Session 3

Program organization

Refactoring

Refactoring is re-arranging code (to improve re-use, maintainability, *etc*), without changing what it does. We shall use:

- Functions, with
  - parameters passed by value
  - parameters passed by `const` reference
  - parameters passed by non-`const` reference
- Using multiple source files.
  - source files and header files
  - dealing with repeated inclusion

Functions and parameter passing

Functions

Consider the program that computes medians and averages.

- Several pieces could be made into re-usable components:
  - read a vector
  - compute the median of a vector
  - compute the average of a vector
- We will split these out as separate functions.
- In C++, functions must be declared before use, so our new functions will go after the `using` declaration, but before `main()`.
- The median and average functions won't do any I/O: this makes them more generally useful.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    cout << "Please enter a series of numbers\n";

    // read numbers from the standard input
    // and store them in a vector
    vector<double> values;
    double x;
    while (cin >> x)
        values.push_back(x);

    // compute and output results
    auto n = values.size();
    cout << n << " numbers\n";
    if (n > 0) {
        // compute the average
        double sum = 0;
        using vec_size = vector<double>::size_type;
        for (vec_size i = 0; i < n; ++i)
            sum += values[i];
        cout << "average = " << sum/n << '\n';

        // sort the whole vector
        sort(values.begin(), values.end());

        // find the middle value
        auto middle = n/2;
        double median;
        if (n%2 == 1) // size is odd
            median = values[middle];
        else // size is even
            median = (values[middle-1] + values[middle])/2;
        cout << "median = " << median << '\n';
    }
    return 0;
}
```

Figure 3.1: Program computing average and median

Median of a vector

```
// the median of the values in a vector
// requires: v.size() > 0
double median(vector<double> v) {
    auto n = v.size();
    // sort the whole vector
    sort(v.begin(), v.end());
    auto middle = n/2;
    if (n%2 == 1) // size is odd
        return v[middle];
    else // size is even
        return (v[middle-1] + v[middle])/2;
}
```

Checking the precondition

- The **median** function promises to return the median of a vector, but only if the vector is non-empty.
- This is the *precondition* on the function.
- If the precondition is not satisfied, the function can do anything. Here we throw an exception **std::domain_error** (from the **<stdexcept>** header).

```
// the median of the values in a vector
// requires: v.size() > 0
double median(vector<double> v) {
    auto n = v.size();
    if (n == 0)
        throw domain_error("median of an empty vector");
    // ...
}
```

Parameter passed by value

```
double median(vector<double> v) {
    sort(v.begin(), v.end()); // rearranges v
    // ...
}

int main() {
    vector<double> values;
    // ...
    cout << "median = " << median(values) << '\n';
    // ... (values unchanged)
}
```

- The parameter **v** is a new vector, initialized as a *copy* of **values**.
- Changes to **v** (*e.g.* **sort**) do not affect **values**.

Avoiding the copy

- In **median**, it makes sense to operate on a copy of the vector, because the function changes the vector (by sorting it), and the rest of the program might require the values in the original order.
- But copying large arguments (*e.g.* vectors) is expensive, and often we want to avoid it.
- Also, we might want to return data through the argument, and we can't do that if we're operating on a copy.
- A solution (Fortran, Pascal, C++, C#, *etc*) is *reference parameters*.
- In C++, reference parameters are marked with **&**.
- If the function is not going to change the argument, it can be marked as **const**.

☞ ***Advice (Effective C++ Item 20)***
Prefer pass-by-reference-to-const to pass-by-value (except for primitive types).

Average of a vector

When computing the average, we can use the original vector (no copying), and not change it:

```
// the average of the values in a vector
// requires: v.size() > 0
double average(const vector<double> &v) {
    auto n = v.size();
    if (n == 0)
        throw domain_error("average of an empty vector");
    double sum = 0;
    using vec_size = vector<double>::size_type;
    for (vec_size i = 0; i < n; ++i)
        sum += v[i];
    return sum / v.size();
}
```

Parameter passed by const reference

