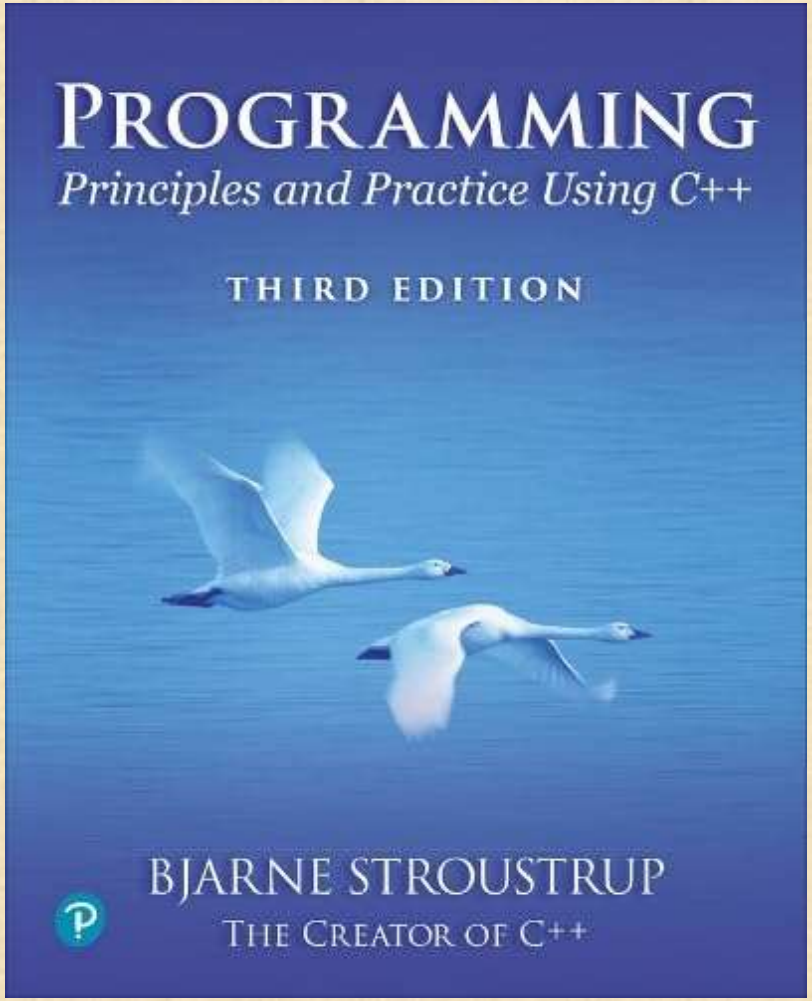
Chapter 3 - Computation



If it doesn´t have
to produce correct results,
I can make it arbitrarily fast.
– Gerald M. Weinberg

Abstract

- Today, we present the basics of computation. In particular, we discuss expressions, how to iterate over a series of values (“iteration”) and to select between two alternative actions (“selection”). We also show how a particular sub-computation can be named and specified separately as a function. Our primary concern is to express computations in ways that lead to correct and well-organized programs.
- To be able to perform more realistic computations, we introduce the vector type to hold sequences of values.

