Review of Prerequisite Topics

1 Mathematical Preliminaries

This is the math that you are required to know for the remainder of the course. It is extremely important that you commit these formulae to memory and know how to apply them in practice, because they do arise often in the study of algorithmic analysis, and in computer science in general.

1.1 Series Summations

Arithmetic Series The formula for an *arithmetic series* is

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2} \tag{1}$$

Equation 1 is easily proved by mathematical induction. From this formula you can solve the more general arithmetic series sum of the form

$$\sum_{k=0}^{n}(ak+b) \tag{2}$$

by writing it out directly and redistributing the terms of the series:

$$\begin{aligned}\sum_{k=0}^{n}(ak+b) &= a\sum_{k=0}^{n}k+\sum_{k=0}^{n}b \\ &= \frac{an(n+1)}{2}+(n+1)b \qquad (3)\end{aligned}$$

If it appears that the series does not start at 0, but say, at c, then observe that

$$\begin{aligned}\sum_{k=c}^{n}(ak+b) &= (a(c+0)+b)+(a(c+1)+b)+\ldots+(a(c+n-c)+b) \\ &= \sum_{k=0}^{n-c}(a(c+k)+b) \qquad (4)\\ &= \sum_{k=0}^{n-c}(ac+ak+b) \qquad (5)\\ &= \sum_{k=0}^{m}(ak+d) \qquad (6)\end{aligned}$$

where $m = n - c$ and $d = ac + b$. In other words, it can always be viewed as starting at 0.

To illustrate, suppose you need to find the sum of the sequence $9, 13, 17, 21, 25, 29, 33, ..., 257$. You observe that the difference between each pair of terms is the constant $a = 4$, from which you can conclude that this is an arithmetic series. The first term is 9, so you can take $b = 9$. So you know that $a = 4$ and $b = 9$. Now you need to know the value of n. The last term is $an + b = 257$; so $4n + 9 = 257$. Solving for n we get $n = (257 - 9)/4 = 62$. Therefore, applying Equation 3, the sum is $4 \cdot 62 \cdot 63/2 + 63 \cdot 9 = 133 \cdot 63 = 8379$.

Quadratic Series The formula for the sum of the sequence of squares 1, 4, 9, 16, and so forth, is

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{7}$$

This can be proved by mathematical induction, which we do below.

Higher Order Series The quadratic series is the special case of $m = 2$ in the more general series

$$\sum_{k=1}^{n} k^m \approx \frac{n^{m+1}}{m+1},\ m \neq -1 \tag{8}$$

A discrete sum such as the above can be viewed as an approximation to the definite integral that defines the area under the curve x^m between the points $x = 0$ and $x = n$. From calculus we know that $\int_0^n x^m dx = n^{m+1}/(m+1)$. When $m = -1$, the denominator in Eq. 8 is zero and the right hand side is undefined. In this case there is a different approximation, which can also be seen through the perspective of calculus:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} \approx \int_1^n \frac{1}{x} dx = \ln n \tag{9}$$

.

The sum in Equation 9 for each value of n is given a name, the ***harmonic number*** of order n, H_n. The absolute difference between the n^{th} harmonic number H_n and the integral $\int_1^n \frac{1}{x} dx$, as $n \to \infty$, is ***Euler's constant***, $\gamma \approx 0.577216$.

Geometric Series The formula for the sum of a *geometric series* for $x \neq 1$, is

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \tag{10}$$

This is proved directly by showing that the product of the left-hand-side and the denominator of the right hand side is equal to the numerator of the right-hand-side. The left-hand side is the polynomial of degree n all of whose coefficients are 1. When it is multiplied by $x - 1$, all terms drop out except the x^{n+1} and the 1. When $0 < x < 1$, as $n \to \infty$, this converges to $1/(1 - x)$.

Another series that we will encounter quite a bit is

$$\sum_{k=0}^{n} k a^k \quad 0 < a \tag{11}$$

where a is either 2 or 1/2. Although it is a bit simpler to derive the formula for this series when $a = 2$ or $a = 1/2$, we derive it for arbitrary a as follows, using Equation 10. First observe that

$$\begin{aligned} a\left(\sum_{k=0}^{n} ka^k\right) &= \sum_{k=0}^{n} ka^{k+1} \\ &= a^2 + 2a^3 + 3a^4 + ... + (n-1)a^n + na^{n+1} \qquad (12) \end{aligned}$$

Second, expand the original sum in Eq. 11:

$$\sum_{k=0}^{n} ka^k = a + 2a^2 + 3a^3 + 4a^4 + ... + na^n \qquad (13)$$

and subtract Eq. 12 from Eq 13, adding 1 and subtracting 1:

$$\begin{aligned} \sum_{k=0}^{n} ka^k - a\left(\sum_{k=0}^{n} ka^k\right) &= 1 + a + a^2 + a^3 + a^4 + ... + a^n - na^{n+1} - 1 \\ &= \left(\sum_{k=0}^{n} a^k\right) - na^{n+1} - 1 \\ &= \left(\frac{a^{n+1}-1}{a-1}\right) - \left(1 + na^{n+1}\right) \qquad (14) \end{aligned}$$

The left hand side of this equation is

$$\sum_{k=0}^{n} ka^k - a\left(\sum_{k=0}^{n} ka^k\right) = (1-a)\left(\sum_{k=0}^{n} ka^k\right)$$

so we can divide both sides of Equation 14 by $(1 - a)$ and, adjusting minus signs on the right-hand side, we get

$$\sum_{k=0}^{n} ka^k = \frac{\left(1 - a^{n+1}\right)}{(a-1)^2} + \frac{1 + na^{n+1}}{a-1} \qquad (15)$$

