# A review of Bjarne's small book

Leanne Dong

September 19, 2020

## 1 The Basics

- Programs
- Functions
- Types, Variables and Arithmetic
- Scope and Lifetime
- Constants
- Pointers, arrays and References
- Tests
- Mapping to Hardware

## 2 User-defined type

The set of C++ build-in types[1] and operations is rich, but deliberately low-level. The directly and efficiently reflect the capabilities of conventional computer hardware. However, they don't provide programmers with high-level facilities to write advanced applications easily. To overcome this, C++ augments the built-in types and operations with a sophisticated set of abstraction mechanisms out of which programmers can build such high-level facilities.

### 2.0.1 Strutures

The first step is building a new type is to putting elements we need into a data structure, a struct:

```
struct Vector {
    int sz;     // number of elements
    double* elem; // pointer to elements
};
```

The first version of Vector consists of an int and a double*. A variable of type Vector can be defined as

```
Vector v;
```

However, by itself that is not of much use because v's elem pointer doesn't point to anything. For it to be useful, we must give v some elements to point to. For instance, construct a Vector like

```
void vector_init(Vector& v, int s)
{
    v.elem = new double[s];
    v.sz = s;
}
```

[1] types that can be built from the fundamental types, the const modifier, and the declarator operator

That is, v's elem member gets a pointer produced by the new operator, and v's sz member gets the number of elements. The & in Vector& indicates that we pass v by non-const ref, so that vector_init() can modify the vector passed to it.

A simple implementation of Vector could be

```
double read_and_sum(int s)
// read s integers from cin and return their sum, s is //assumed to be positive
{
    Vector v;
    vector_init(v, s); // allocate s elements for v
    for (int i = 0; i!=s; ++i)
    {
        std::cin >> v.elem[i]; // read into elements
    }
    double sum = 0;
    for (int i = 0; i!=s; ++i)
    {
        sum+=v.elem[i];
    }
    std::cout << sum << " \n";
    return sum;
}

int main()
{
    read_and_sum(10);
}
```

There was a long to go from above to the elegant std::vector.

Comiler Explorer: `https://godbolt.org/z/vjEMeP`

We use .(dot) to access struct members through a name (and through a reference) and -> to access struct members through a pointer. For instance

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz; // access through name
    int i2 = rv.sz; // access through reference
    int i3 = pv->sz; // access through pointer
}
```

Now another example from Herb's talk.

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

struct User {
    std::string name;
    int age;
};

std::vector<User> users = { {"Cat", 3}, {"Fish", 5} };

int main()
{
    auto sort_by_age = [] (auto& lhs, auto& rhs)
    {
        return lhs.age < rhs.age;
    };
    std::sort(users.begin(), users.end(), sort_by_age);
}
```

Compiler explorer: `https://godbolt.org/z/noea68`

### 2.0.2 Classes

### 2.0.3 Unions

### 2.0.4 Unions

### 2.0.5 Enumerations