The *for* loop

Loop patterns : loop until done

```
#include <iostream>

using namespace std;

int main()
{
    int  x, y;

    cin >> x >> y;

    int r=0;

    while(x){
        r = r + y;
        x = x - 1;
    }

    cout << r << endl;
}
```

Loop patterns : loop until done

```
#include <iostream>

using namespace std;

int main()
{
    int  x, y;

    cin >> x >> y;

    int r=0;

    while(x){
        r = r + y;
        x = x - 1;
    }

    cout << r << endl;
}
```

```
int  x, y;

cin >> x >> y;

int r=0;

while(x){
    r = r + y;
    x = x - 1;
}

cout << r << endl;
```

Loop until done

```
int  x, y;

cin >> x >> y;

int r=0;

while(x){
    r = r + y;
    x = x - 1;
}

cout << r << endl;
```

```
int  x, y;

cin >> x >> y;

int r=0;

if(x){
    r = r + y;
    x = x - 1;
}
if(x){
    r = r + y;
    x = x - 1;
}
if(x){
    r = r + y;
    x = x - 1;
}
//  more ...

cout << r << endl;
```

Loop over a range of integers

```
int  x, y;

cin >> x >> y;

int r=0;

while(x){
    r = r + y;
    x = x - 1;
}

cout << r << endl;
```

```
int  x, y;

cin >> x >> y;

int r=0;
int i=0;
while( i < x ){
    r = r + y;
    i = i + 1;
}

cout << r << endl;
```

The `for` loop

```
int  x, y;

cin >> x >> y;

int r=0;

for(int i=0; i<x; i=i+1 ){
    r = r + y;

}

cout << r << endl;
```

```
int  x, y;

cin >> x >> y;

int r=0;
int i=0;
while( i < x ){
    r = r + y;
    i = i + 1;
}

cout << r << endl;
```

The `for` loop

```
int  x, y;

cin >> x >> y;

int r=0;

for(int i=0; i<x; i=i+1 ){
    r = r + y;

}

cout << r << endl;
```

$$r = \sum_{i=0}^{x-1} y$$

The `for` loop

```
int  x, y;

cin >> x >> y;

int r=0;

for(int i=0; i<=x; i=i+1 ){
    r = r + y;

}

cout << r << endl;
```

$$r = \sum_{i=0}^{x} y$$

Half-open ranges

- Algorithms often rely on integer ranges
- Each range can be open, half-open, or closed
  - Open : (a , b) = a+1, a+2, …, b-2, b-1
  - Half-open : [a , b) = a, a+1, a+2, …, b-2, b-1
  - Closed : [a , b] = a, a+1, a+2, …, b-2, b-1, b

- Half-open ranges are common : `[begin,end)`
  - First value is `begin`
  - Last value is `end-1`

Half-open vs closed iteration

[i, N)

```
int r=0;
for(int i=0; i<N; i=i+1 ){
    r = r + f(i);
}
```

$$r = \sum_{i=0}^{x-1} f(i)$$

```
int r=0;
for(int i=0; i<=N; i=i+1 ){
    r = r + f(i);
}
```