When we substitute $a = 1/2$ in Equation 15, we have

$$\begin{aligned} \sum_{k=0}^{n} \frac{k}{2^k} &= \frac{\left(1 - 1/2^{n+1}\right)}{1/4} + \frac{1 + n/2^{n+1}}{-1/2} \\ &= 4 - \frac{1}{2^{n-1}} - 2 - \frac{n}{2^n} \\ &= 2 - \frac{1}{2^{n-1}} - \frac{n}{2^n} \qquad (16) \end{aligned}$$

As $n \to \infty$, the right hand side approaches 2, since both $1/2^{n-1}$ and $n/2^n$ approach 0 as $n \to \infty$. Hence

$$\sum_{k=0}^{\infty} \frac{k}{2^k} = 2$$

When $a = 2$ in Eq. 15the result, which you can verify, is

$$\sum_{k=0}^{n} k2^k = 2 + (n-1)2^{n+1} \tag{17}$$

1.2 Modular Arithmetic (Congruences)

You should be familiar with the basic rules of modular arithmetic, but perhaps you have not heard the language associated with it. Computer science students write $a = b \,\%\, N$ to mean that a is the remainder of the integer division b/N. But actually, in mathematics, we say that two numbers a and b are ***congruent modulo*** N if their absolute difference, $|a - b|$, is divisible by N, or in other words, if there exists an integer q such that $q \cdot N = |a - b|$, and we write $a \equiv b \, mod \, N$, or $a \equiv_N b$. I will use both notations here.

When numbers are congruent modulo N, their remainders when divided by N are the same. E.g., $53 \equiv_8 29 \equiv_8 13$ because they all have remainder 5 when divided by 8. In general, modular arithmetic obeys these same rules as integer arithmetic:

- If $a \equiv b \, mod \, N$, then $a + c \equiv b + c \, mod \, N$ and
- If $a \equiv b \, mod \, N$, then $ad \equiv bd \, mod \, N$

With ordinary integer arithmetic, we know that if $ab = 0$, then either $a = 0$ or $b = 0$. But if $ab \equiv 0$ mod N, it does not imply that one of a or b must be 0. For example, $ab \equiv 0 \mod 12$ can be true if $a = 3$ and $b = 4$.

Furthermore, with rational numbers, we know that for any number $a \neq 0$, $ax = 1$ has a unique solution called its multiplicative inverse. But with modular arithmetic this is not true; a number does not necessarily have a multiplicative inverse. For example there is no x such that $3x \equiv 1$ mod 12.

When N is a prime number, however, which we will now write as p instead of N, the picture becomes much more interesting, because the set of numbers $0, 1, 2, ..., p - 1$ has been turned into a ***field***, specifically a ***finite field***, and the following statements are true:

- $ab \equiv 0 \, mod \, p$ implies that at least one of a or b is divisible by p. For example, if $ab \equiv 0 \, mod \, 53$ then either a or b is 0 or 53.
- If $ax \equiv 1 \, mod \, p$, then x is uniquely determined: there is a single integer x for which it is true, and it is called the ***multiplicative inverse*** of a. For example, if $8x \equiv 1 \, mod \, 11$, then $x = 7$ because $8 \cdot 7 = 56 \equiv 1 \, mod \, 11$.
- The equation $x^2 \equiv a \, mod \, p$ has either no solution or exactly two solutions if $0 < a < p$. For example, $x^2 \equiv 1 \, mod \, 11$ has the solutions $x = 1$ and $x = 10$, whereas the equation $x^2 \equiv 2 \, mod \, 11$ has no solutions.

1.3 Greatest Common Divisor

The ***greatest common divisor*** of two integers a and b is the largest integer that divides both a and b, which we denote by $gcd(a,b)$. For example, $gcd(24,30) = 6$ and $gcd(5,15) = 5$. Formally, d is the greatest common divisor of a and b if d divides both a and b and if any number c divides both a and b, then c divides d. When $gcd(a,b) = 1$, we say that a and b are ***relatively prime***, or ***co-prime***. For example, 5 and 8 are relatively prime. For the domain of integers, every pair of numbers has a unique greatest common divisor. For other domains, the gcd is not necessarily unique, but we restrict our discussion to the integers. An important result concerning greatest common divisors is the following theorem.

Theorem 1. *Let a and b be two integers and let $d = gcd(a,b)$. Then there exist two integers x and y such that $d = ax + by$.*

Proof. Let $\mathbb{N}$ be the set of all integer linear combinations of a and b. Formally, $\mathbb{N} = \{ra + sb \,|\, r,s \in \mathbb{Z}\}$. Observe that $\mathbb{N}$ is closed under addition (and subtraction) and multiplication, because for any integers $r_1, s_1, r_2, s_2, t \in \mathbb{Z}$, we have

$$(r_1a + s_1b) \pm (r_2a + s_2b) = (r_1 \pm r_2)a + (s_1 \pm s_2)b \in \mathbb{N}$$

since $(r_1 \pm r_2), (s_1 \pm s_2) \in \mathbb{Z}$. Also,

$$t(ra + sb) = (tr)a + (ts)b \in \mathbb{N}$$

since $(tr), (ts) \in \mathbb{Z}$.

Since $\mathbb{N}$ is a subset of the integers, it has some smallest positive member. Let d be the smallest positive number in $\mathbb{N}$. By the divisibility property of the integers, we know that, for any $n \in \mathbb{N}$ there are integers q and r such that

$$n = qd + r$$

where $r = 0$ or $0 < r < d$. Suppose $r \neq 0$. Then $0 < r < d$ and $r = n - qd = 1 \cdot n + (-q) \cdot d$. Since $n \in \mathbb{N}$ and $d \in \mathbb{N}$ and $\mathbb{N}$ is closed under addition, r must be in $\mathbb{N}$ as well. But then r would be a member of $\mathbb{N}$ smaller than d which is a contradiction, because we chose d to be the smallest positive member of $\mathbb{N}$. Therefore, $r = 0$. But then

$$n = qd$$

for some $q \in \mathbb{Z}$. This shows that $d|n$. Since we chose $n \in \mathbb{N}$ arbitrarily, this shows that for every $n \in \mathbb{N}$, d is a divisor of n. Let $r = 1$ and $s = 0$. Then

