The goal of this course is to learn techniques by which you can write better programs, to solve harder problems, than the ones which you might have encountered in the introductory programming course. Specifically expect yourself to get better at the following:

1. Reasoning about programs. When you write a program, can you argue that it will produce the result that you want? Can you estimate the amount of time it will take to produce the result?

2. Representing objects on a computer. In the introductory programming course, you might have dealt only with numbers, numerical or other sequences, matrices. In this course you will learn ways to also represent real life objects such as road networks, math expressions, circuits etc.

3. Knowing (and to some extent designing) algorithms to solve more complex problems than the ones you encountered in the introductory programming course.

While the course is about programming, the main activity in the course will be thinking about programs. The main concerns will be to argue that a program runs correctly and also to estimate the time and space required by the program. We might also ask some fundamental questions such as whether an algorithm even exists for solving a problem quickly, where quickly will of course have to be suitably defined.

In the rest of this chapter we study how to argue that a program is correct. By the phrase "program is correct" we mean (a) that the program eventually terminates, and (b) by the time it terminates it produces the correct answer no matter what input it is given. Some programs might be based on very clever or complex ideas, and it might not be obvious that they should always work. In addition, you might also have programs whose basic idea is simple, but which might have been implemented with a "silly mistake". Indeed, errors in programs (conceptual or due to silly mistakes) have caused major accidents leading to loss of life as well as monetary losses. Thus proving correctness is very helpful. Note that when you write a program you will run many test cases and check that the program may work correctly on all those

– but that does not guarantee that the program will work correctly on all cases.

We will study two central ideas: *invariants* and *potential function arguments*. We will consider progressively harder programs to illustrate these idea.

1 Proofs and program correctness

A proof is a sequence of assertions, some of which are assumed to hold (axioms), and others which follow from previously stated assertions. For example, suppose you have proved assertion A, and another assertion which says "A implies B". Then from these assertions it is natural to infer the assertion B. As justification of this inference you might say "But that is what A implies B means!", or you might give the technical name for this kind of inference, *modus ponens*. Other kinds of claims and justifications are also possible, for example you might claim something and justify it using mathematical induction.

A proof of program correctness is similar, except that we will make claims about values taken by variables or values printed by the program. The usual justifications, e.g. *modus ponens*, mathematical induction, and so on will be available to us. But in addition, we will be able to also justify statements based on the definition of the programming language. For example, suppose we proved that in the ith step a certain integer variable x has value 5 and in the $i+1$th step, we are executing the statement `x++;`. Then we can conclude that after the $i+1$th step `x` will have value 6. The justification for this claim is our knowledge of C++. Indeed the manual for C++ will say that the operator `++` will cause an integer variable to increment by 1.

The general idea of the proof then, is to use our knowledge of the language, strictly speaking what is given in the language manual, to deduce the values taken by variables as the program executes, and also which loop tests or conditions in `if` statements succeed and so on.

We begin by applying this strategy to straight line code, i.e. a program in which there are no loops. Then we move on to loops.

2 Straight line code

Our first example program is rather trivial. Given an integer n representing a distance in inches, it is required to print 3 numbers y, f, i respectively denoting the distance in yards, feet and inches.

```
int n,i,f,y,w;  cin >> n; // 1
i = n % 12;               // 2
w = n / 12;               // 3
y = w / 3;                // 4
f = w % 3;                // 5
cout << y <<' '<< f <<' '<< i << endl;
```

Since there are no loops, this program is guaranteed to terminate, i.e. there is no question of it going into an infinite loop. So we need to only worry whether it prints the correct answer. Before we argue this, we need to specify what correct even means. This is not always easy to specify. In the present case, we might say that we require that (a) $36i + 12f + y = n$, and (b) $0 \leq i \leq 11$, $0 \leq f \leq 2$. Note that the requirements (b) are not stated explicitly in the problem statement above, but we might consider these to be *common sense.*

Next we make claims about what happens to the program variables as each statement executes. The main statements here are assignment statements, say `p = q;`. Such a statement causes the expression `q` to be evaluated and its value placed in the variable `p`, and the value of no variable besides `p` changes.[1]