$$r = \sum_{i=0}^{x} f(i)$$

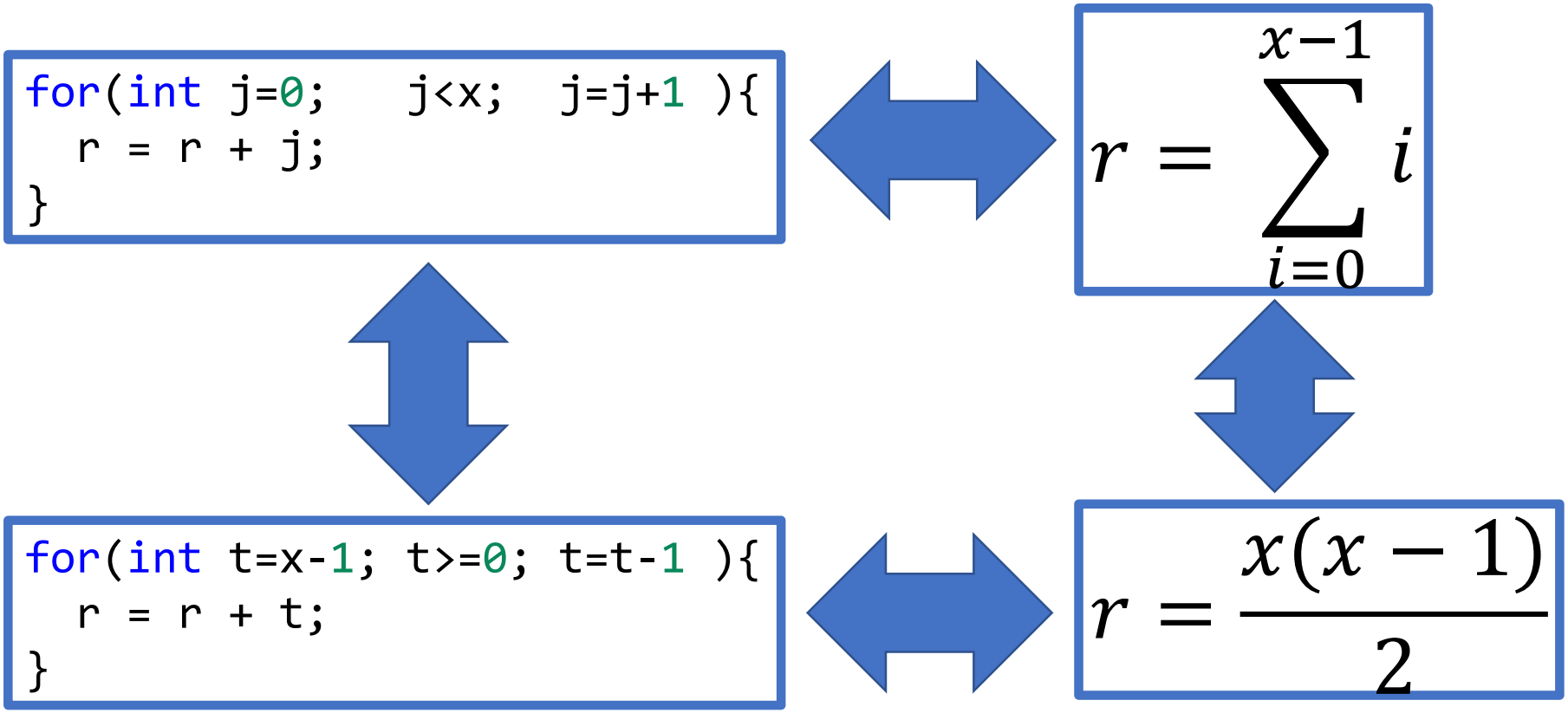
[i, N]

Counting down

```
int  x, y;

cin >> x >> y;

int r=0;

for(int i=x-1; i>=0; i=i-1 ){
    r = r + y;
}

cout << r << endl;
```

$$r = \sum_{i=0}^{x-1} y$$

Some loops are not order dependent



Special case : addition is commutative

$$a + b = b + a$$

Many loops *are* order dependent

```
for(int j=0;    j<x;  j=j+1 ){
  cout << j << endl;
}
```

```
for(int j=x-1; j>=0; j=j-1 ){
  cout << j << endl;
}
```

Many statements have *side-effects*

Perform externally visible input/output

Change internal state in non-linear way

Many loops *are* order dependent

```
for(int j=0;   j<N;  j=j+1 ){
  x = sin( x + j );
}
```

```
for(int j=N-1; j>=0; j=j-1 ){
  x = sin( x + j );
}
```

Many statements have *side-effects*

Perform externally visible input/output

Change internal state in non-linear way

Sequencing : algorithms vs maths

- Mathematical descriptions are unordered[1]
  - Symbols are assigned values once
  - Equations have no side-effects (no input/output)
  - You can do the numerical calculation in any order
- Algorithmic descriptions are ordered[2]
  - Variables can be changed over time
  - Statements can read/write to input/output
  - There is only one legal order for calculation

Maths : describes what the solution(s) should look like
Code : describes exactly how to calculate one solution

1 – Usually. Some pure mathematical notations incorporate state
2 – Usually. Some languages do not allow you to change variables

for loops as a special-case of **while**

```
for(int x=0; x<N; x=x+1){
    r = r + sin(x);
}
```

```
{
    int x=0;
    while( x<N ){
        r = r + sin(x);
        x=x+1;
    }
}
```

for loops as a special-case of **while**

```
for(INIT; COND; ITER){
    STATEMENT
}
```