$$d \,|\, (ra + sb) \Rightarrow d \,|\, a$$

and letting $r = 0$ and $s = 1$

$$d \,|\, (ra + sb) \Rightarrow d \,|\, b$$

so d is a common divisor of a and b. Suppose that c divides a and c divides b. Then clearly c divides ra and c divides sb, so $c \,|\, (ra + sb)$. This implies that c divides all numbers in $\mathbb{N}$. In particular, c divides d, which implies that $c \leq d$ and therefore d must be a greatest common divisor of a and b. Is it unique? Since $d \in \mathbb{N}$, there are integers x and y such that $d = ax + by$. If there were another number d_1 that was a gcd of a and b then we would have to have $d \,|\, d_1$ by the definition of the gcd, and since $d = ax + by$ and d_1 divides a and b, $d_1 \,|\, d$. If $d \,|\, d_1$ and $d_1 \,|\, d$ then $d = d_1$. Therefore d is the unique gcd of a and b. □

This is a very important statement. The *gcd* of any two numbers is a linear combination of the two numbers. If a and b are relatively prime, their *gcd* is 1, and therefore there are x and y such that $1 = ax + by$. This also implies that every integer c can be written as a linear combination of a and b, because $c = c \cdot 1 = c \cdot (ax + by) = acx + bcy$. The x and y do not have to be positive; in fact it is not always possible to find two non-negative integers x and y in this theorem. But under certain conditions, we can:

Theorem 2. *Let a and b be two relatively prime positive integers. If $d \geq (a-1)(b-1)$ then there exist non-negative integers x and y such that $d = ax + by$.*

Proof. Since a and b are relatively prime, $1 = gcd(a, b)$. By Theorem 1, there exist integers x and y such that $1 = ax + by$. Therefore, $d = adx + bdy$. Let $x_0 = dx$ and $y_0 = dy$. Then $d = ax_0 + by_0$. Since d is a non-negative number, at least one of x_0 or y_0 must be non-negative. Let us assume that $x_0 \geq 0$ but $y_0 < 0$. Observe that for any x and y, if $d = ax + by$ then $d = a(x - b) + b(y - a)$, because $a(x-b) + b(y+a) = ax - ab + by + ab = ax + by = d$. In particular,

$$d = a(x_0 - b) + b(y_0 + a)$$

If $y_0 < 0$ it is not possible that $x_0 - b < 0$. To see this, suppose to the contrary that $x_0 - b < 0$. Then $x_0 < b$, implying that $x_0 \leq b - 1$. Also, since $y_0 < 0$, $y_0 \leq -1$. This would imply that

$$d = ax_0 + by_0 \leq a(b-1) + b(-1) = ab - a - b < ab - a - b + 1 = (a-1)(b-1)$$

which contradicts the premise that $d \geq (a-1)(b-1)$. Hence $x_0 - b \geq 0$. This shows that if we have a pair of numbers x and y such that $d = ax + by$ but that $y < 0$, we can find a second pair of numbers $x' = x - b$ and $y' = y + a$ such that $d = ax' + by'$ but for which $x' \geq 0$ and $y' > y$. This leads to the pair we need, as we now formalize.

Let $x_1 = x_0 - b$ and $y_1 = y_0 + a$. In general, let $x_{i+1} = x_i - b$ and $y_{i+1} = y_i + a$, for all $i \geq 0$. You should see that all pairs (x_i, y_i) satisfy $d = ax_i + by_i$. The sequence $y_0, y_1, y_2, \ldots$ is a strictly increasing sequence and so not all y_i are negative. Let k be the largest index such that y_k is negative. Then $y_{k+1} = y_k + a$ is non-negative. For this k, we have $d = ax_k + by_k$ with $y_k < 0$, so $x_{k+1} = x_k - b \geq 0$ by our preceding discussion. Therefore, we have found a pair, (x_{k+1}, y_{k+1}) such that $d = ax_{k+1} + by_{k+1}$ and both x_{k+1} and y_{k+1} are non-negative. □

1.4 Proofs

Proofs are important. When we make a claim that something is true, we have to prove that it is true. Although there are philosophical arguments about what constitutes a proof, we will restrict this discussion to methods of proof that no one disputes are valid. Three such methods of proof can be used to solve most problems:

1. proof by mathematical induction,
2. proof by contradiction, and
3. proof by counterexample.

We describe and give examples of each.

1.4.1 Proof by induction

Suppose a statement S can be formulated as depending on a single non-negative integer n. A proof of S by mathematical induction has two parts: a base case that establishes that $S(n)$ is true for a finite number of values of n, typically that it is true for $n = 0$, and an inductive argument that shows that if S is true for arbitrary k, then it is also true for $k + 1$. I.e., $S(k) \implies S(k + 1)$. We demonstrate this method by proving Equation 7 above using induction. We display the equation again for easy reference:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

Base Case: n = 0 . The left side of the equation is 0 and the right hand side is also 0. Thus, the base case is true.

Inductive Hypothesis We assume the equality is true for $n = N$:

$$\sum_{k=0}^{N} k^2 = \frac{N(N+1)(2N+1)}{6}$$

Inductive Step We prove that it will be true when $n = N + 1$, given that it is true for $n = N$:

$$\begin{aligned}
\sum_{k=0}^{N+1} k^2 &= \sum_{k=0}^{N} k^2 + (N+1)^2 \\
&= \frac{N(N+1)(2N+1)}{6} + (N+1)^2 \\
&= (N+1)\left(\frac{N(2N+1)}{6} + \frac{6(N+1)}{6}\right) \\
&= (N+1)\left(\frac{2N^2+7N+6)}{6}\right) \\
&= (N+1)\left(\frac{(N+2)(2N+3)}{6}\right) \\
&= = (N+1)\left(\frac{(N+1+1)(2(N+1)+1)}{6}\right)
\end{aligned}$$

which shows that the equation is valid when $n = N + 1$.

