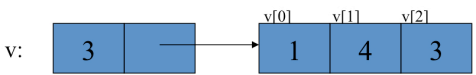
vector<int> v; // start off empty

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

v.push_back(4); // add an element with the value **4** at end (“the back”)



v.push_back(3); // add an element with the value **3** at end (“the back”)



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 in Fahrenheit, e.g. 64.6
  double temp;
  while (cin>>temp)  temps.push_back(temp); // read and put into vector

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

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

Traversing a Vector

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

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

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

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

// 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 ...
```

Combining Language Features

You can write many new programs by combining language features, built-in types, and user-defined types in new and interesting ways.

So far, we have:

- Variables and literals of types `bool`, `char`, `int`, `double`
- `vector`, `push_back()`, `[ ]` (subscripting)
- `!=`, `==`, `=`, `+`, `-`, `+=`, `<`, `&&`, `||`, `!`
- `max( )`, `sort( )`, `cin>>`, `cout<<`
- `if`, `for`, `while`

You can write a lot of different programs with these language features!
Let's try to use them in a slightly different way...

Example – Word List

```
// preliminaries left out

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

sort(words);                                 // 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.
*/
```

Example – 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);

for (int 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 – Word List

Eliminate Duplicates (cntd)


```
// 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);

vector<string> w2;
if (0<words.size()) {                    // note style { }
  w2.push_back(words[0]);
  for (int 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";
```

Algorithms

- We just used a simple algorithm
- An algorithm is (from Google search)

  "a logical arithmetical or computational procedure that, if correctly applied, ensures the solution of a problem.-- Harper Collins

  "a set of rules for solving a problem in a finite number of steps, as for finding the greatest common divisor.-- Random House

  "a detailed sequence of actions to perform or accomplish some task. Named after an Iranian mathematician, Al-Khawarizmi. Technically, an algorithm must reach a result after a finite number of steps. [...] The term is also used loosely for any sequence of actions (which may or may not terminate)." – Webster's

- We eliminated the duplicates by first sorting the vector (so that duplicates are adjacent), and then copying only strings that differ from their predecessor into another vector.

Ideal

Basic language features and libraries should be usable in essentially arbitrary combinations.

- We are not too far from that ideal.
- If a combination of features and types make sense, it will probably work.
- The compiler helps by rejecting some absurdities.

Outline

1. Data Types

2. Type safety

3. Computation