?

```
{
    INIT;
    while( COND ){
        STATEMENT;
        ITER;
    }
}
```

for loops as a special-case of **while**

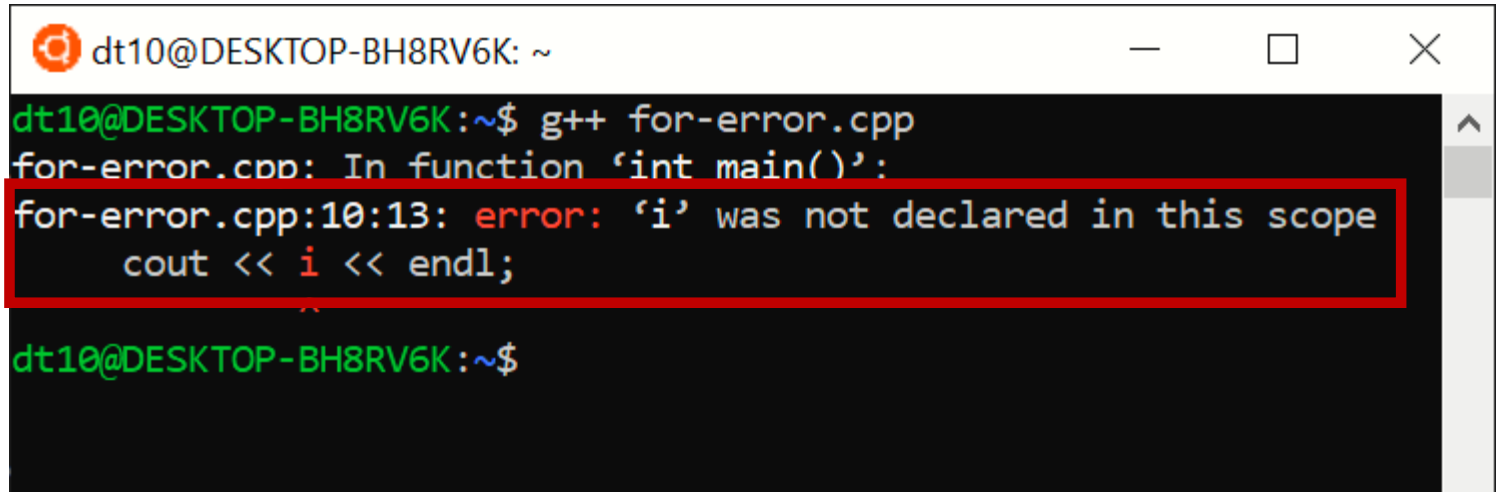
```
for(INIT; COND; ITER){
    STATEMENT
}
```

```
if(true){
    INIT;
    while( COND ){
        STATEMENT;
        ITER;
    }
}
```

Variable scope

```
int main()
{
    for(int i=0; i<10; i=i+1){
        cout << i << endl;
    }
    cout << i << endl;
}
```

```
int main()
{
    if(true){
        int i=0;
        while(i<10){
            cout<<i<<endl;
            i=i+1;
        }
    }
    cout << i << endl;
}
```



Scopes and lifetimes : brief intro

- Each variable has a *scope*
  - In what region of the program can I refer to the variable?
  - Each pair of curly brackets introduces a new scope
- Each variable has a *lifetime*
  - When does a variable come into existence?
  - How long is physical storage associated with the variable?

So far we have mainly scoped to the `main` function

- *Scope* : can use variable anywhere in main
- *Lifetime* : variables live for the entire execution

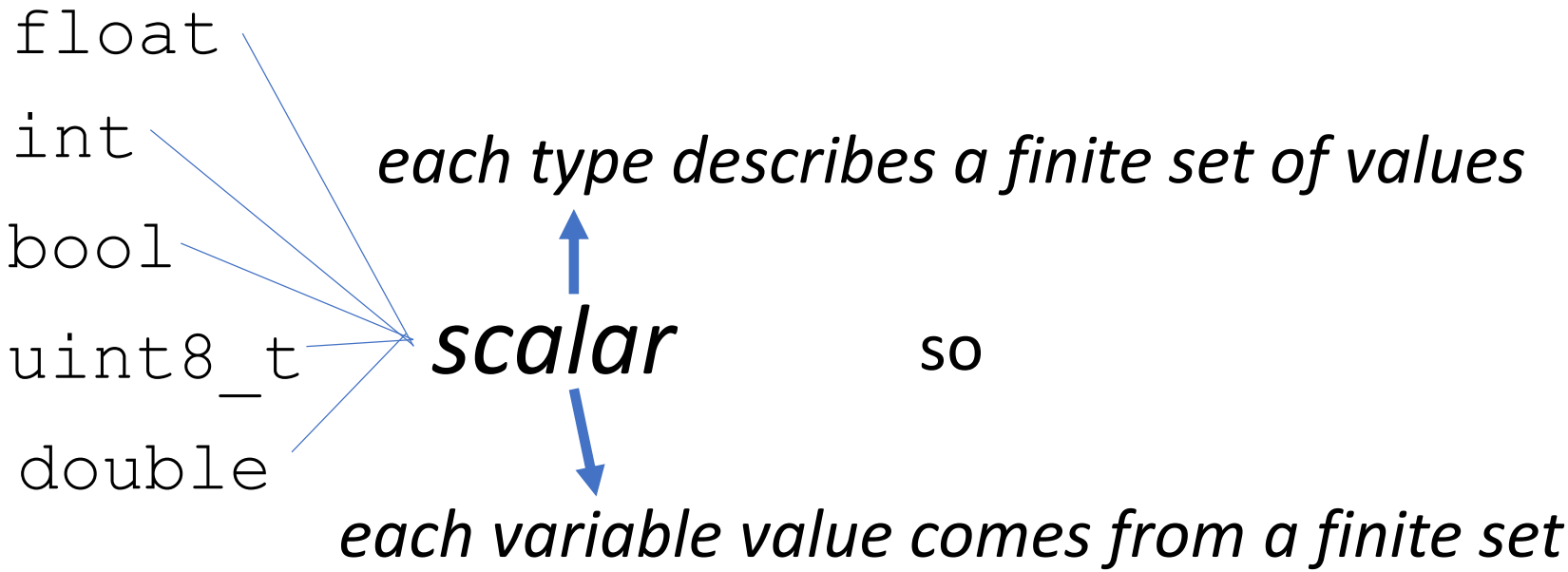
We'll investigate more when we look at functions