- Selection, Iteration, Function, Vector

Overview

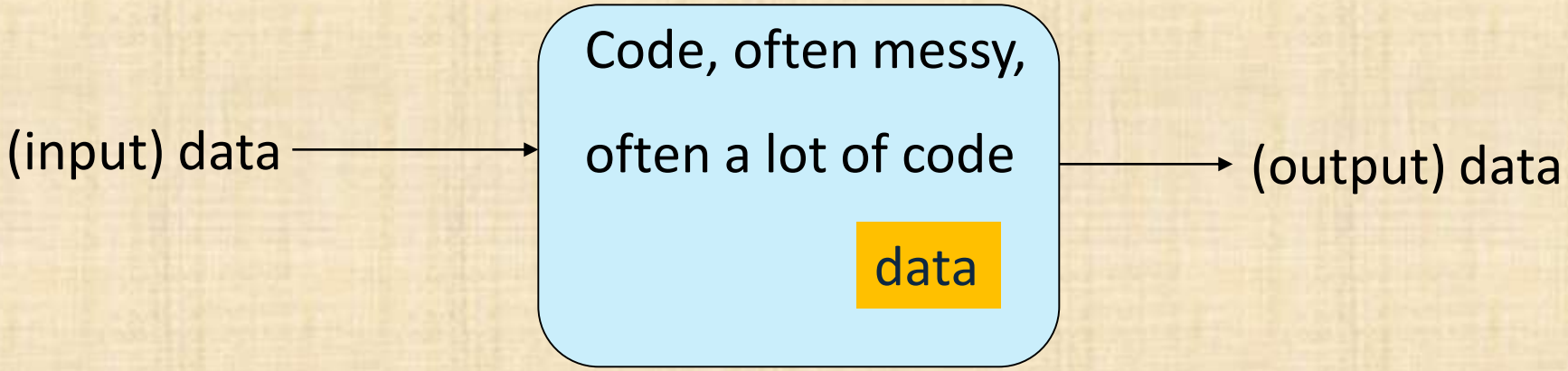
- Computation
  - What is computable? How best to compute it?
  - Abstractions, algorithms, heuristics, data structures
- Language constructs and ideas
  - Sequential order of execution
  - Expressions and Statements
  - Selection
  - Iteration
  - Functions
  - Vectors

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
  - 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
  - When writing code for the use of others, we must do it responsibly so that they can rely on it

Language features

- Each programming language feature exists to express a fundamental idea
  - For example
    - Addition **+**
    - multiplication *****
    - selection **if** (*expression*) *statement* **else** *statement ;*
    - iteration **while** (*expression*) *statement* **;**
    - function call **f(x);**
    - …
- 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.

Operators

- 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** (values **true** and **false**)
  - Equality operators: **= =** (equal), **!=** (not equal)
  - Relational operators: **<** (less than), **>** (greater than),
    <= (less than or equal), **>=** (greater than or equal)
- Logical operators: **&&** (and), **||** (or), **!** (not)
- Character type: **char** (values e.g., **'a'**, **'7'**, and **'@'**)
- Integer types: **short, int, long** (values e.g., **7**, **42,** and **12345678**)
  - arithmetic operators: **+**, **-**, *****, **/**, **%** (remainder)
- Floating-point types: e.g., **float, double** (values e.g., **12.45** and **1.234e3**)
  - arithmetic operators: **+, -, *, /**

Concise Operators

- 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** and **a = a+1**
  - "Concise operators" are generally better to use
    - they express ideas more directly

Constant expressions

- Constants (constant values) are important in most computations
- Name your important constants
  - **constexpr double pi = 3.14159;**
  - **pi = 7;** *// error: assignment to constant*
  - **double c = 2*pi*r;** *// OK: we just read pi; we don't try to change it*
- Naming constants
  - Makes code more understandable
  - Prevents accidental change of value
  - Use constexpr for values that must be known at compile time
    - **constexpr int the_answer = 42;**
  - Use const for values that must be initialized with a value that depends on input
    - **int val = 0;** *// an input variable*
    - **cin >> val;**
    - **const int length = val;** *// a value that mustn't change*