1.4.2 Proof by Contradiction

In a proof by contradiction, the idea is to assume that the statement to be proved true is false and then show that the assumption that it is false leads to some contradiction. In symbolic language, let S be a statement to be proved true. Suppose that S implies that some statement R and its negation, $\sim R$ are both true. Then S implies (R and $\sim R$). Since (R and $\sim R$) must be logically false, S implies false, which by *Modus Tollens* implies that S is false. (*Modus Tollens states that if P implies Q and Q is false, then P is false.*)

Example 3. Proof that the square root of 2 is an irrational number.

Suppose that the square root of 2 is rational.

Then by the definition of a rational number, there are two integers p' and q' with $q' \neq 0$, such that $(p'/q')^2 = 2$.

This implies that there are two integers p and q that are relatively prime to each other such that $(p/q)^2 = 2$, because we can let $p = p'/\gcd(p', q')$ and $q = q'/\gcd(p', q')$ where gcd means "greatest common divisor".

Since $(p/q)^2 = 2$, $p^2 = 2q^2$ implying in turn that p^2 is even.

This means p is even because if it were odd, p^2 would be odd. So $p = 2m$ for some m, and $p^2 = 4m^2$. Hence $4m^2 = 2q^2$, implying $2m^2 = q^2$ implying q^2 is even which then implies q is even for the same reason that we stated above regarding p.

But then p and q are each even and cannot be relatively prime, which is a contradiction. So the supposition that square root of 2 is rational leads to a contradiction and therefore it must be false.

Example 4. This is another proof that the square root of 2 is an irrational number, but it is more intuitive; it is from Alexander Bogomolny. It is based on the *Fundamental Theorem of Arithmetic*, which states that every number is uniquely (up to the order of factors) representable as a product of primes. Assume the square root of 2 is rational and let $(p/q)^2 = 2$ for some integers p and q. Then $p^2 = 2q^2$. Factor both p and q into a product of primes. p^2 is factored into a product of the very same primes as p each taken twice. Therefore, p^2 has an even number of prime factors. So does q^2 for the same reason. Therefore, $2q^2$ has an odd number of prime factors. As $p^2 = 2q^2$, it cannot have both an even number of prime factors and an odd number of prime factors simultaneously, so this is a contradiction.

1.4.3 Proof by counterexample

There are two types of quantified statements: *universal statements* and *existence statements*. A universal statement is of the form, "For every such-and-such, P(such-and-such) is true" where P is some predicate involving such-and-suches. The symbol $\forall x$ means "for all x." For example,

"Every month has 31 days."

is a statement that says, "for every month x, 'has-thirty-one-days(x)' is true". This is clearly false and is easily proved by showing that there is some month that does not have 31 days, such as April. An existence statement is of the form, "There is a such-and-such for which Q(such-and-such) is true." This can be proved by finding a single such-and-such that makes Q true, but it is disproved by showing that for any possible such-and-such, Q(such-and-such) is false.

The proof that P is not universally true for every month is an example of a proof by counterexample. Proof by counterexample is used to prove that universal statements are false. It cannot be used to prove that universal statements are true.

Example 5. The *Fibonacci* numbers are defined recursively as follows:

$F_0 = 1$

$F_1 = 1$

$F_{n+2} = F_{n+1} + F_n$

F_k is the k^{th} Fibonacci number, which are important numbers that arise in unexpected places in nature and mathematics alike. Suppose we want to prove that the statement $(\forall k)F_k <= k^2$ is false. This is a universal statement, and to prove it false we need a witness to its falsehood, i.e., a counterexample. We can take $k = 11$: $F_{11} = 144 > 121$.

2 *C++* Review

This is a review of selected topics about *C++* that should have been covered in the prerequisite courses. They are here mostly for reference.

2.1 Functions with Default Arguments

C++, unlike *C*, allows any function to have default arguments for its parameters. A default argument is the value assigned to a formal parameter in the event that the calling function does not supply a value for that parameter.

Example 6. The declaration

```
void point2d(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type `int`. It can be called in any of these ways:

```
point2d(1,2);
point2d(1);
point2d();
```

The last two calls are equivalent to `point2d(1,4)` and `point2d(3,4)`, respectively.

The syntax for declaring default arguments is:

`return_type function_name (` $t_1\ p_1$, $t_2\ p_2$, ..., $t_k\ p_k = d_k$, ..., $t_n\ p_n = d_n$`);`

where each t_k is a type symbol, p_k is a parameter symbol, and d_k is an initializing expression, which should be a constant literal or a constant global variable. It is not allowed to be a local variable. If parameter p_k has a default value, then all parameters p_i, with $i > k$ must also have default values. In other words, *trailing parameters*, the parameters to its right in the list of parameters must have default values also.

If a function is declared prior to its definition, as in a class interface, the defaults should not be repeated again – it is not necessary and will cause an error.

If default parameters are supplied and the function is called with fewer than n arguments, the arguments will be assigned in left to right order, as the `point2d()` example illustrated. As another example, given the function declaration,

```
void carryOn( int count, string name = "", int max = 100);
```

`carryOn(6)` is a legal call that is equivalent to `carryOn(6, "", 100)`, and `carryOn(6, "flip")` is equivalent to `carryOn(6, "flip", 100)`. If argument `k` is not supplied, then all arguments to the right of `k` must be omitted as well. The following are all invalid.

```
void bad1(int a = 2, int b    , int c);
void bad2(int a = 1, int b = 2, int c);
void bad3(int a    , int b = 3, int c);
```

The C++ standard also allows you to add default arguments to overloaded function declarations (not definitions) at a later time. The following program is valid code.