```
double average(const vector<double> &v) {
    // ... (doesn't change v)
}

int main() {
    vector<double> values;
    // ...
    cout << "average = " << average(values) << '\n';
```

```
    // ... (values unchanged)
}
```

- The parameter **v** is another name for **values** (no copying).
- **const** promises that the function will not change **v**, and the compiler checks this.

Other uses of const

The **const** specifier can also be used

- to declare global constants, *e.g.*

```
const int days_per_week = 7;
```

(This is much better than using C-style **#define**s.)

- to declare parameters and local variables with fixed values, *e.g.*

```
    const auto middle = v.size()/2;
```

- on member functions in classes (session 6)

☞ ***Advice (Effective C++ Item 3)***
Use **const** whenever possible.

Another example of const

For efficiency, we might save **v.size()** in a variable **n**, which we then do not change:

```
// the average of the values in a vector
// requires: v.size() > 0
double average(const vector<double> &v) {
    const auto n = v.size();
    if (n == 0)
        throw domain_error("average of an empty vector");
    double sum = 0;
    using vec_size = vector<double>::size_type;
    for (vec_size i = 0; i < n; ++i)
        sum += v[i];
    return sum / n;
}
```

A function that modifies a parameter

Because a reference parameter is an alias for the argument, changing it modifies the argument directly, e.g.

```
// add one to each element of a vector
void add_one(vector<double> &v) {
    using vec_size = vector<double>::size_type;
    for (vec_size i = 0; i < v.size(); ++i)
        v[i]++;
}

int main() {
    vector<double> values;
    // ...
    add_one(values);  // changes values
}
```

Other uses of references

We can also use references when we want to re-use the same memory location (rather than re-using a value), *e.g.*:

```
    vector<double> v;
    // ...
    double &x = v[12];
    x = 1 + 2*x;
```

Here **x** is not a new variable, but an alias for **v[12]**. Everything that is done to **x** is actually done to **v[12]**.

Caution

- C++ references have no counterpart in Java.
- Java references are similar to C++ pointers (session 9).

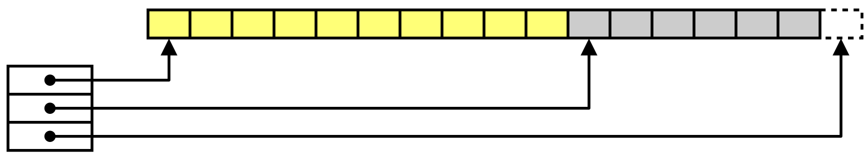
Function to read a vector of numbers

This function also operates on the original input stream (no copying), but changes it so the parameter **in** is passed by reference, but without **const**:

```
// read numbers from an input stream
// and return them in a vector
vector<double> read_vector(istream &in) {
    vector<double> v;
    double x;
    while (in >> x)
        v.push_back(x);
    return v;
}
```

Why is returning a vector cheap?

Recall the implementation of a vector (details may vary):



- Normally if you assign a vector, both parts get copied.
- But here the vector is in a local variable, which is about to disappear, so the compiler knows that only the record needs to be copied, not the array. (The vector is *moved* rather than copied.)
- A particularly smart compiler might even avoid copying the record.

Main program

```
int main() {
    cout << "Please enter a series of numbers\n";

    // read numbers from the standard input
    // and store them in a vector
    const auto values = read_vector(cin);

    // compute and output results
    const auto n = values.size();
    cout << n << " numbers\n";
    if (n > 0) {
        cout << "average = " << average(values) << '\n';
        cout << "median = " << median(values) << '\n';
    }
    return 0;
}
```

Three kinds of parameters

- parameter passed by value:

  ```
  double median(vector<double> v)
  ```

  The parameter is a *new variable*, initialized as a copy of the actual parameter.

- parameter passed by `const` reference:

  ```
  double average(const vector<double> &v)
  ```

  The parameter is an *alias* for the actual parameter, but the function will not change it.

- parameters passed by non-`const` reference:

```
vector<double> read_vector(istream &in)
```

  The parameter is an *alias* for the actual parameter, and the function may change it.

Return by reference

- Normally a function returns its result by value (so it is copied).
- It is possible to define a function that returns by reference, *e.g.* the `<iostream>` library function to a read a line:

```
istream& getline(istream& in, string &s)
```

  This returns its parameter `in`.

- Then the function call is an alias for whatever is returned, e.g. in

```
    string s;
    while (getline(cin, s))
        cout << s.size() << '\t' << s << '\n';
```

- You can't return a temporary or local variable by reference (because they will be gone when the function returns).

Multiple source files

Separate compilation

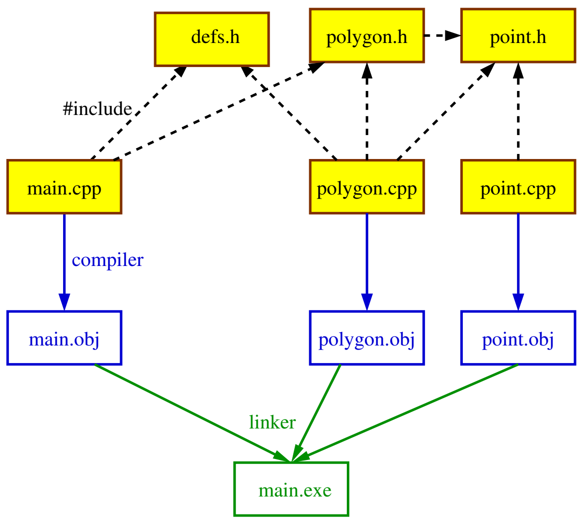
- For large projects, it is common to split programs into several files, which can be compiled separately as required.
- In addition to the source files (`.cpp` or `.cc`, *etc*), we put declarations in *header files* (`.h`).
- Typically, *foo*`.h` contains declarations for things defined in *foo*`.cpp`.
- source files (and other header files) access these definitions using `#include` lines, which insert the text of those header files at compile time:

```
#include "foo.h"
```

  Note the different syntax from system headers.

- In particular, *foo*`.cpp` should include *foo*`.h` (so the compiler checks that the definitions match the declarations.

Typical program structure



Rebuilding a program after edits

- If a source file has been changed, it must be recompiled, generating the corresponding object file.
- If a header file has been changed, all the source files that include it, directly or indirectly, must be recompiled.
- If any source files have been recompiled, the object files must be linked to regenerate the executable program.

☞ ***Advice (Effective C++ Item 31)***
Minimize compilation dependencies between files.

- Minimize the number of header files included in other files.
- This also means putting the minimum in each header file.

Splitting our program

We shall re-arrange our program to extract an embryonic statistics library. The new program will consist of the following files:

stats.h A header file, with declarations of the functions `median` and `average` (just their parameters and return types)

stats.cpp A source file that starts with `#include "stats.h"` and gives the full definitions of the functions `median` and `average`.

main.cpp A source file that starts with `#include "stats.h"` and gives the full definitions of the functions `read_vector` and `main`.

The auxiliary source file `stats.cpp`

```
#include "stats.h"
#include <vector>
#include <algorithm>
#include <stdexcept>

using namespace std;

// the median of the values in a vector
// requires: v.size() > 0
double median(vector<double> v) {
    const auto n = v.size();
```

and so on (rest of **`median`** and all of **`average`**).

Placement of `#includes`

```
#include "stats.h"
#include <vector>
#include <algorithm>
#include <stdexcept>

using namespace std;
```

- We include our own header files before system headers, which come before any **`using`** declaration. That ensures that our header files don't assume anything had been included.
- We only include the system headers we need. (In this case, we don't need **`<iostream>`**.)

The header file `stats.h`

```
#include <vector>

// the median of the values in a vector
// requires: v.size() > 0
double median(std::vector<double> v);

// the average of the values in a vector
// requires: v.size() > 0
double average(const std::vector<double> &v);
```

- Never put **`using`** declarations in header files – that would force names on clients.
- This means all standard names must be fully qualified.
- Give only function signatures, followed by '**`;`**', not the bodies.
- For the types, we only need the **`<vector>`** system header.

The main source file `main.cpp`

- The file begins with the headers, following the same principles as in **`stats.cpp`**:

```
#include "stats.h"
#include <iostream>
#include <vector>

using namespace std;
```

- This is followed by the functions **`read_vector`** and **`main`**.
- The **`main`** function uses **`median`** and **`average`**, which are declared in **`stats.h`**.

Handling repeated inclusion

- Sometimes a header file can be included twice, *e.g.*:

```
#include "line.h"    // includes point.h
#include "polygon.h" // also includes point.h
```

- With just function declarations this is harmless, but with later features it will cause compilation errors.
- We need to ensure that the second inclusion of any header file does nothing.
- A widely-implemented, but non-standard method is to place this line at the top of the header file:

```
#pragma once
```

- The fully portable method uses *include guards*.

`stats.h` with include guards

```
#ifndef GUARD_stats_h
#define GUARD_stats_h

#include <vector>

// the median of the values in a vector
// requires: v.size() > 0
double median(std::vector<double> v);

// the average of the values in a vector
// requires: v.size() > 0
double average(const std::vector<double> &v);

#endif
```

Include guards

```
#ifndef GUARD_stats_h
#define GUARD_stats_h

// body of header file

#endif
```

- The **#** lines are features from the C preprocessor that shouldn't be used for anything else.
- The first time the file is included, **GUARD_stats_h** is not defined, so the whole is included, including the second line, which defines **GUARD_stats_h**.
- The next time the file is included, **GUARD_stats_h** is defined, so the rest of the file is skipped.

```
#include "stats.h"
#include <vector>
#include <algorithm>
#include <stdexcept>

using namespace std;

// the median of the values in a vector
// requires: v.size() > 0
double median(vector<double> v) {
    const auto n = v.size();
    if (n == 0)
        throw domain_error("median of an empty vector");
    // sort the whole vector
    sort(v.begin(), v.end());
    const auto middle = n/2;
    if (n%2 == 1) // size is odd
        return v[middle];
    else // size is even
        return (v[middle-1] + v[middle])/2;
}

// the average of the values in a vector
// requires: v.size() > 0
double average(const vector<double> &v) {
    const auto n = v.size();
    if (n == 0)
        throw domain_error("average of an empty vector");
    double sum = 0;
    using vec_size = vector<double>::size_type;
    for (vec_size i = 0; i < n; ++i)
        sum += v[i];
    return sum / n;
}
```

Figure 3.2: `stats.cpp`

```
#include "stats.h"
#include <iostream>
#include <vector>

using namespace std;

// read numbers from an input stream
// and return them in a vector
vector<double> read_vector(istream &in) {
    vector<double> v;
    double x;
    while (in >> x)
        v.push_back(x);
    return v;
}

int main() {
    cout << "Please enter a series of numbers\n";

    // read numbers from the standard input
    // and store them in a vector
    const auto values = read_vector(cin);

    // compute and output results
    const auto n = values.size();
    cout << n << " numbers\n";
    if (n > 0) {
        cout << "average = " << average(values) << '\n';
        cout << "median = " << median(values) << '\n';
    }
    return 0;
}
```

Figure 3.3: `main.cpp`

Exercises

Most of these questions build on the program developed in the lecture. In doing these questions, think carefully about when **&** and **const** would be appropriate.

1. Add a **score** function, based on last week's exercise 2, to **stats.cpp**, make appropriate changes to **stats.h**, and use the function in **main.cpp**.

2. Write and test functions to find the maximum and minimum elements of a vector of doubles.

3. Write and test a function to write out a vector of doubles to a given output stream (**ostream**).

4. Write and test a function that takes a vector of doubles and reverses its contents. You can use the library function **swap** from the **<utility>** system header:

   ```
   swap(x, y);
   ```

   swaps the contents of **x** and **y**. Test your function using the function from the previous part.

5. Move **read_vector** and the functions from the previous three parts into a separate source file of vector utilities, with an appropriate header file and any other changes.

6. Write and test a function **longest** that takes a vector of strings and returns the longest. Try to avoid copying vectors or strings.