Thus we can say that after statement 1, variable `n` must have the value n that was typed by the user. After statement 2, the value of `n` does not change but `i` gets the value $n \bmod 12$. After statement 3, we will know that the value of `n,i` will not have changed but `w` will get the value $\lfloor n/12 \rfloor$. After statement 4, we will have `y` get the value $\lfloor \lfloor n/12 \rfloor /3 \rfloor$, and after statement 5, the value of `f` will become $\lfloor n/12 \rfloor \bmod 3$. Now it is an algebraic exercise to show that the printed values $y = \lfloor \lfloor n/12 \rfloor /3 \rfloor, f = \lfloor n/12 \rfloor \bmod 3, i = n \bmod 12$ satisfy the correctness requirements (a), (b).[2]

[1]We assume here that the expression `q` only contains simple arithmetic operators, i.e. there are no operators or function calls that cause side-effects. If the expression contains operators such as `++` which produce a value as well as modify their operand, or if the function calls not only return values but modify variables in the calling program, we will need to be more careful.

[2]You may need the identities $\lfloor \lfloor p/q \rfloor /r \rfloor = \lfloor p/(qr) \rfloor$ and $(p \bmod q) + \lfloor p/q \rfloor * q = p$ for integers p, q, r.

The general idea was to determine the values of the variables *symbolically*, in terms of the values typed in by the user. As we will see, we might also assert relationships between values taken by variables rather than specify the value taken by a single variable.

3 Loop Example 1

We consider the problem of adding up the first n terms of the series for $\sin(x)$:

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Here is a possible program fragment for this.

```
int n; cin >> n;
double x; cin >> x;
double sum = 0, term = x;
i=1;
while(i <= n){
  sum = sum + term;
  term = - term * x * x / (2*i*(2*i+1));
  i++;
}
cout << sum;
```

Once you have a loop in your program, there is the potential that the program does not halt: the loop termination condition may never be met!

Of course, in this program it is easy to see that the loop terminates. the variable `i` starts at 1, increments by 1 in each iteration, and goes up to `n` which does not change in the loop. Thus the loop executes exactly `n` iterations.

Can we be sure that the correct value is calculated? The key observation is the following:

> When control reaches the beginning of the loop body `term` = `i`th term of the series = $(-1)^{i-1}x^{2i-1}/(2i-1)!$ `sum` = sum of first `i-1` terms of the series.

Here we are using `i` and i to both denote the value of the variable at the beginning of the loop body. We will ignore the distinction if our intent is clear from the context.

It might seem that the claim does not say anything about `sum` at the end of the program. But before the loop is exited, control does indeed reach the top of the loop to make the check. At this time `i` has the value $n+1$. Thus the claim (with $n+1$ substituted for `i`) says: `sum` $=$ sum of the first n terms. So if we can prove the claim in general, we would have proved the correctness of the program.

The claim is proved by induction on the number of times the loop is visited. Specifically, note that the first time the loop is visited, `i=1, sum=0, term=x` from the initialization before the loop. The claim with 1 substituted for `i`, says that `sum` must equal the sum of the first `i-1=0` terms of the series, and it indeed does. The claim also says that `term` must equal the first term x of the series, and this is indeed the case.

Now suppose that the claim is true for a certain visit. At the beginning of that iteration, let i denote the value of the variable `i`. Then `sum` equals the sum of the first $i-1$ terms of the series, and `term` equals the `i`th term. Note that the `i`th term is $(-1)^{i-1}\frac{x^{2i-1}}{(2i-1)!}$. The first statement will thus cause `sum` to become equal to the sum of the first `i` terms of the series. The next statement will cause term to equal the $(-1)^{i-1}\frac{x^{2i-1}}{(2i-1)!}\frac{-x^2}{(2i)(2i+1)} = (-1)^i\frac{x^{2i+1}}{(2i+1)!}$, i.e. the value of the $i+1$th term of the series. Note that the last statement in the loop causes the value of `i` to increase by 1, i.e. become $i+1$. But because of this update, at the end of the body the value of `sum` is indeed the sum of the first `i-1` terms and the value of `term` the `i`th term (for the new value of `i`). But these values do not change when control goes to the top of the loop for the next iteration. Thus the claim is true at the next visit also. Thus proved.