```
void f(int a, int b, int c);           // f() has no defaults
void f(int a, int b, int c = 1);       // c is given a default
void f(int a, int b = 1, int c);       // b is given a default
void f(int a = 1, int b, int c);       // a is given a default

int main()
{
   return 0;
}

void f( int a, int b, int c)
{
    //stuff here
}
```

Be warned though that there are not many good reasons to do this in your programs. The reason that default arguments are important is that they are particularly useful in reducing the number of separate constructor declarations within a class. One can write good programs without ever using default arguments for function parameters.

2.2 Member Initializer Lists

Consider the following class definitions.

```
class MySubClass
{
public:
    MySubClass(string s) { m_name = s; }
private:
    string m_name;
};

class X
{
public:
        X(int n, string str ): pmc(quirk), quirk(str), x(n)  { }
    private:
        const int x;
        MySubClass quirk;
        MySubClass& pmc;
    };
```

The constructor

```
    X(int n, string str ): pmc(quirk), quirk(str), x(n) { }
```

causes the constructors for the `int` class and the `MySubClass` class to be called prior to the body of the constructor, in the order in which the members *appear in the definition*[1], i.e., first `int` then `MySubClass`, then passing the address of `quirk` to the constructor for `pmc`.

Member initializer lists are necessary in three circumstances:

- To initialize a constant member, such as `x` above,
- To initialize a member that does not have a default constructor, such as `quirk`, of type `MySubClass`,
- To initialize a reference member, such as `pmc` above. References are explained below.

2.3 Separation of Interfaces and Their Implementations

A class definition should always be placed in a ***header file*** (a `.h` file) and its implementation in an ***implementation file***, typically called a `.cpp` file. The header file should be thoroughly documented; it serves as a contract between the implementation of the class and the client code that uses it, and if you expect someone to use your class or read its interface, then you must describe in unambiguous, complete, and consistent language, what each member function and friend function does. This is nothing new.

If you would like to distribute this class to other users, you should distribute the thoroughly documented header file and the ***compiled implementation file***, i.e., ***object code***. The users do not need to see the source code for the implementation file. Of course, you can only do this if you know the target machine architecture and can compile for that architecture.

For a class, the implementation needs the interface, so you must put an `#include` directive in the `.cpp` file. Remember that the `#include` directive is executed by the preprocessor (*cpp*) and that it copies the included file into the position of the directive in a copy of the source file that it creates. You should always put a ***header guard*** into the header file. A header guard is a construction of the form

```
    #ifndef __HEADERNAME_H
       #define __HEADERNAME_H

       the interface definitions here

    #endif // __HEADERNAME_H
```

[1]Not the order in which they appear in the list!

where `__HEADERNAME_H` is a placeholder for a suitable name. Usually it is the name of the file in uppercase. It is used to prevent multiple inclusions of the same file, which would cause compiler errors. The first time it is encountered, the `ifndef` is true (because it is short for "if not defined"), the symbol `__HEADERNAME_H` is then defined, and the code is included. Subsequent attempts to include the file fail because the symbol is defined so `ifndef` is false.

For those wondering why we bother, if you do not do this, then multiple definitions of the same class will occur when the header file is included indirectly by other files, and this will cause a compile-time error.

Separating the interface and implementation is part of the process of organizing a program into distinct modules to make it easier to maintain the project, and for this reason it is important. It is better to create many small files than a few large ones, because changes are easier to control, compiling and relinking becomes faster, and debugging becomes easier.

Lastly, separation of interface and implementation is not just a good idea for classes, but for modules in general. If you are creating a program that has several utility functions that are needed by many other functions, but they are not really related to each other, you could create two files, `utilities.h` and `utilities.cpp` that have the function prototypes and their implementations respectively, and this will make the program easier to maintain.

2.4 Separate Compilation of Multifile Programs

If you have a project that contains multiple source code files, then these should be compiled separately into individual object files, which would then be linked into a single project executable file. The main reason for doing this is that when a change is made to just a single file, the least possible recompilation and linking is done, saving time.

If your project consists of the files `utilities.cpp, utilities.h, tree.cpp. tree.h, io.cpp, io.h`, and `main.cpp`, then the following lines build the project executable `myproject`:

```
g++ -c utilities.cpp
g++ -c tree.cpp
g++ -c io.cpp
g++ -o myproject main.cpp utilities.o tree.o io.o
```

If `tree.cpp` is changed, it suffices to repeat only

```
g++ -c tree.cpp
g++ -o myproject main.cpp  *.o
```

2.5 Vectors and Strings

There have been some changes in the *C++* standard, *C++11*, that allow you to do things that you could not do before with vectors and with strings. These notes will not discuss the changes, but you are advised to review the textbook to see what they are. Some of these changes are very useful, others, less so. One useful addition is the *range for-loop*. This type of loop existed in the original UNIX Bourne shell (`sh`), was retained in `bash`, made its way into languages like Perl, and has now been added to C++. This loop

```
int sum = 0;
int squares[] = {0,1,4,9,16,25,36};
for ( int x : squares )
    sum += x;
```

states that the variable `x` declared within the loop expression takes on the value of every element of the vector `squares`, so you as coder, do not need to know or specify how many values that is. This cannot be used to modify the vector, since `x` is just a copy of the element. In this particular example, the type of `x` is declared as `int`. If we wanted a loop that worked for any type that supported addition, we could use the new `auto` keyword:

```
int sum = 0;
int squares[] = {0,1,4,9,16,25,36};
for ( auto x : squares )
    sum += x;
```

Here, the compiler determines the type of `x` automatically.

3 *C++* Details

The assumption is that you know about pointers, but might need a reminder about a few things. Here it is.

3.1 Dynamic object creation

In *C++*, the `new` operator dynamically allocates memory on the heap and returns a pointer to the starting address of the created object, as in

```
myclass = new MyClass;
```

The `new` operator is overloaded to create arrays as well, as in

```
p = new int[100];
```

The old *C++* standard specified that, if it fails, it would return a `NULL` pointer. In the most recent standard, it will throw an exception that, unless it is handled, will terminate the program. Thus, if you plan on using *C++11*, you must catch the `std::bad_alloc` exception that `new` might throw, as in

