Data Structures Notes

Taught by John Nevard

Daniel Kim, Sameer Pai

October 3, 2019

These are a collection of notes (typed live) of all classes of Data Structures taught by John Nevard at the Bergen County Academies.

Contents

§1 Review of C++

§1.1 Reference

In Java, everything is a **reference**, while C++ is mainly **pass-by-value**. For example, consider the following snippet of Java code:

```
Array <Int> x = new Array<Int>(17);
Array <Int> y = x;
```

Here, `x` and `y` refer to the same block of memory (that is, the array of integers of length 17). Meanwhile, in C++ , if we have:

```
vector<int> x(17);
vector<int> y = x;
```

Then, `x` and `y` refer to separate arrays of integers of length 17 respectively.

§1.2 Operators

In C++ , we have the following operators, in order of precedence:

1. `::` (the scope resolution operator), `::`$_1$

   For example, this operator allows us to access static variables of classes:

   ```
   class X {
       static int a;
   };
   int y = x::a
   ```

   Henceforth, we use the subscript $_1$ to refer to the same operator in a **unary** context, meaning that it requires only one argument.

2. `[]`, `()`

3. `*`$_1$, `&`$_1$, `!`, `~`, `+`$_1$, `-`$_1$, `new`, `delete`, `delete[]` (which is used for arrays of objects), `++`, `--`

4. `+`,`-`

5. `==`, `!=`

6. `>`, `>=`, `<`, `<=`

7. `&`

8. `^`

9. `|`

10. `&&`

11. `||`

12. `?  :` (**ternary choice**)

    For example, consider the following code:

    ```
    double y;
    if (x > 0) {
        y = sqrt(x);
    } else {
        y = sqrt(-x);
    }
    ```

    We can significantly condense this using a ternary opertor:

    ```
    double y = x > 0 ? sqrt(x) : sqrt(-x);
    ```

13. `=`, `+=`, `-=`, `*=`, `/=`, `%=`

    For example,

    ```
    a += b
    a = a + b
    ```

    These two lines do the exact same thing.

§1.3 Short Circuit

Consider the following condition:

```
if (i >= 0 && a[i] < 0) {
```

If the computer evaluates `i >= 0` to be false, then the whole condition will be false no matter what (because of logical AND). Thus, the computer saves time by just ignoring the rest of the statement (namely `a[i] < 0`) and continues on with the program. In this fashion, we can avoid accessing an array out of bounds.

§1.4 Bit Operators

An `int` is usually 4 bytes. Its formal keyword is `int_32`. A `long` is usually 8 bytes. Its formal keyword is `int_64`. An `unsigned int` also has 8 bytes, and its formal keyword is `uint_64`.

§1.5 Trivial I/O

To provide us with basic input and output, we first include the line

```
#include <iostream>
```

If you ONLY include this line, then you will have to write

```
std::cout << "Hello World!\n";
```

to print `Hello World!`. Otherwise, if we start with:

```
#include <iostream>
using namespace std;
```

Then, we can simply write:

```
cout << "Hello World!\n";
```

to do the same thing. However, it is recommended not to use `using namespace std`. Instead, one should write:

```
#include <iostream>
using std::cout;
```

using the scope resolution operator. This enables us to do the same as before.

Here is an example of how `cout` allows us to print concatenations of variables of various data types:

```cpp
int x = 17;
double y = 3.14;
string s = "Huh?";
cout << x << ' ' << y << ' ' << s << '\n';
```

This prints: `17 3.14 Huh?`.

Furthermore, we can conveniently read input using `cin`. The following code takes in two integers and a string provided by the user in that order:

```cpp
int x, y;
string s;
cin >> x >> y >> s;
```

§1.6 "Hello World"

Here is the classic "Hello World" program for C++ :

```cpp
#include <iostream>
using std::cout;
int main() {
    cout << "Hello World\n";
    return 0;
}
```

§1.7 Preprocessor Directive

Consider the following block of code:

```cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
    cout << "Enter x & y: ";
    int x;
    double y;
    cin >> x >> y;
    cout << "x + y = " << x+y << '\n'; // Use endl if you want to flush the output
    return 0;
}
```

There's also *spooky* I/O:

```cpp
#include <cstdio> // C I/O
// "stdio.h" is the same as "cstdio"
int main() {
    // More efficient than cout
    printf("HI!");
    int x;
    scanf("%d", &x);
    return 0;
}
```

Now, the `#` in `#include <iostream>` is called a **preprocessor directive**. There are many different possible preprocessor instructions that can be used after the `#`, but here the "include" statement tells the compiler to add the functions and classes another file to the code. Namely, `iostream` is called a **header file**.

We can also include custom header files (not in the standard library) by using double quotes, i.e. `#include "my_cool_stuff.h"`.

To prevent redefinitions of variables and functions, we use something called a **include guard**. How this works is in each header file we use a "preprocessor if statement" to ensure the file has not been included in the past.

```
#ifndef __my_cool_stuff__ // Only #include once
#define __my_cool_stuff__
// code here
#endif
```

§1.8 Variables

Another important keyword is `const`. This keyword means that the variable we declare cannot change, ever. If we were to write:

```
const double PI = 3.14159;
PI = 3; // Oops!
```

This would fail, because we already said that PI must be 3.14159. A good practice in programming is to use `const` whenever possible.