Sequence types

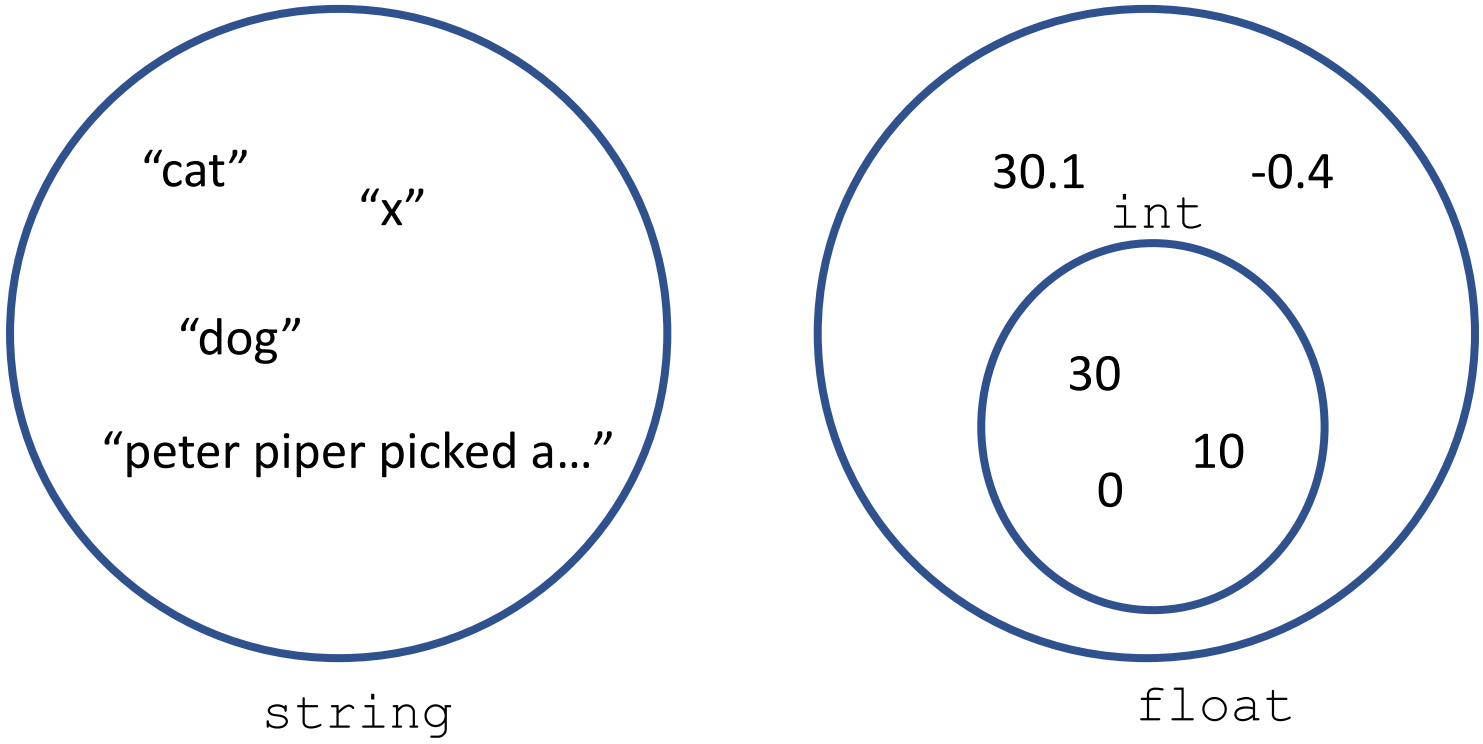
Scalar Types

*Each variable has a single **type***

and

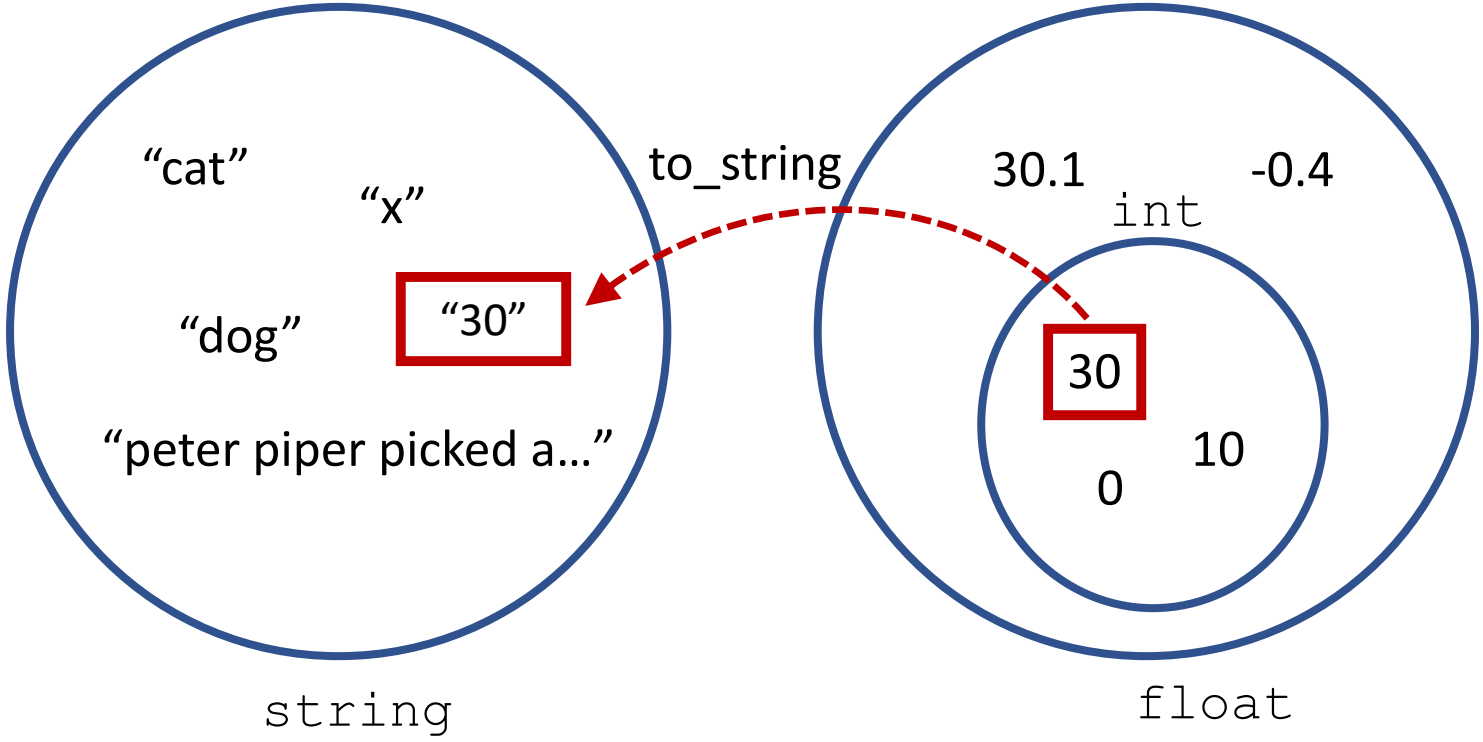


Strings versus numbers