```
#include <cstdio>
#include <new>
using namespace std;
int main()
{
        int sum = 0;
        try {
            int* p = new int[200000000000];
        }
        catch ( bad_alloc ) {
            printf(" too much memory requested \n" );
        }
        return 0;
    }
```

The `delete` and `delete[]` operators free the storage associated with the pointer; the latter is used when the pointer is to an array. Assuming the objects are those created above, we release their resources with

```
    delete myclass;
    delete[] p;
```

If you fail to release memory when you are finished, you will be wasting memory, and guilty of causing ***memory leaks*** as in

```
    while ( 1 ) {
        int* p = new int[1000000000];
        printf("la di da . The ship is leaking and I don\'t care.\n");
    }
```

3.2 References Versus Pointers

The thing to remember is that a reference is another name for the same object, not an object containing its address[2]. Thus,

```
    int x = 4;
    int & y = x;      // y is another name for x, so y == 4
    int z = y;        // z is not a reference
    int* px = &x;     // px is a pointer to x;
    z = 2;
    int & m;          // illegal - m must be bound to an object when declared
    y++;              // increments x
    px++;             // increments px, which now points to something else.
```

3.3 Return Values

A function should generally return by value, as in

[2]This is a white lie. A reference *is* a pointer, but it is a special pointer that can be used with the same syntax as the thing that is pointed to. So if `y` is a reference to `x`, then `y` contains the address of `x`, but can only be used in the program as if it did not contain an address but was a substitute for the name `x`.

```
double sum(const vector<double> & a);
string reverse(const string & w);
```

`sum` returns a `double` and `reverse` returns a `string`. Returning a value requires constructing a temporary object in a part of memory that is not destroyed when the function terminates. If an object is large, like a class type or a large array, it may be better to return a reference or a constant reference to it, as in

```
const string & findmax(const vector<string> & a);
```

which searches through the string vector `a` for the largest string and returns a reference to it. It does not copy the string. Of course if the caller needs to modify it, then passing by `const` reference is not a solution. But in general, passing by reference can be error-prone – if the returned reference is to an object whose lifetime ends when the function terminates, the result is a runtime error. In particular, if you write

```
int& foo ( )
{
    int temp = 1;
    return temp;
}
```

then your function is returning a reference to `temp`, which is destroyed when the function terminates.

Usually you return a reference when you are implementing a member function of a class, and the reference is to a private data member of the class or to the object itself.

3.4 Constructors, Destructors, Copy Constructors, and Copy Assignment Constructors

3.4.1 Default Constructor

A ***default constructor*** is a constructor that can be called with no arguments. If a class does not have a user-supplied constructor of any kind, the compiler tries to generate a default constructor at compile time. If it has any kind of constructor, the compiler will not do this.

3.4.2 Destructor

A ***destructor*** is called when an object goes out of scope or is deleted explicitly by a call to delete. The compiler supplies destructors if the user does not. The reason to supply a destructor is to remove memory allocated by a call to `new`, to close open files, and so on. The destructor created by the compiler will be a ***shallow destructor***, meaning that it simply deletes the actual members of the class, and not any memory that members may point to, directly or indirectly.

3.4.3 Copy Constructor

A ***copy constructor*** is called in the following situations:

1. When an object needs to be constructed because of a declaration with initialization from another object of a type that can be converted to the object's class, as in

   ```
   IntCell C;
   IntCell B = C; // called here
   IntCell B(C);  // called here
   ```

   but not

   ```
   B = C;
   ```

   because `B` already has been constructed, so this is not a constructor call of any kind.

2. When an object is passed by value to a function.

3. When an object is returned by value from a function. If an object is returned by reference, then it is not copied. If by value, the object to be returned is constructed as a copy of the object within the function.

Again, *C++11* has made things more complex, by the inclusion of the ***move*** operator and ***call by rvalue-reference parameters***. It defines another type of constructor called a ***move constructor***. This is mentioned only briefly here. The move operation is what it sounds like – unlike an assignment such as `x = y`, which *copies* the value of `y` into `x`, the assignment `x = std::move(y)` does not perform a copy, but instead gives `x` the value stored in `y` and removes the value stored in `y`; it moves it from `y` to `x`.

3.4.4 Copy Assignment Operator

The copy assignment operator is called when two objects already exist and one is being assigned to the other. In *C++* it is `operator=`. Again, *C++11* has increased the complexity, as there are two different assignment operators, the ***copy assignment operator*** and the ***move assignment operator***. The copy assignment operator is called when the right hand side of the assignment is a lvalue, i.e., the name of an object. The move assignment operator is called in *C++11* when the right hand side is a temporary object that is about to be destroyed.

3.4.5 Using Defaults or Not

In general, you should either declare no destructors or constructors or assignment operators of any kind, or define all of them. If your data members include pointers, in general you should define all of these functions. Even if it does not include pointers, then whether or not you need to depends on whether any of the conditions described in Section 3.4.3 are true and you need to implement a copy constructor. *C++* does a great deal for you, but in turn you must understand its complex semantics if you are to avoid hard-to-diagnose bugs. This is why it is best to follow the simple rule of either all-or-nothing when it comes to these functions.

4 Templates

One difference between *C* and *C++* is that *C++* allows you to define templates for both classes and functions. It is easier to understand class templates if you first understand function templates.

Suppose that in the course of writing many, many programs, you find that you need a swap function here and there. Sometimes you want a swap function that can swap two integers, sometimes you want one that can swap two doubles, and sometimes you need to swap objects of a class. In *C*, you have to write a swap function for each type of object, or you can reduce the work by writing something like this[3]:

```
typedef int elementType;

void swap ( elementType *x, elementType *y)
{
    elementType temp = *x;
    *x = *y;
    *y = temp;
}
```

and you would call this with a call such as

