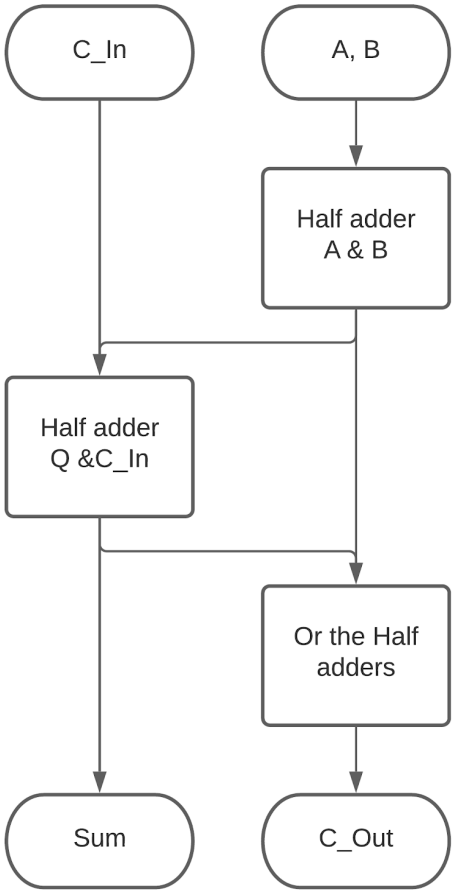
Digital Objects

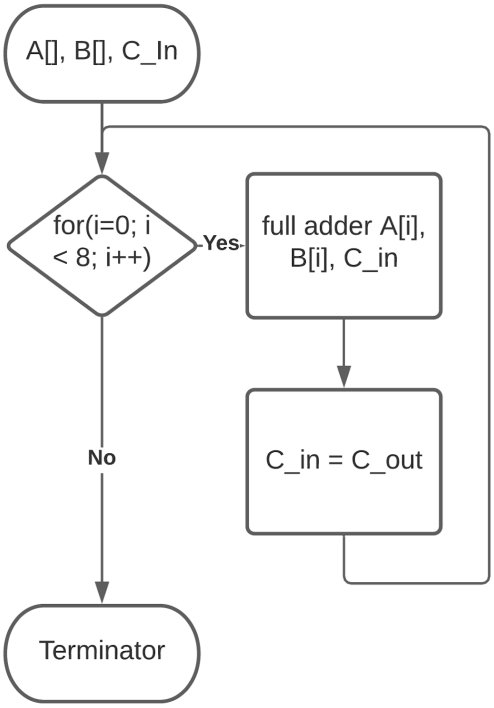
Preparation - 02/03/21

1) Describe the overall structure of your program.

My program calls the full adder once each time for the size of the register, currently set at 8. This full adder calls the half adder twice, and uses an OR gate on the c_outs. The half adder uses the AND and XOR functions.



Full Adder



8-bit ripple counter

My modified programs will use class based operations to solve the AND, OR, and XOR operations. This is because they have similar structures so can be initialised easily as derived classes.

2) Define your gate classes using inheritance including any destructors you might need.

```
class gate
{
public:
    gate() = default;
    gate(bool in_A, bool in_B)
    {
        A = in_A;
        B = in_B;
        Q = 0;
    }
    bool op()
    {
        return Q;
    }

private:
    bool A;
    bool B;
    bool Q;
};

class AND : public gate
{
public:
    AND() = default;
    AND(bool in_A, bool in_B)
    {
        A = in_A;
        B = in_B;
        Q = in_A && in_B;
    }

private:
    bool A;
    bool B;
    bool Q;
};
```

```
class OR : public gate
{
public:
    OR() = default;
    OR(bool in_A, bool in_B)
    {
        A = in_A;
        B = in_B;
        Q = in_A || in_B;
    }

private:
    bool A;
    bool B;
    bool Q;
};
class XOR : public gate
{
public:
    XOR() = default;
    XOR(bool in_A, bool in_B)
    {
        A = in_A;
        B = in_B;
        Q = in_A ^ in_B;
    }

private:
    bool A;
    bool B;
    bool Q;
};
```

3) In general when should you use delete and delete[]

The delete[] operator is used to delete arrays, so should be used on arrays and when pointers are being used. The delete function should be used by default in any other instance.

Derived Classes - 03/03/12

To implement the derived classes I declared the classes and their member functions in the header file. I also defined the classes in the main program, The classes were easy to implement as the constructor creates a variable with that classes result.

```
class gate
{
    public:
    gate() = default;
    gate(bool in_A, bool in_B);

    bool op();  //returns Q

    private:
    bool A;
    bool B;
    bool Q;
};

class AND : public gate
{
    public:
    AND() = default;
    AND(bool in_A, bool in_B);

    private:
    bool A;
    bool B;
    bool Q;
};
```

```
gate::gate(bool in_A, bool in_B)
{
    A = in_A;
    B = in_B;
    Q = 0;
}
bool gate::op()
{
    return Q;
}

AND::AND(bool in_A, bool in_B)
{
    A = in_A;
    B = in_B;
    Q = in_A && in_B;
}
```

To implement this code in the main body of my code I had to modify the code for the half adder and full adder functions. I changed it so they initialised a variable of the gate class and returned the value of the operation.