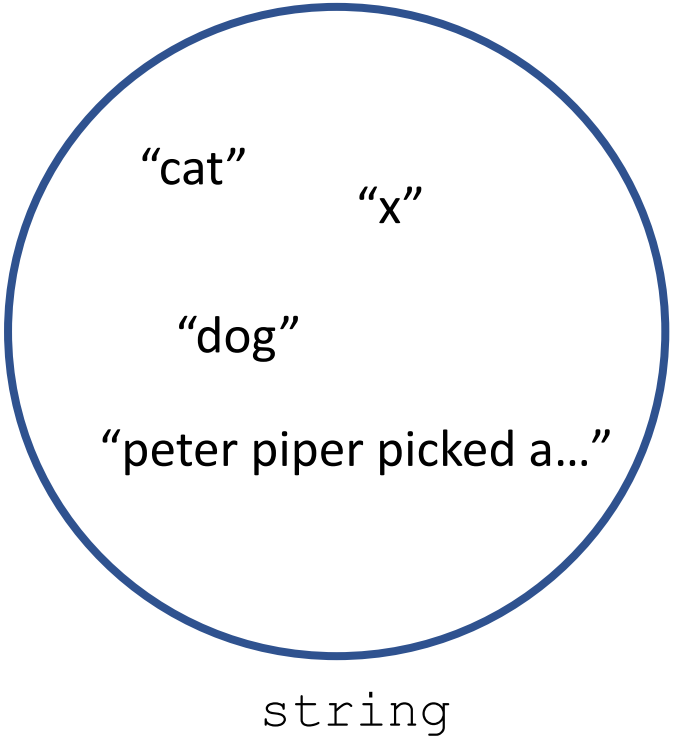


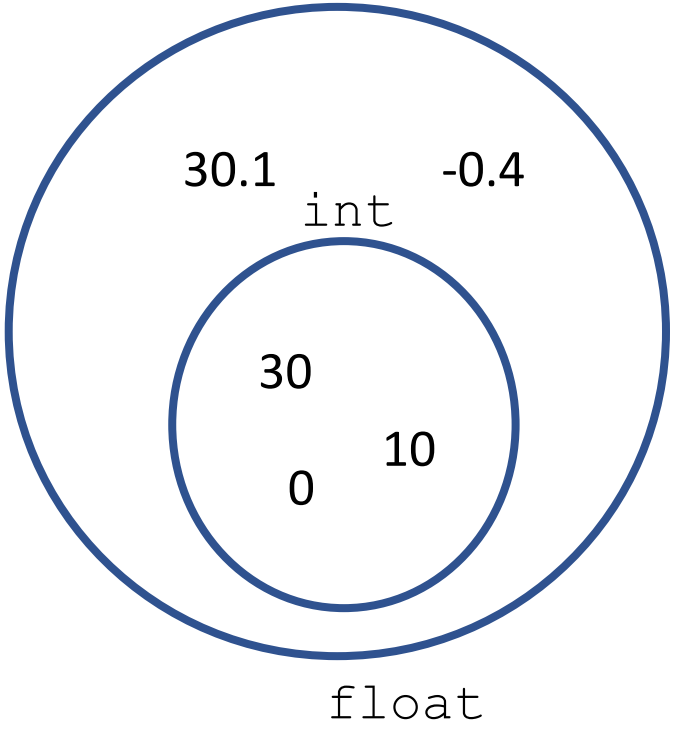
In reality int is not a subset of float,
but here we'll pretend it is

Mapping between types



How big is the set of strings?