3.1 Remarks

The claim in the comment applies to all iterations of the loop, and is therefore called a `loop invariant`.

In proving the invariant, we made (inductive) assumptions about the values of a variable at the beginning of the iteration and using these deduced the values at the end of the loop (which is equivalently the beginning of the next iteration). Note that the loop body itself was straight line code, so the core of the reasoning in some sense similar to that in Section 2.

Writing down the invariant to be proved (which implies program correctness) is the crux of the proof. The rest of the proof is routine and a bit laborious, and in this course, you will only be expected to write down the (correct and complete) invariant.

Writing down the invariant is far from easy in geneal. In the above case, we might have made a claim only about the value of `sum` without talking about the value of `term`. Such a claim would be correct but we would not be able to prove it on its own – the induction used above requires us to also know how `term` changes in each iteration. Thus the complete invariant would have to also talk about what values `term` takes, as was done in above.

We also note that it is possible to get the above program wrong, e.g. you might exchange the order of the updates to `sum` and `term`, or instead of dividing by `2*i*(2*i+1)` you might divide by `(2*i+1)*(2*i+2)`. But you would not make such a mistake if you carefully write down the invariant either before writing the code or after. The invariant would either prevent the error from happening or alert you to the error.

4 A more involved loop

We consider Euclid's algorithm for finding the Greatest Common Divisor of two positive integers x, y.

```
int x,y; cin >> x >> y;
while(x % y != 0){
  int newy = x % y;
  int newx = y;
  x = newx;
  y = newy;
}
cout << y << endl;
```

Claim: Every time the program checks the loop condition, the GCD of `x,y` equals the GCD of the original values typed in by the user into the variables `x,y`. Also, at the time of the check `x,y > 0`.

Observe first that if this claim is true, and if the program did halt, then it must have printed the correct value. For this, note that before printing `y` the program must have checked the loop condition and found it false, i.e. it must have found that `x` is a multiple of `y`. Thus we know that `y` must be the

GCD of the values of `x,y` at the time of the loop test. But the claim assures us that this must also be the GCD of the original values of `x,y`.
Proof of claim: The proof is by induction on the number of times the loop is tested. Clearly the claim is trivially true on the first test: the variables `x,y` contain the values typed by the user. Also, we are expecting that the user will type positive values.

So suppose that the claim is true just before the ith loop test. If the loop test fails we are done because we have proved the claim for all loop tests that happen. Suppose that the loop test succeeds, i.e. `x % y != 0`. So we now enter the loop and assign new values to `x,y`. By inspecting the loop body, we can see that the `newx,newy` equal `y,x%y`. Note that given that `x,y > 0`, and `x% y != 0`, `newx,newy > 0`.

We know by the following Lemma that `y,x%y` have the same GCD as `x,y`. But at the end of the loop `x,y` get the values of `newx,newy`. Thus the GCD of `x,y` does not change even though the values of `x,y` do. ■

Euclid's Lemma: For integers $x, y > 0$ if $x \bmod y = 0$ then $GCD(x, y) = y$. Otherwise $GCD(x, y) = GCD(y, x \bmod y)$.
Proof: If $x \bmod y = 0$, clearly y is ght GCD. Otherwise, Suppose d is any common divisor of x, y. Then $x = ad, y = bd$ for integers a, b. We know that $x \bmod y = x - ky$ for some k. Thus $x \bmod y = (a - kb)d$. Thus d is a divisor of $x \bmod y$.

Suppose instead that e is any common divisor of $y, z = x \bmod y$. Thus $y = pe, z = qe$. But $x = z + ky$ for some k. Thus $x = (q + kp)e$, i.e. every common divisor of $y, z = x \bmod y$ also divides x, y.