A difference between `#define` and `const` is memory usage: the preprocessor will generally use less memory, but the peace of mind granted by `const` more than makes up for this.

§1.9 Function Prototypes

Again, we point to another fundamental difference between Java and C++ : objects. In C++ , we can declare a function before explicitly writing the code for it. The way we do it is a **function prototype**.

```
int f (double x); // Prototypes

// Rest of code

y = f(3.7);
```

Even though the compiler does not know exactly what `f` does, it knows what its input and return types are from the prototype, so it can still use `f`, even if it is defined in another file.

In most cases, you can find a way to order your functions so that prototypes are not needed. However, if you have a "loop," then you need to write some prototypes. Consider the example:

```
int g(); // Necessary
int f(); // Optional

int f() {
   g();
}

int g() {
   f();
}
```

Here, both functions call each other, so in order to use a function before it is defined, we need to utilize at least one prototype.

§1.10 Swap 2 Values

There are many ways to swap two values in C++ . One is the `swap` function included in `<algorithm>`:

```cpp
#include <algorithm>
using std::swap;

// Rest of code

swap(x,y);
```

But how was `swap` even implemented? Let's first try (but fail to do so):

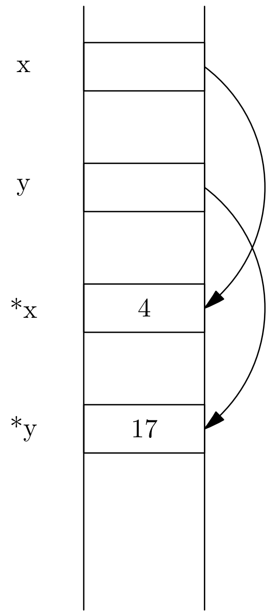
```cpp
// wrong code
void swap (int x, int y) { // For now, assume we are swapping int variables
    int t = x;
    x = y;
    y = t;
}
```

The problem is that `t`, `x`, `y` are **local variables**. C++ is **pass-by-value**, meaning that any variables passed in to a function have their values copied to another location in memory before running the function. So, any changes to `x` and `y` are not actual changes to the original variables.

The problem with memory might encourage us to use pointers instead (the unary asterisk declares that `x` is a pointer to an integer in memory):

```cpp
void swap (int *x, int *y) {
    int t = *x; // dereference the pointer
    *x = *y; // Replace contents of address x by contents of address y
    *y = t; //
}

int u = 3, v = 17;
swap(&u, &v); // Unary ampersand takes the memory address of the variable
```



Although this works, it's annoying to write `&x` and `&y` instead of plain `x` and `y`.
Thus, our last attempt uses references.

```cpp
void swap(int& x, int& y) { // int& is a reference to int (aliases)
    int t = x;
    x = y;
    y = t;
}
int u = 3, v = 17;
swap(u,v); // This will work!
```

§1.11 Classes

Now, we discuss the `class` in C++ . Consider the following example of code:

```cpp
class BUC {
    // Code here
};

void f(BUC b1, BUC b2); // prototype
BUC buc1, buc2;

// Wasteful code
f(buc1, buc2); // Copies buc1 to b1, and buc2 to b2
```

However, we can use references to avoid copying these values:

```cpp
class BUC {
    // Code here
};

void f(BUC& b1, BUC& b2); // prototype
BUC buc1, buc2;

f(buc1, buc2);
```

However, let's say that the operations `+`, `-` were defined for our class `BUC`. Then,

```cpp
f(b1+b2, b1-b2);
```

would not compile. First, the compiler would look up the code for `+`, `-` for the class `BUC`, and then find the values for `b1+b2` and `b1-b2`, and store them in respective places in memory. But if we take the references of these values, the compiler would not know where to look for in memory.

In fact, `b1+b2` and `b1-b2` are not **Lvalues**, meaning that they are not things that can be assigned to. Some examples of Lvalues include:

```cpp
x = ...
*x = ...
x[1] = ...
*x[i+j] = ...
i + j = k \\ Not valid
```

In essence, an Lvalue is a thing that can appear on the left side of an assignment `=`.

So how do we fix a function to where we can pass in objects that are *not* Lvalues? The way we do this is the use of the `const` keyword:

```cpp
void f(const BUC& b1, const BUC& b2) {
    cout << b1;
    cout << b2;
}
```

The `const` keyword tells the compiler that we do not intend to change the two BUCs. This means we can pass in non-Lvalues without worrying about a compiler error. The restriction is that we cannot change the values in the function.