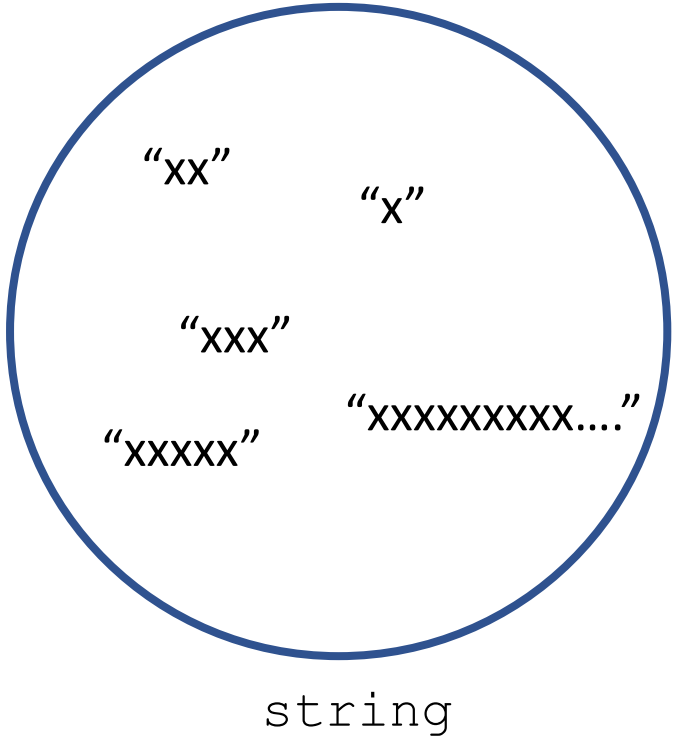


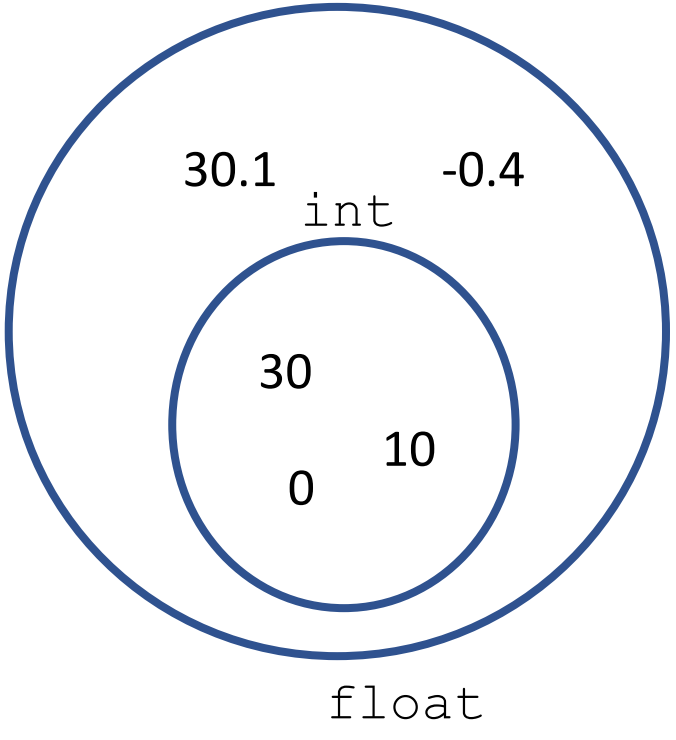


Fixed storage per variable

Finite set of values

How big is the set of strings?



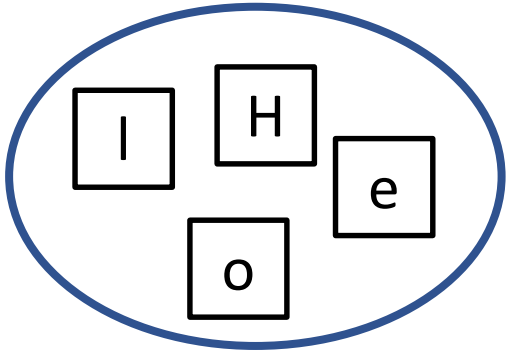


Dynamic storage per variable
“Infinite” set of values

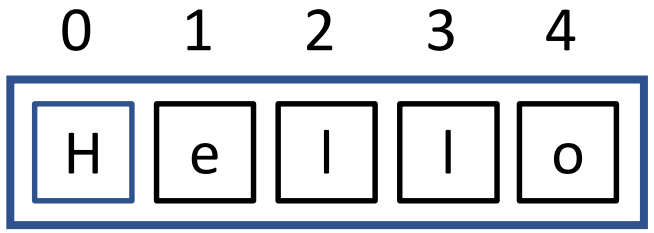
Fixed storage per variable
Finite set of values