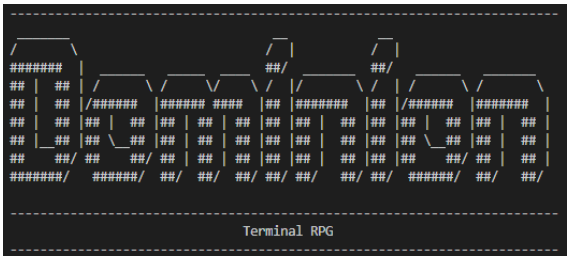
```
sum half_adder(bool A, bool B)
{
    sum result;
    XOR Q = XOR(A, B);
    AND c_out = AND(A, B);

    result.Q = Q.op();
    result.c_out = c_out.op();

    return result;
}
sum full_adder(bool A, bool B, bool c_in)
{
    sum fa;

    //half add A & B
    sum ha_1 = half_adder(A, B);
    //half add Q & c_in
    sum ha_2 = half_adder(ha_1.Q, c_in);
    //set the output
    OR c_out = OR(ha_1.c_out, ha_2.c_out);

    fa.Q = ha_2.Q;
    fa.c_out = c_out.op();

    return fa;
}
```

This code worked for my adder/subtractor for combinations of addition and subtraction.

```
Input your first number (A): 12
Your corresponding value A in binary is: 00001100
Input your second number (B): 6
Your corresponding value B in binary is: 00000110
Input your value for C_In: 0
Select operation ( + or - ):-

Sum is: 6
Overflow bit is: 1

Input your first number (A): -40
Your corresponding value A in binary is: 11011000
Input your second number (B): 30
Your corresponding value B in binary is: 00011110
Input your value for C_In: 0
Select operation ( + or - ):-

Sum is: -70
Overflow bit is: 1

Input your first number (A): -40
Your corresponding value A in binary is: 11011000
Input your second number (B): -30
Your corresponding value B in binary is: 11100010
Input your value for C_In: 0
Select operation ( + or - ):-
-40--30

Sum is: -10
Overflow bit is: 0
```

Dominion - 03/03/12

To begin with I created a cool printheader. As we all know a game is only as good as its title screen.



I implemented member functions to access and modify the strength and hit point variables.

```
int Creature::getStrength()
{
    return strength;
}
void Creature::modifyStrength(int strengthmodify)
{
    strength += strengthmodify;
}

int Creature::getHitPoints()
{
    return hitpoints;
}
void Creature::modifyHitPoints(int hitpointsmodify)
{
    hitpoints += hitpointsmodify;
}
```

Then for each necessary class I modified the getDamage() function. This means instead of a set of if statements the function is different for any derived class with a modifier.

```
int Demon::getDamage()
{
    int damage;
    // All creatures inflict damage, which is a
    // random number up to their strength
    damage = (rand() % strength) + 1;
    std::cout << getSpecies() << " attacks for " << damage << " points!" <<
std::endl;
    // Demons can inflict damage of 50 with a 5% chance
    if ((rand() % 100) < 5)
    {
        damage = damage + 50;
        std::cout << "Demonic attack inflicts 50 "
                  << " additional damage points!" << std::endl;
    }

    return damage;
}
int Elf::getDamage()
{
    int damage;
    // All creatures inflict damage, which is a
    // random number up to their strength
    damage = (rand() % strength) + 1;
    std::cout << getSpecies() << " attacks for " << damage << " points!" <<
std::endl;
    // Elves inflict double magical damage with a 10% chance

    if ((rand() % 10) == 0)
    {
        std::cout << "Magical attack inflicts " << damage << " additional damage
points!" << std::endl;
        damage = damage * 2;
    }

    return damage;
}
```

```
int Balrog::getDamage()
{
    int damage;
    // All creatures inflict damage, which is a
    // random number up to their strength
    damage = (rand() % strength) + 1;
    std::cout << getSpecies() << " attacks for " << damage << " points!" <<
std::endl;
    // Demons can inflict damage of 50 with a 5% chance
    if ((rand() % 100) < 5)
    {
        damage = damage + 50;
        std::cout << "Demonic attack inflicts 50 "
                  << " additional damage points!" << std::endl;
    }
    // Balrogs are so fast they get to attack twice

    int damage2 = (rand() % strength) + 1;
    std::cout << "Balrog speed attack inflicts " << damage2 << " additional
damage points!" << std::endl;
    damage = damage + damage2;

    return damage;
}
```

Unfortunately this where I ran out of time and was unable to create functions for character creation and random encounters.