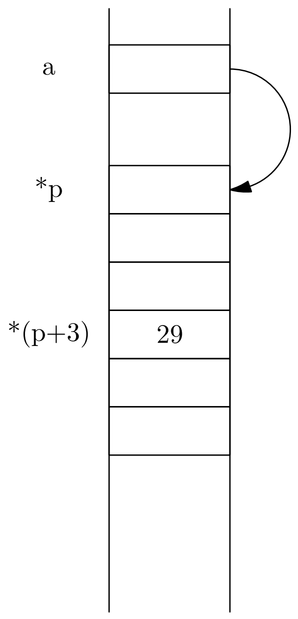
§1.12 Arrays

How do we declare an array in C++ ? We can use the following code:

```cpp
int a[17];
int *p = a;
```

In this code, when we declare the array `a`, it is necessary that the *size specifier* inside the brackets is a constant expression.

When we declare a pointer and assign it to the array `a` of size 17, the pointer will point to the first element of the array. However, if we then take `p+3`, we are adding 3 times the size of `int` to the pointer, so `p+3` will now point to the fourth element in `a`. In terms of memory addresses, if `p` $= 1000$, then `p+3` $= 1012$.



If we wanted to assign the fourth element to 29, then the following lines would all do the same thing:

```cpp
int a[17];
int *p = a;
*(p+3) = 29;
p[3] = 29;
a[3] = 29;
3[a] = 29
```

Furthermore, we note that `f(int a[])` and `f(int *a)` are equivalent. However, passing this argument alone leaves us with a major flaw:

```cpp
f(int a[]) {
   a[5] = 53; // No clue if enough space!
}
```

We don't know if the array `a` even has a fifth element available. Thus, we usually specialize the number of elements in `a`, as such:

```cpp
f(int a[], int na) {
   // Rest of code
}
```

Another way to define an array is using the **new** keyword. For example:

```cpp
int *a = new int[17];


delete[] a; // no automatic garbage collection
```

The `new` keyword has the advantage that your allocated memory will never go out of scope: it will remain there until you `delete` it. Also of note is that whenever you use **heap allocation**, the memory you allocate has a *header* which is not part of the data.

Using `new` comes with the caveat of having to garbage collect your own variables as well. Unlike Java, C++ does not automatically garbage collect (the onus is on you).

```cpp
class Dog {
    // Insert dog code
};

Dog *sirius = new Dog();
Dog *pack = new Dog[17];
// Do stuff
delete sirius;
delete[] pack;
```

It is good practice to never use `new` outside of classes.

§1.13 `vector`

However, one data structure usually works much better than just a normal array: the `vector`. To be able to start using it, make sure you include the line:

```cpp
#include <vector>
using std::vector;

int main() {
    vector<int> a(17); // Don't do a[17]! That would make an array of 17 vectors!
    vector<int> a(8, 5) // 8 elements initialized to 5.
    a[3] = 42;
    a[17] = 100; // error!
}
```

When the vector goes out of scope, its destructor is called automatically, and each of the allocated members of the array are deleted automatically. So, if we use classes correctly, we do not have to worry about memory leaks.

To make our lives easier, we can make a shortcut for `vector<int>`. We can also declare multi-dimensional vectors. The following code demonstrates these:

```cpp
using VI = vector<int> // or, typedef vector<int> VI;
using VD = vector<double>;

vector<VD> m(100, VD(100)); // 100 x 100 matrix of doubles, all initiated to 0

m[3][17] = 3.14159;

double determinant(const vector<VD>& m) {
    double det;
    vector<VD> m_copy = m;
    // stuff that changes m_copy, not m
    return det;
}
```

Alternatively, instead of declaring `m_copy`, we could write

```cpp
double determinant(vector<VD> m) {
    double det;
    // stuff that changes m
    return det;
}
```

but that would be bad practice since we are modifying the argument that is passed in. Certainly, if we write:

```cpp
double determinant(const vector<VD> m) {
    double det;
    vector<VD> m_copy = m;
    // stuff that changes m_copy, not m
    return det;
}
```

This code is wasteful, since not passing by reference already makes a copy of `m`.

§1.13.1 Useful member methods in `vector`

```cpp
using VD = vector<double>;
int main() {
    VD a(17);
    cout << a.size(); // outputs 17
    a.clear(); // a is now empty
    cout << a.size(); // 0
    a.resize(1000); // changes a's size to 1000

    VD b;
    bool done = false;
    while(!done) {
        double x;
        cin >> x;
        done = (x == PI);
        b.push_back(x); // adds x to the end of b
    }

}
```

For `a.resize(1000);`, this will leave the first thousand (or less) elements in vector `a` alone, while the data beyond that will be destroyed.

But how do `push_back` and `resize` work? In each `vector` object, there is an underlying array of some size. If there is still room in the underlying array, we simply add the element to the array.

However, if not, we *reallocate* all the elements into an array of twice the size, then add the necessary elements.

If we calculate (which we might do later), we will find that this only takes twice as much time as a static array, which is as good as we can get.

