

Outline

1. Computation
2. Control Structures
3. Functions
4. Vector

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”

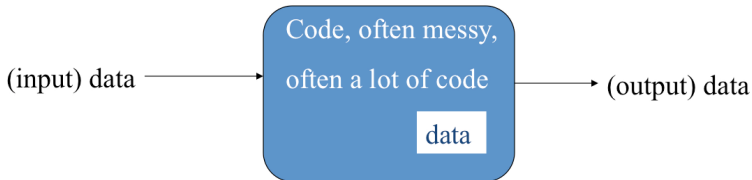
“do that 100 times”

- 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: `+`, `-`, `*`, `/`

Expressions - Precedences

Order of precedence from highest to lowest:

- Unary operators
- Multiplicative operators
- Additive operators
- Shift operators
- Relational operators
- Equality operators
- Bitwise operators
- Logical operators

Concise Operations

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

For example:

<code>a += c</code>	means	<code>a = a+c</code>
<code>a *= scale</code>	means	<code>a = a*scale</code>
<code>++a</code>	means	<code>a += 1</code> or <code>a = a+1</code>

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 ...

Outline

1. Computation
2. Control Structures
3. Functions
4. Vector

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: <code>i</code>
Initialize the control variable;	here: <code>int i = 0</code>
A termination criterion;	here: <code>if i<100</code> is false, terminate
Increment the control variable;	here: <code>++i</code>
Something to do for each iteration;	here: <code>cout <<</code>

```
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)`?

Outline

1. Computation
2. Control Structures
3. Functions
4. Vector

Functions

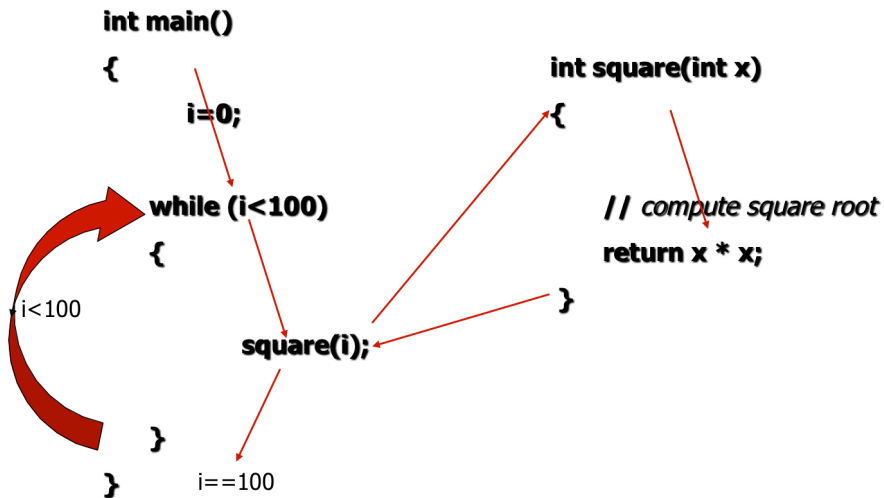
What was `square(i)`?

- 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
```

Outline

1. Computation
2. Control Structures
3. Functions
4. Vector

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"
```

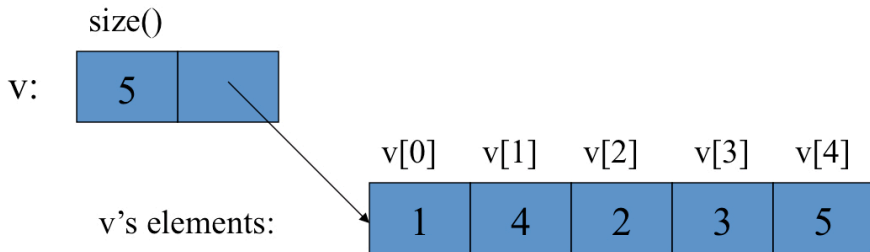
Vector

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



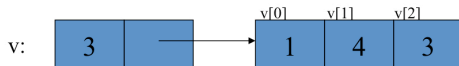
`v.push_back(1);` // add an element with the value 1



`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 'algorithm' or 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. Computation
2. Control Structures
3. Functions
4. Vector