`string` : a special sequence type

- Strings are a special-case type
  - Humans really like strings
  - We get a special type with useful operators
  - String constants are specially supported
- A string is ***not*** just a set of characters



{ ‘e’, ’H’, ‘l’, ‘o’ }



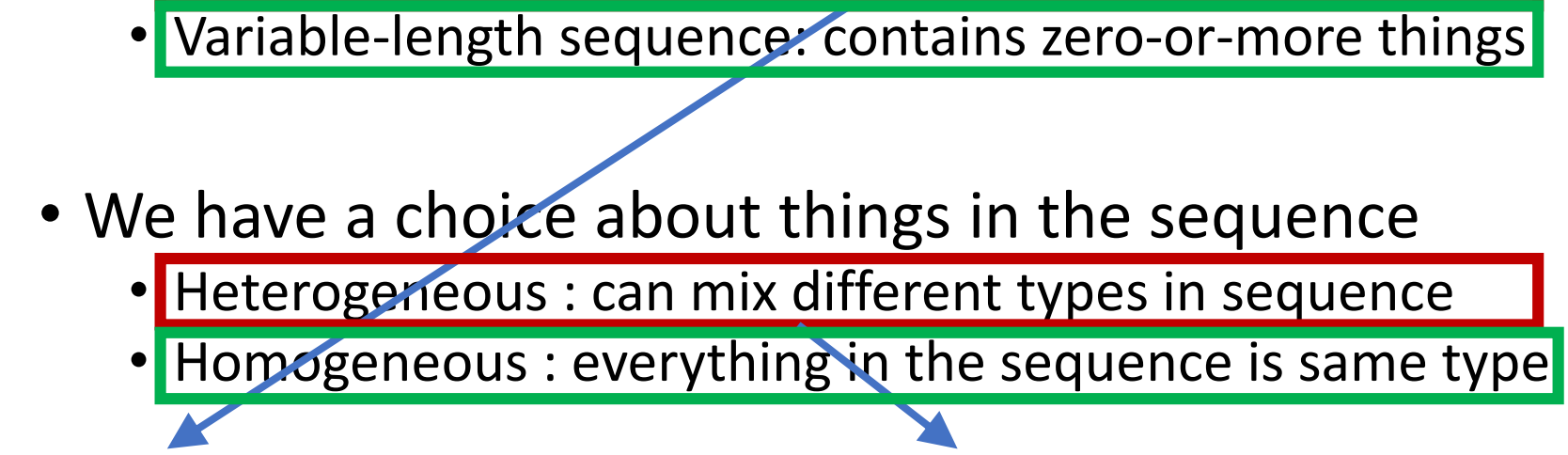
[’H’, ‘e’, ‘l’, ‘l’, ‘o’]