```
int a, b;
a = ... ; b = ... ;
swap(&a, &b);
```

In *C*, the parameters need to be pointers to the variables to be swapped, and their address must be passed. If you wanted to swap doubles, you would change the `typedef` by replacing the word "`int`" by "`double`."

In *C++*, you could do the same thing using reference parameters:

```
typedef int elementType;

void swap ( elementType &x, elementType &y)
{
    elementType temp = x;
    x = y;
    y = temp;
}
```

and you could call this with code such as

```
int a, b;
a = ... ; b = ... ;
swap(a, b);
```

Although you do not have to write a separate swap function for each different element type, it is inconvenient. The *C++* language introduced function templates as a way to avoid this.

[3]There are other methods as well, but these are the two principal approaches.

4.1 Function Templates

A ***function template*** is a template for a function. It is not an actual function, but a template from which the compiler can create a function if and when it sees the need to create one. This will be clarified shortly. A template for the swap function would look like this:

```
template <class elementType>
void swap ( elementType &x, elementType &y)
{
    elementType temp = x;
    x = y;
    y = temp;
}
```

The word *class* in the template syntax has nothing to do with classes in the usual sense. It is just a synonym for the word *type*. All types in *C++* are classes. The syntax of a (single-parameter) function template definition is

```
template <class type_parameter> function-definition
```

where *function-definition* is replaced by the body of the function, as `swap()` above demonstrates. The syntax for a (single-parameter) function template declaration (i.e., prototype) is

```
template <class type_parameter > function-declaration
```

You need to repeat the line

```
template <class type_parameter>
```

before both the declaration and the definition. For example:

```
// Declare the function template prototype
template <class T>
void swap( T & x, T & y );

int main()
{
     int n= 5;
     int m= 8;
     char ch1 = 'a', ch2 = 'b';
     // more stuff here
     swap(n,m);
     swap(ch1, ch2);
     // ...
}
```

```
// Define the function template declared above
template <class T>
void swap( T & x, T & y )
{
    T temp = x;
    x = y;
    y = temp;
}
```

You will often see just the letter "T" used as the type parameter in the template.

When the compiler compiles the main program above, it sees the first call to a function named `swap`. It is at that point that it has to create an instance of a function from the template. It infers from the types of its parameters that the type of the template's parameter is `int`, and it creates a function from the template, replacing the type parameter by `int`. When it sees the next call, it creates a second instance whose type is `char`.

Because function templates are not functions, but just templates from which the compiler can create functions, there is a bit of a problem with projects that are in multiple files. If you want to put the function prototype in a header file and the function definition in a separate `.cpp` file, the compiler will not be able to compile code for it in the usual way if you use that function in a program. To demonstrate, suppose that we create a header file with our swap function prototype, an implementation file with the definition, and a main program that calls the function.

This is `swap.h`:

```
#ifndef SWAP_H
#define SWAP_H

template <class T>
void swap( T &x, T &y);
#endif
```

and `swap.cpp`:

```
template <class T>
void swap( T &x, T &y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

and `main.cpp`:

```
#include "swap.h"
int main ()
{
        int a = 10, b = 5;
        swap(a,b);
        return 0;
    }
```

When we run the command

```
g++ -o demo swap.cpp main.cpp
```

we will see the error message

```
/tmp/ccriQBJX.o: In function ‘main’:
main.cpp:(.text+0x29): undefined reference to ‘void swap<int>(int&, int&)’
collect2: ld returned 1 exit status
```

This is because the function named `swap` does not really exist when `main` is compiled. It has a reference only to a function template. The solution is to put the function template implementation into the header file, as unsatisfying as that is because it breaks the wall that separates interface and implementation. This can be accomplished with an `#include` directive:

```
#ifndef SWAP_H
#define SWAP_H

template <class T>
void swap( T &x, T &y);

#include “swap.cpp”
#endif
```

The general rule then, is to put the function template prototypes into the header file, and at the bottom, include the implementation files using an `#include` directive. There will be no problem with multiply-defined symbols in this case when you compile the code.

Note. Function templates, and templates in general, can have multiple parameters, and they do not have to be classes, but that is a topic beyond the scope of this introduction. You may also see the word `typename` used in place of the word `class` as the type of the template parameter. For the most part, these are interchangeable, but it is better to use `class` until you know the subtle difference. The interested reader can refer to a good C++ book for the details.

4.2 Class Templates

Imagine that you want to implement a list class. There is nothing in the description of a list that is specific to any particular type of data object, other than the ability to copy objects. For a sorted list, the objects do have to be comparable to each other in some linear ordering, but that is about the only limitation in terms of the data's specific properties. It stands to reason that you should be able to create a generic kind of list, one whose definition does not depend on the underlying element type. This is one reason that *C++* allows you to create templates for classes as well. A class template is like a generic description of that class that can be instantiated with different underlying data types.

4.2.1 Defining Class Templates

As with function templates, a *C++* class template is not a class, but a *template for a class*. An example of a simple class template interface is

```
template <class T>
class Container
{
public:
    Container();
    Container( T initial_data);
    void set( T new_data);
    T get() const;
private:
    T mydata;
};
```

Notice that a class template begins with the `template` keyword and template parameter list, after which the class definition looks the same as an ordinary class definition. The only difference is that it uses the type parameter `T` from the template's parameter list. The syntax for the implementations of the class template member functions when they are outside of the interface is a bit more complex. The above functions would have to be defined as follows:

```
template <class T>
void Container<T>::set ( T initial_data )
{
    mydata = new_data;
}

template <class T>
T Container<T>::get() const
{
    return mydata);
}
```

Notice the following:

1. Each member function is actually a function template definition.
2. All references to the class are to `Container<T>` and not just `Container`. Thus, the name of each member function must be preceded by `Container<T>::`.