§1.14 Classes in Java and C++

As Java is an object-oriented language, it supports inheritance:

```cpp
// Inheritance
class Dog : public Mammal {
}
```

Suppose we write the following class in C++ (note that this is a *header file*):

```cpp
// File: Dog.h
using std::string;

class Dog {
    public:
    // Constructor
    Dog(const string& name,
        const string& breed = "mutt", // default arg
        const int& age = 0); // default arg

    // Destructor
    ~Dog(); // no arguments

    private:
    string name;
    string breed;
    int age;
    const int id;
};
```

The differences from Java are that we need a destructor (which has no arguments) that de-allocates all memory. We also do not implement the methods in the class: we are allowed to do it in another file (Here, we will call it Dog.cpp). In general, header files only declare methods, and they are implemented in a regular C++ file.

Now we demonstrate the implementation of class `Dog`:

```
// File: Dog.cpp

// Definition
Dog::Dog(const string& name,
         const string& breed,
         int age,
         int id) // cannot put default args in definition
: name(name), // first "name" refers to "name" in private attributes, second "name"
    refers to the string passed in as arg
  breed(breed),
  age(age),
  id(17) // ctor initializers
 {
    // Rest of code
 }

Dog:: ~Dog() { } // Always have destructor even if it's empty
```

§1.15 Some Diagnostic Review

Horner's method for evaluating polynomials with n multiplications:

```
double eval(double x, double a[], int n) {
    double y = a[--n];
    while (--n >= 0)
        y = a[n] + x * y;
    return y;
}
```

Merge two sorted arrays:

```
void merge(double c[], double a[], double b[], int na, int nb) {
    int i = 0, j = 0, k = 0;
    while (i < na && j < nb) {
        c[k++] = (a[i] <= b[j] ? a[i++] : b[j++]);
    }
    while(i < na)
        c[k++] = a[i++];
    while(j < nb)
        c[k++] = b[j++];
}
```

$$2^{2^{n+1}}, 2^{2^n}, n!, e^n, 2^n, n^{\log\log n}, n^3, n\log n, \sqrt{2}^{\log n} = \sqrt{n}, \sqrt{\log n}, \log\log n, n^{\frac{1}{\log n}} = 2$$

§1.16 Operator Overloading

Consider making a class R3, consisting of three-dimensional points in $\mathbb{R}^3$. We want to be able to use the points in convenient ways, like this:

```
R3 p (1,2,3), q(2, 5, 6);
cout << 2*p + 3*q; // Want to be able to do this with impunity
// Desired console output: (8, 19, 24)
```

So can we "modify" addition and cout to make this work? Consider the following class implementation:

```
class R3 {
public:
    R3(double x = 0, double y = 0, double z = 0): x(x), y(y), z(z) {} // Stroustrup
        says no!

    // Add two R3 class members together
    R3 operator+(const R3& p) const { // Compiler chooses this when invoked p is const
        return R3(x+y.x, y+p.y, z+p.z);
    }
    R3 operator+(const R3& p) { // Otherwise, this one
    }

    // Multiply an R3 class member by scalar
    friend R3 operator*(double t, const R3& p) // Not a member of the class; the double
        t is the "family friend"
    {
        return R3(t*p.x, t*p.y, t*p.z);
    }

    // Allow console to print an R3 class member in some format
    friend ostream& operator<<(ostream& os, const R3& p);


    ~R3();

private:
    double x, y, z;
};

ostream& operator<<(ostream& os, const R3& p) {
    return os << "(" << p.x << " " << p.y << " " << p.z << ")";
}

R3 operator*(double t, const R3& p) {
    return R3(t*p.x, t*p.y, t*p.z);
}
```

Some new syntax used in this code: when we put const after a member function, it means that the object that the function is called on never changes.

"You know, only close friends can access your private members"

— Nevard, 9/12/19

Before we continue with operator overloading, here are some functions that should always be in any class:

```cpp
class X {
    X(); // default constructor
    X(double x = 0, double y = 0, double z = 0); // still counts as default constructor
        since all parameters are optional
    // Cannot have both constructors; choose one

    X(const X& x); // copy constructor

    X& operator=(const X& x); // Assignment overload
};

vector<X> a(17); // needs default constructor
f(X x); // calls copy constructor
X x,y,z;
y = x = z; // Assignment parses right to left: y = (x = z)
// Then we need to pass back a reference to x to assign something to y
```

When an assignment operator is not explicitly defined, the compiler tries to make a ”basic” one for you. However, this can have unintended effects. As we discussed earlier, a vector actually contains a pointer to an array, and not the array itself. So, if we were to use the ”basic” assignment for vectors, it would just copy the pointer over, and not the actual data. Therefore, this would make a *shallow copy*, where changes in one object cause changes in the other.