Thus the greatest common divisor of x, y must also be equal that of $y, x \bmod y$. ■

Finally we will prove that the program in fact stops and does not go into an infinite loop.
Claim: The program terminates.
Proof: If the loop test fails and the body is executed, then the value of the variable `y` reduces because the new value, `x%y` must always be smaller than `y` and also non negative. If the loop test were to never succeed, then `y` would have to eventually be negative. Since this does not happen the loop test must eventually succeed. Thus the program terminates after a finite number

of iterations. ■.

In this, the value of `y` sometimes called a *potential*, in analogy to the potential energy of a physical system which somehow characterizes its behavious and must decrease presumably until it drops down to 0.

4.1 Remarks

Note that we don't really need two variables `newx, newy`, we could have directly written `int temp=y; y = x % y; x = temp;`. Using two variables makes the logic more obvious, and is therefore recommended.

5 Correctness of recursive programs

We will consider the recursive expression of the GCD algorithm and prove its correctness.

```
int gcd(int x, int y){
  if(x % y == 0) return y;
  return gcd(y, x % y);
}
```

To argue correctness, we must first clearly state what we expect `gcd` to do.

> For `x,y>0`, `gcd(x,y)` will return the greatest common divisor of `x,y`.

Next we use induction on a carefully chosen parameter. In the present case, the second argument is convenient.
Base case: `gcd` works correctly when second argument `y = 1`.
Proof: The GCD of any number and 1 is 1. The function will find `x % 1 == 0` for any `x`, and will thus return 1. ■

Induction step: Suppose `gcd` correctly works when the second argument `y` has value larger than any number n. Then it works correctly when `y` has the value n.

Proof: If `x % y == 0` then the GCD must be `y`, which is indeed returned. Else the value of the function call `gcd(y, x % y)` is returned. Clearly the second argument of this call lies between 1 and $n-1$. Thus by the induction hypothesis this call correctly returns the value of $GCD(y, x \bmod y)$. But by Euclid's Lemma this equals $GCD(x, y)$. ■

6 Concluding remarks

The description of a programming language describes what happens when each statement of the language executes. As an example, the manual might say "For an integer variable `x` the statement `x++;` will cause the value of `x` to increase by 1.". We can take such descriptions as axiomatic, and use them in proofs of program correctness. The implications of such claims can be composed to deduce what happens to variables as the program executes.

It may be observed that in a sense a proof restates the logic we might have had when we designed the program. This is correct. However, when we write a proof, we are forced to check that we cover all possibilities. We may have the correct idea but we might make a silly mistake when we write the program. For example in the $\sin(x)$ calculation program we might exchange the order of updates to `sum, term, i`, or we might divide by $2i+1, 2i+2$ instead of by $2i, 2i+1$. Such mistakes will be caught if we write a proof of correctness. Of course, we can also make a mistake in writing the proof of correctness, but at least we get a second chance at discovering our mistake.

It is a bit cumbersome to write a full proof of correctness. It is recommended however, that you at least write the loop invariant completely and precisely.

Exercises

1. Modify the program for computing $\sin(x)$ to be consistent with the following invariant: The value of `sum,term` equals the sum of the first `i` terms and the `i`th term respectively.

2. Write the code for computing $\sin^{-1}(x)$ using the associated Taylor series. State the invariant.

3. Write a program that reads an integer n and prints it in binary, from the least significant bit to the most. Prove its correctness. Note that the loop invariant will have to include statements about what has already been printed. Thus you might say "On any visit the numbers printed comprise the least significant bits of n and ...". You will need to state and prove something like "The binary representation of n consists of n mod 2 followed by the binary representation of $\lfloor n/2 \rfloor$."

4. Write the above program recursively and argue its correctness.

5. Write a program that reads in what the user types, one character at a time, and stops if the user types the sequence of letters "kitkat" in succession.

   Your program must have a variable `matched`. As you are about to read the next letter, `matched` should equal the size of the partial match you have obtained till then. As an example, suppose the letters read till then were "abckiqkit" then the partial match that can potentially extend consists of the 3 letters "kit". Thus `matched` should equal 3. Use this as the invariant to design your program.