In general the syntax for creating a class template is

```
template <class T>  class class_name {  class_definition  };
```

and a member function named `foo` would have a definition of the form

```
template <class T>
return_type class_name<T>::foo ( parameter_list ) { function definition }
```

4.2.2 Declaring Objects

To declare an object of a class that has been defined by a template requires, in the simplest case, using a syntax of the form

```
class_name<typename> object_name;
```

as in

```
Container<int>    int_container;
Container<double> double_container;
```

If the `Container` class template had a constructor with a single parameters, the declarations would instead be something like

```
Container<int>    int_container(1);
Container<double> double_container(1.0);
```

The following is a complete listing of a very simple program that uses a class template.

Listing 1: A program using a simple class template.

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

template <class T>
class MyClass
{
  public:
      MyClass(  T initial_value);
      void set( T x) ;
      T get( )    ;

  private:
      T val;
};

template < class T >
MyClass < T >:: MyClass (T initial_value)
{
    val = initial_value;
}

template < class T >
void MyClass < T >:: set (T x)
{
    val = x;
}

template < class T >
T MyClass < T >::get ()
```

```
{
    return val;
}

int main ()
{
    MyClass<int>      intobj(0);
    MyClass<double>   floatobj(1.2);

    cout << "intobj value = " << intobj.get()
         << " and floatobj value = "  << floatobj.get() << endl;
    intobj.set(1000);
    floatobj.set (0.12345);
    cout << "intobj value = " << intobj.get()
         << " and floatobj value = "  << floatobj.get() << endl;

    return 0;
}
```

Again, remember that *a class template is not an actual definition of a class, but of a template for a class.* Therefore, if you put the implementation of the class member functions in a separate implementation file, which you should, then you must put an `#include` directive at the bottom of the header file of the class template, including that implementation file. In addition, make sure that you do not add the implementation file to the project or compile it together with the main program. For example, if `myclass.h`, `myclass.cpp`, and `main.cpp` comprise the program code, with `myclass.h` being of the form

```
#ifndef MYCLASS_H
#define MYCLASS_H

// stuff here

#include “myclass.cpp”
#endif // MYCLASS_H
```

and if `main.cpp` includes `myclass.h`, then the command to compile the program must be

```
g++ -o myprogram main.cpp
```

not

```
g++ -o myclass.cpp main.cpp
```

because the former will cause errors like

```
myclass.cpp:4:6: error: redefinition of ‘void MyClass<T>::set(T)’
myclass.cc :4:6: error: ‘void MyClass<T>::set(T)’ previously declared here
```

This is because the compiler will compile the `.cpp` file twice! This is not a problem with function templates, but it is with classes, because classes are turned into objects.

4.3 Object and Comparable

The textbook makes use of `Object` and `Comparable` as generic types. The `Object` class represents any class with a default constructor, an `operator=`, and a copy constructor. The `Comparable` class is intended to represent any class that, in addition, has an `operator<`. When a class has an `operator<`, the collection of all of its instances is a totally ordered set; i.e., given any two `Comparables A` and `B`, either `A < B` or `B < A`.

4.4 Function Objects

A ***function object*** is a special type of object in *C++*. It is a class that contains a public overload of the ***function call operator***, `operator()`. A function object may be used instead of an ordinary function . In *C++* this is also called a ***class type functor***. An example or two will illustrate.

A simple function object can look like this:

```
class is_less_than
{
public:
    bool operator()(const int &a, const int &b) const
    {
        return a < b;
    }
};
```

Note that the name of this class uses the function name style rather than the *PascalCase* class naming convention. This is a class with nothing but the overloaded `operator()`. It has two parameters and compares their values and returns true if the first is less than the second, and false otherwise. It can be called like this:

```
int x = 10, y = 7;
if ( is_less_than(x,y) )
    /* more stuff */
```

The call is indistinguishable from a call to a function named `is_less_than`. We can pass a function object, not an instance of the function, but the class name itself, to a routine that expects a function pointer. Listing 2 demonstrates.

Listing 2: Function object example 1.

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

/* The function object is named is_less_than */
class is_less_than
{
public:
    bool operator()(const int &a, const int &b) const
    {
        return a < b;
    }
};


int main()
{
    std::vector<int> items;

    /* put numbers in reverse order into the vector */
    for ( int i = 10; i > 0; i--)
        items.push_back(i);
    /* The sort() algorithm from the standard algorithm library has the
       syntax
        void sort (RandomAccessIterator first,
                   RandomAccessIterator last,
                   Compare comp);
        where the first two parameters are iterators to the start and one
           past the end of the range of the container to be sorted, and
           the third parameter is a binary function that accepts two
           elements in the range as arguments, and returns a value
           convertible to bool. The third argument can be a function
           pointer or a function object.
    */

    std::sort(items.begin(), items.end(), is_less_than());

    /* prove it is sorted by printing it out */
    for ( int i = 0; i < 10; i++)
        std::cout << items[i] << " ";
    std::cout << std::endl;

    return 0;
}
```

Function objects can also have data members to maintain their state. There is no restriction about this. They can also have a constructor to initialize the data members. The `Ticker` function object below retains its state and can be used like the ticker in a store that makes you take a number to be served. It is named using *PascalCase* because it has a constructor and this also looks like an object, not just a function:

```
class Ticker {
private:
    int &count;
public:
    Ticker(int &n) : count(n) {}
    int operator()()
    {
        return count++;
    }
    };
```

The following listing shows how it could be used.

Listing 3: Function object with state.

```
#include <iostream>
#include <iterator>
#include <algorithm>

class Ticker {
private:
    int &count;
public:
    Ticker(int &n) : count(n) {}
    int operator()()
    {
        return count++;
    }
};

int main ()
{
    int numbers[20];
    int startvalue(10);

    std::generate_n (numbers, 20, Ticker(startvalue));

    std::cout << "Numbers given out today are:";
    for (int i=0; i<20; ++i)
        std::cout << ' ' << numbers[i];
    std::cout << '\n';

    return 0;
}
```