§1.17 Assignment Operators

Now, we discuss how to define assignment operators for custom classes:

```cpp
class R3 {
    // rest of code for R3 class

    // Overloaded versions must be members
    R3& operator+= (const R3& p) { // a += b is an Lvalue
        x += p.x;
        y += p.y;
        z += p.z;
        return *this; // Reference
    }

    R3 operator+ (const R3& p) const {
        return R3(x+p.x, y+p.y, z+p.z);
    }
    // Either we can define + inside R3, or...
};

// OR: outside of R3, we can define binary operator for +
R3 operator+ (const R3& p, const R3& q) {
    R3 r = p; // copy constructor
    r += q; // calls assignment operator
    return r; // calls copy constructor ("move constructor")
}
```

Let’s do another elaborate example:

```cpp
// A complex number class

class Complex {
public:
    Complex(double x = 0.0, double y = 0.0)
    : x(x), y(y) {}

    Complex(const Complex& z): x(z.x), y(z.y) {}

    Complex& operator=(const Complex& z) {
        x = z.x;
        y = z.y;
        return *this;
    }

    Complex& operator/=(const Complex& z) {
        double norm = z.x*z.x + z.y*z.y;
        double nx = (x * z.x + y * z.y)/norm;
        double ny = (y * z.x - x * z.y)/norm;
        x = nx;
        y = ny;
        return *this;
    }

    // similar for +=, *=, -=

    Complex operator-() const {
        return Complex(-x, -y);
    }
private:
    double x, y;
};

Complex operator+ (const Complex& z, const Complex& w) {
    Complex sum = z;
    sum += w;
    return sum;
}
```

A 1-parameter constructor is a **conversion operator**. For example, if we call `Complex z1(17);`, we are converting a double, 17, to a complex number. This renders something like the following legal:

```
Complex w;
w = 3;
```

However, this feature can be disabled. For example,

```
vector<int> a(17), b;
b = 17; // Illegal
```

the compiler might assume that from the 1-parameter constructor `vector<int> a(17)`, we are able to convert from a double to a vector. But in the `vector` class,

```
class vector {
    // Rest of code
    explicit vector(size_t n_elts, T t = 0) { ... }
}
```

we explicitly define the constructor for `vector`. This disallows the conversion constructor.

If we wanted to override the ostream `<<` operator for this class, we would write

```
ostream& operator<< (ostream& os, const Complex& z) {
    return os << z.re() << " + " << z.im() << "i";
}
```

although this implementation would not account for complex numbers like $2 + 0i$ or $0 + 1i$.

We can also override the input for this class (i.e. we can directly read a Complex number from the user):

```
istream& operator>>(istream& in, Complex& z) {

}
```

§1.18 Templates

Templates allow us to generalize the type of arguments passed into a function.

```
template<typename T>
void swap(T & x, T & y) {
    T t = x; // Needs copy constructor
    x = y; // Needs assignment operator
    y = t; // Needs assignment operator
}

// Working code
long x = 17, y = 39;
int z = 3, w = 5;
swap(x,y);
swap(z,w);

swap(z,x); // Error because different data type
```

In fact, the class `vector` uses a template:

```
template<typename T>
class vector {
private:
    size_t size;
    size_t cap;
    T* ptr;
}

vector<int> a(17);
vector<double> b(23);
vector<Mod> c(117);
vector<Dog> d;
Dog fido(...);
d.push_back(fido);
```

```
#include <utility>

template<typename F, typename S>
class pair {
public:
    F first;
    S second;
    pair(const F &first, const S &second): first(first), second(second) {}
};

using PSD = std::pair<string, double>;
PSD x("john", 17);
```

```
template<class T>
class Sortable {
public:
    // other code
    bool operator<(const Sortable& x) const;
}
```

Templates encourage **code reuse**, which is the principle of reusing code as much as possible, a good practice in programming.

§1.19 Pointers to Objects

```cpp
class X {
public:
    void f();
    int g();
private:

}

X *p = new X();
// Instead of writing (*p).f();
p->f();
p->a;
```

§1.20 Functors

A class with a function call can be overloaded. This class would then be called a **functor**. Consider, for example, this Polynomial class:

```cpp
#include <iostream>
#include <vector>
using VD = vector<double>;
using std::cout;

class Poly {
public:
    Poly(const VD& coeffs); // c[0]+c[1]*x+...+c[n-1]x^{n-1}
    double operator()(double x) const; // use Horner's method
private:
    VD c;
};

template<typename F, typename D>
double newton_solve(double x0, const F& f, const D& df) {
    for(int i = 0; i < MAX; ++i) {
        double dx = f(x0)/df(x0);
        if (fabs(df(x0)) < EPS * fabs(df(x0))) {throw new BAD_NEWTON();}
        if(fabs(dx) < EPS * fabs(x0)) return x0;
        x0 -= dx;
    }
    cerr << "Max iterations exceeded!\n";
    throw new BAD_NEWTON();
    return 0; // rage against the machine
}

int main() {
    Poly q(VD{2, 3, 5}); // q = 2 + 3x + 5x^2
    cout << q(4) << '\n';
}
```

```cpp
#include <algorithm>

using VD = vector<double>;
using FID = double(*)(int); // Pointer to a function taking an int arg returning a
    double
using FDDB = bool(*)(double, double);

```

```
template<typename I, typename Cmp>
void sort(I beg, I end, Cmp cmp = ...) { // where cmp is arbitrary compare functor
//code
    if(cmp(*p, *q) < 0) {
        // Code here
    }
}


FDDB p = my_cmp; // passing a function without parentheses makes it a function pointer;
VD a(17);
sort(a.begin(), a.end(), p);
```

§1.21 Iterators

An **iterator** is an object with certain functions overloaded:

1. The unary `*` operator
2. The unary `++` operator
3. The unary `--` operator
4. The unary `[]` operator (subscripting)

An iterator with the first two operators a **forward iterator**. An iterator with the first three is called a **bidirectional iterator**. If the iterator also supports subscripting, it is a **random-access iterator**.