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 soon 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, England, 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)  {                                    // keep doing until i reaches 100
        cout << i << '\t' << square(i) << '\n';     // write out a square
        ++i ;                                       // increment i
}
```

Iteration (**for**-loop)

- Another iteration form: the **for**-loop
  - Place 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

- But 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 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

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**

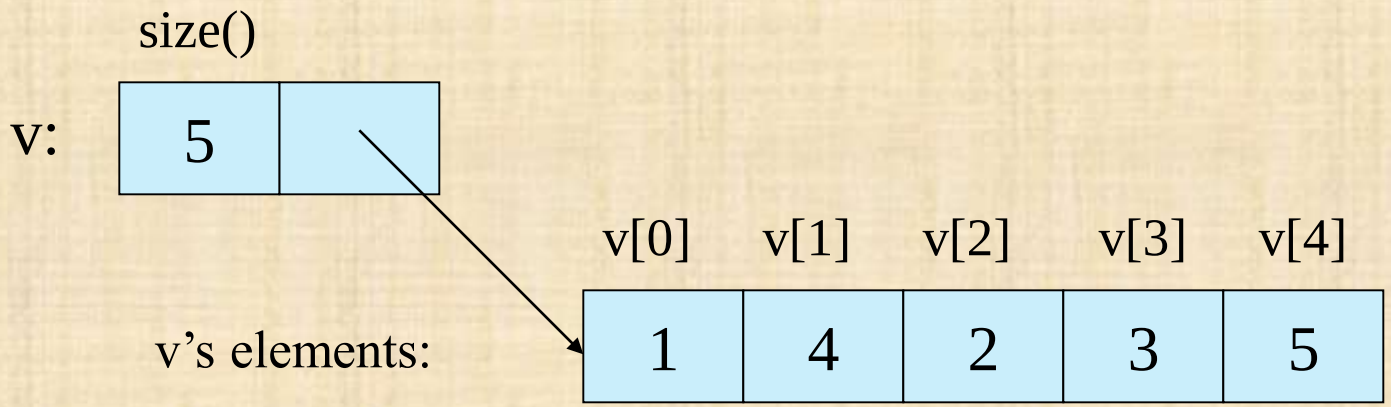
- To do just about anything of interest, we need a collection of data to work on. We can store this data in a **vector**. For example:

```
int main()
  // read some temperatures into a vector:
{
  vector<double> temps;       // declare a vector of type double to store temperatures - like 62.4
  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"
```

vector

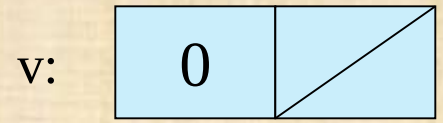
- **vector** is the most useful standard library data type
  - a **vector<T>** holds a sequence of values of type **T**
  - Think of a vector this way

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

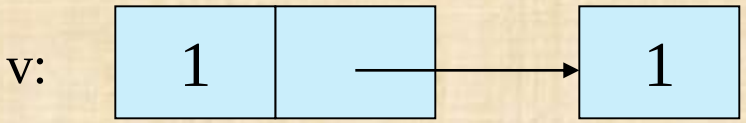


vector

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

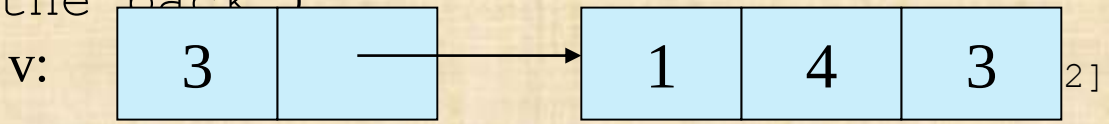



```
v.push_back(1);    // add an element with the value 1
```




```
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”)
```



2]

vector

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

```
int main()
  // compute mean (average) and median temperatures:
{
  vector<double> temps;                  // temperatures in Fahrenheit, e.g., 64.6
  double temp;
  while (cin>>temp)                      // read and put into vector
        temps.push_back(temp);

  double sum = 0;
  for (int i = 0; i< temps.size(); ++i)  // sums temperatures
       sum += temps[i];
  cout << "Mean temperature: " << sum/temps.size() << '\n';
  ranges::sort(temps);                   // from PPP.h
  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)                    // list all elements
  cout << v[i] << '\n';

// there is a simpler kind of loop for that (a range-for loop):

for (int x : v)                                      // for each x in v …
  cout << x << '\n';
```

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

```
// "boilerplate" left out

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

ranges::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.
*/
```

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";  )          // && means AND
      words.push_back(s);

 ranges::sort(words);

 for (int i=1; i<words.size(); ++i)
      if (i==0 || words[i-1]!=words[i])           // || means OR
           cout << words[i] << '\n';


//    there are many ways to avoid the duplicated words, many of them
 are messy (that's typical).
//    Our job as programmers is to choose a simple clean solution
```

Example (cont.) Eliminate Words!

```
// Eliminate the duplicate words by copying only unique words:
     vector<string> words;
for (string s; cin>>s && s!= "quit";  )
     words.push_back(s);
ranges::sort(words);
vector<string>w2;
if (0<words.size()) {
     w2.push_back(words[0]);
     for (int i=1; i<words.size(); ++i)
          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";
```

Algorithm

- 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

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.

The next lecture

- How to deal with errors