Iterators can also be classified into **write-only** and **read-only**, depending on whether the `*` operator returns an Lvalue or not.

Note that a built-in pointer is an iterator! So, we can do the following:

```
int a[] = {3,1,4,1,5,9};
sort(a,a+6); // also sorts entire array
```

A keyword useful when dealing with iterators is `auto`. It tells the compiler to figure out what type to use.

```
template<class V>
void f(const V& v) {
    // V::const_iterator it = v.begin();
    auto it = v.begin();
}
```

§2 Structures

§2.1 Linked Lists

§2.1.1 Definition

Linked lists are put together with each element having not just content to store but also a `next` pointer which points to the next element in the list.

```cpp
template <typename T>
struct Link {
    explicit Link (const T& info, Link *next=nullptr) : info(info), next(next) {}
    ~Link() { delete next; } // deletes the whole list
    // If you just want to delete one element, set next=nullptr first and then delete
        next

    T info;
    Link *next;
};

Link<string> *head = new Link<string>("hi!");
vector<string> vs {"one","two","three"};
for (int i = 0; i < vs.size(); ++i) {
    head = new Link<string>(vs[i], head);
}
// "three" --> "two" --> "one" --> "hi"
```

Linked lists often are not "allowed" to be empty. Instead, we use a special element called a **sentinel**. A sentinel is an element of the linked list that signifies that we have reached the end of the list.

§2.1.2 Circular Linked Lists

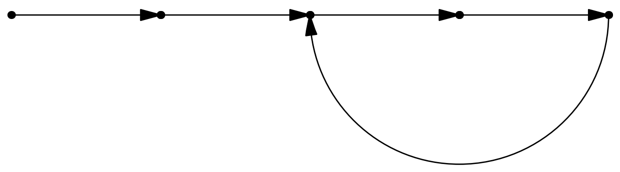
A **circular linked list** is a linked list whose end element's next pointer points to the "head," creating a "circular" chain of elements. Thus, it can be implemented with our without a "head."

§2.1.3 Doubly-Linked Lists

Doubly-Linked Lists are just like regular linked lists, except there is also a `prev` pointer which points to the previous element in the chain.

§2.1.4 Floyd's Algorithm

Floyd's algorithm is an efficient way to detect loops in a linked list. It is not hard to see that the only possible way a linked list can loop is as follows:



We call the **tail length** the number of links that are not part of the loop, and the **cycle length** the number of links that are part of the loop. In the above graph, for example, the tail length is 2 and the cycle length is 3. How can we determine:

1. Whether there is a loop?

2. If it exists, what is the cycle length?
3. What is the tail length?

We use **Floyd's Algorithm** to find these three details: Start with two pointers, one "slow" (call it `s`) and one "fast" (call it `f`). First set `s` and `f` at the starting node.

1. Increment `s` one link at a time, and `f` two links at a time, until `f == nullptr` (then there is no loop) or `f == s` (then there is a loop).
2. Determine cycle length: Advance `s` until `s == f`.
3. Determine length of tail: Move `s` and `f` to the beginning. Move `f` the cycle number of times. Then move `s` and `f` one link at a time and count until `s == f`.

What is the runtime of this algorithm? We reach the end of the tail after moving `s` t times and `f` $2t$ times. Then, since initially the distance between s and f is t links, and each move brings them one closer, the number of moves needed to bring the two together is $(-t) \pmod c \leq c$ turns, which is actually $3c$ moves. So the total number of moves for step 1 is $3(c+t)$ at most. Therefore, it is $O(t+c)$, or linear in the total number of links. Clearly, step 2 takes c moves. Finally, step 3 has $c+2t$ moves in total: c to set up f and $2t$ to reach the end of the tail. In total, the entire algorithm takes less than $5n$ moves.

§2.1.5 Brent's Algorithm

Let $\{a_n\}_{n=0}^{\infty}$ be strictly increasing positive integers. Then:
Start with two pointers, s and f. For all integers i, starting at 0 and increasing:

1. Increment f a_i times, stopping if f becomes null or `s == f`.
2. Set `s = f`.

In practice, we set $a_n = 2^n$.
This algorithm is generally faster than Floyd's algorithm, even though we will not prove anything about its runtime.

§2.1.6 The Josephus Problem

The following theoretical problem is based on a tale of Joseph from Greek mythology. Consider a circle of n people in a circle, labeled $0, \dots, n-1$. Start from person 0, and go around the circle in order, going to the next kth person and "kill" that person. For example, if we have $n = 10$ and $k = 3$, we would kill person 3 first. After person 3 is killed, we move along the circle another 3 times and kill person 6. Similarly, person 9 is killed as well. But at this point, the only people surviving are

$$0, 1, 2, 4, 5, 7, 8.$$

After killing person 9, we count 3 people *out of the surviving ones* and kill person 2 next. Then, continuing until all people in the circle are killed, the **Josephus sequence** for $n = 10$ and $k = 3$ would be

$$3, 6, 9, 2, 7, 1, 8, 5, 0, 4,$$

the order in which the people are killed (i.e. person 4 would be the last to get killed).
The idea to solve for the Josephus sequence is to use a linked list for the circle of people, and eliminate one person at a time.

§2.2 Stacks

Before talking about data structures, we must start with a definition:

Definition 2.1. An **Abstract Data Type (ADT)** is a structure whose behavior is specified, but not the implementation.

```
#include <stack>

stack<int> s;
s.push(3);
s.push(17);
while (!s.empty()) {
    int t = s.top(); s.pop(); // rage against the machine
    cout << t << '\n';
}
```

From this code we see what we want a stack to be able to do:

1. Create an empty instance
2. Test if empty
3. Add an element to stack
4. Remove most recently added item

One possible implementation of the stack uses the linked list. We can let the stack point to the head of the list.

```
class Stack {
public:
    void push(int x) {
        stack = new Link(x, stack);
    }
    int top() const {
        if (!stack)
            throw new EmptyStackException();
        return stack->info;
    }
    void pop() {
        if (!stack) {
            throw new EmptyStackException();
        }
        Link* n = stack->next;
        delete stack;
        stack = n;
    }
private:
    Link* stack;
}
```

However, storing all of those next pointers is inefficient. A better implementation is as an array: We keep an array of integers, with a `stack` pointer pointing to the top element. To add an element we do: `*stack++ = x`. To pop an element we do: `return *--stack`.

We maintain a stack pointer which always points to the end of the stack, which is just many stack frames put together. A **stack frame** is a block of memory which keeps track of, in order:

- Return address

- Return value
- Parameters
- Pointer to next stack frame

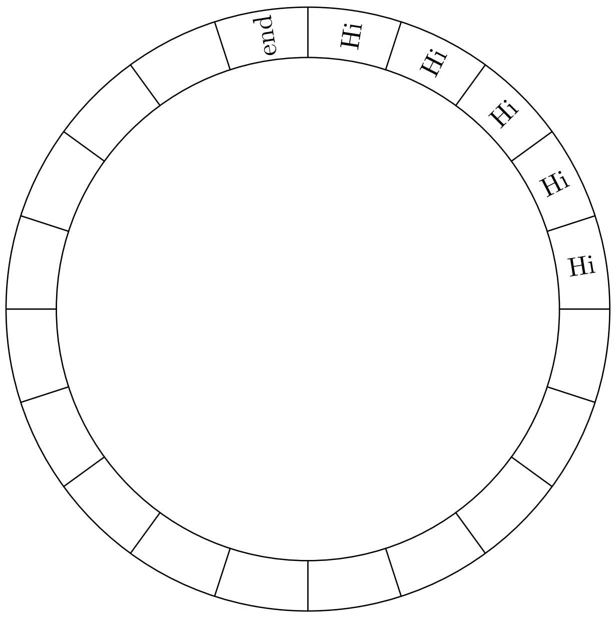
It remains to comment that the stack is a last in, first out structure, or **LIFO** structure.

§2.3 Queue

A **queue** is another ADT that has similar operations to a stack, but functions in a different way. In particular, it is a **FIFO** structure: first in, first out. We want a queue to support the following operations:

- `create`
- `is_empty`
- `enqueue`
- `dequeue` (remove oldest element)
- `front` (show oldest element)

A queue is somewhat more complicated to implement than a stack. If we use the same array-based approach, then we would have to keep track of both the front and back of the queue (since we remove from the front but add at the back). However, there is a way to remain efficient while doing this, by using a **circular array**.



With an array implementation, enqueue and dequeue are $O(1)$ operations.

Without further ado, we delve into the code. Here is C++ 's implementation of the queue:

```cpp
#include <queue>

queue<int> q;
q.push(5);
```

```
q.push(3);
q.push(7);
cout << q.front() << "\n"; // 7
cout << q.back() << "\n"; // 5
q.pop();
cout << q.front() << "\n"; // 3
```

Here is an array implementation:

```
template<typename T>
class Queue {
public:
    void enQ(const T& x) {
        if (full()) {
            grow();
        }
    }
    void grow() { // Double size of array
        T *p = new T[2*(end - a)];
        // Copy elements from a to p
        // Update f,b
        delete[] a;
        a = p;
        end = m;
    }
}
```

It is important to note that doubling the size of the array when we grow ensures that (asymptotically) the expansion only takes as much time as actually inserting each element.

§2.4 Deque

A **deque** is a double-ended queue that supports the following operations:

- `push_front`
- `push_back`
- `pop_front`
- `pop_back`
- `is_empty`

All of these should take $O(1)$ time.
Deques are implemented in exactly the same way as queues.

§3 Asymptotic Analysis

We seek to classify algorithms by time and space requirements.

Size can be associated with an instance of the algorithm. For example, in sorting, the "size" of the input could be considered the number of input elements (although this is an abstraction, since we are not specifying the size of the items).

As a function of n, how is the fast is the algorithm? How much space does it require? For each of these questions, it is important for us to consider:

- Average case
- Best case
- Worst case

Sometimes, the average case is hard to compute, especially without discussing probability distributions. The worst case can be unnecessarily pessimistic.

Now, how do we specify what units we use when we describe time requirements? We can't use "seconds" or "number of instructions" because that is specific to your machine, and not a property of the algorithm. We usually specify a higher level operation, and measure how many times that operation is used. For example, in sorting, we would use "number of comparisons," or the "number of swaps."

Therefore, for a given application, what do we count? It should be platform and language independent, meaning that it doesn't matter whether you're running the algorithm on your own laptop or a supercomputer, or using Java or Python. Furthermore, how do we express such a number?

Space is the number of bytes. Again, we consider: How much space does the algorithm require? But this brings us to the same issue: what do we count?

To gain, let $T_1(n) = 2.1n^2 - 3n + 17$ and $T_2(n) = 2.5n^2 - 53n - 1000$ for two algorithms A_1 and A_2 respectively. For small n, T_2 is faster than T_1, but for large n, T_1 is faster (just look at the leading terms):

$$\lim_{n\to\infty} \frac{T_1(n)}{T_2(n)} = \frac{2.1}{2.5} < 1.$$

Definition 3.1. $f(n) \sim g(n)$ (read as "$f(n)$ is **asymptotic** to $g(n)$") if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

A famous example in mathematics is the Prime Number Theorem. Let $\pi(n)$ be the number of positive primes less than or equal to n. For instance, $\pi(10) = 4$ and $\pi(100) = 25$. Then,

$$\pi(n) \sim \frac{n}{\ln n}.$$

Specifically, mathematician Pafnuty Chebyshev proved that if $\lim\limits_{n\to\infty} \frac{\pi(n)}{\left(\frac{n}{\ln n}\right)}$ exists, it is 1. It was proven later that the limit does in fact exist.

Definition 3.2. We say $f(n) = O(g(n))$ if

$$\exists M > 0 : \lim_{n\to\infty} \left| \frac{f(n)}{g(n)} \right| < M$$

In other words, for sufficiently large n,

$$|f(n)| \leq M|g(n)|.$$

In computer science, we usually drop the absolute values since everything is positive.

For example, $T_1(n) = 2.1n^2 - 3n + 17 = O(n^2)$, because for large n, $T_1(n) < 3n^2$.

It is important to realize that $T_1(n)$ is also $O(2^n)$, since all we care about is that f is smaller than g.

Definition 3.3. $f(n) = \Theta(g(n))$ if

$$\exists M_1, M_2 > 0 : \text{for sufficiently large } n,\ M_1|g(n)| \leq |f(n)| \leq M_2|g(n)|.$$

For example, $T_1(n) = \Theta(n^2)$ but $T_1(n) \neq \Theta(n^3)$.

Definition 3.4. $f(n) = \Omega(g(n))$ if for sufficiently large n,

$$\exists M > 0 : M|g(n)| \leq |f(n)|.$$

Definition 3.5. $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

For example, $T_1(n) = o(n^3)$.

Definition 3.6. $\lg^*(n)$ is the greatest number of lgs required so that

$$\underbrace{\lg\lg\lg\lg\cdots\lg\lg\lg(n)}_{\lg^*(n)} < 1.$$

Problem 3.7. Rank by order of growth. If $f(n) = \Theta(g(n))$ they have same rank:

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$\left(\frac{3}{2}\right)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{\frac{1}{\lg n}}$
$\ln\ln n$	$\lg^* n$	$n2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n\lg n$	$2^{2^{n+1}}$

Solution.

1. $n^{\lg n}$
2. $\lg(\lg^* n)$
3. $\lg^*(\lg n)$, $\lg *n$
4. $2^{\lg^* n}$
5. $\ln\ln n$
6. $\sqrt{\lg n}$
7. $\ln n$
8. $\lg^2 n$

   Note that this is the same as $2^{2\lg\lg n}$.

9. $2^{\sqrt{2\lg n}}$
10. $\sqrt{2}^{\lg n}$
11. n, $2^{\lg n}$
12. $n \lg n$, $\lg n!$

    Notice that $\lg n! = \sum_{k=2}^{n} \lg k = \Theta\left(\int_1^n \lg x \, dx\right)$. Furthermore, **Stirling's Approximation** notes that $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.
13. n^2, $4^{\lg n}$
14. n^3
15. $(\lg n)!$

    As a fun fact, here is the **Gamma Function**:

    $$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} \, dt,$$

    from which it follows from integration by parts that $\Gamma(x+1) = x\Gamma(x)$ and more generally $\Gamma(n) = (n-1)!$.
16. $(\lg n)^{\lg n}$, $n^{\lg \lg n} = (2^{\lg n})^{\lg \lg n}$
17. $\left(\frac{3}{2}\right)^n$
18. 2^n
19. $n2^n$
20. e^n
21. $n!$
22. $(n+1)!$
23. 2^{2^n}
24. $2^{2^{n+1}}$ $\square$