More general sequences

- We would like to describe sequences of *anything*

- We have a choice about sequence length
  - Fixed-length sequence: always contains exactly n things
  - Variable-length sequence: contains zero-or-more things

- We have a choice about things in the sequence
  - Heterogeneous : can mix different types in sequence
  - Homogeneous : everything in the sequence is same type



`vector` : a general sequence type

- `vector<T>` is a family of types
  - Each member of the family is specialised by a type `T`

- You can create vectors of most things

  `vector<int>` : sequence of zero-or-more integers

  `vector<float>` : sequence of zero-or-more floats

  `vector<bool>` : sequence of zero-or-more bools

  `vector<string>` : …

Creating a vector

- In header `<vector>`
- Use like a normal type
  - Must specify element type
- Name variable as normal

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

}
```

Adding values

- "Push" values into vector
  - Use `push_back` function
- Vector is not a builtin type
  - Name is not a keyword
  - Implemented using code
- It is an *object* type
  - Has "member" functions

For now we just *use* objects

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
}
```

Adding values

- “Push” values into vector
  - Use `push_back` function
- Add zero or more values

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
}
```

Adding values

- “Push” values into vector
  - Use `push_back` function
- Add zero or more values

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);
}
```

Adding values

- "Push" values into vector
  - Use `push_back` function
- Add zero or more values

```
veci == []
```

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);
}
```

Adding values

- “Push” values into vector
  - Use `push_back` function
- Add zero or more values

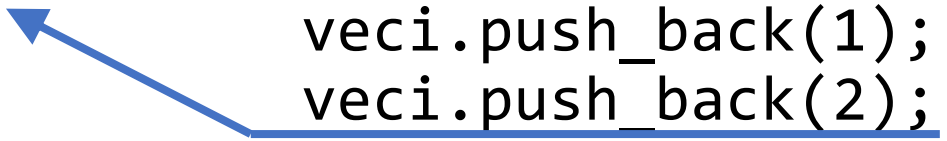
`veci == [1]`

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);
}
```

Adding values

- "Push" values into vector
  - Use `push_back` function
- Add zero or more values

`veci == [1,2]`

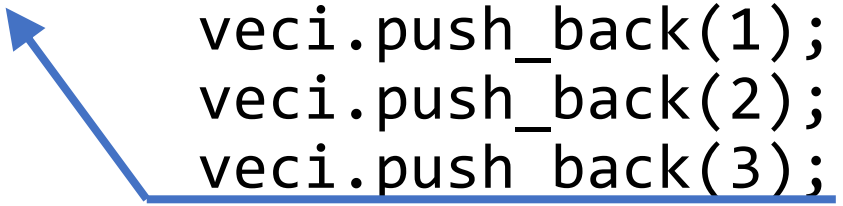


```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);
}
```

Adding values

- "Push" values into vector
  - Use `push_back` function
- Add zero or more values

`veci == [1,2,3]`



```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);
}
```

Accessing values

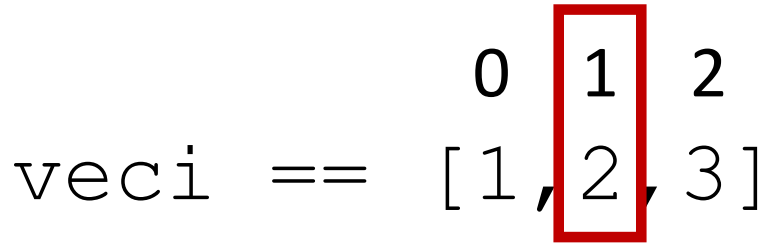
- Each element has an index
  - Starts at zero

```
            0  1  2
veci == [1,2,3]
```

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

}
```

Accessing values

- Each element has an index
  - Starts at zero
- Access elements by index
  - Use *square* brackets




```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    cout << veci[1];
}
```

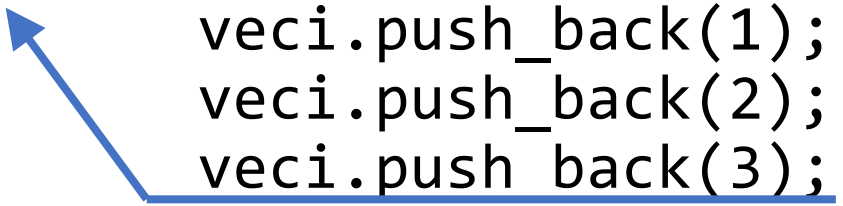
Accessing values

- Each element has an index
  - Starts at zero
- Access elements by index
  - Use *square* brackets

```
              0  1  2
veci == [1,2,3]
```

- Can also write values

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    veci[1] = 7;
}
```



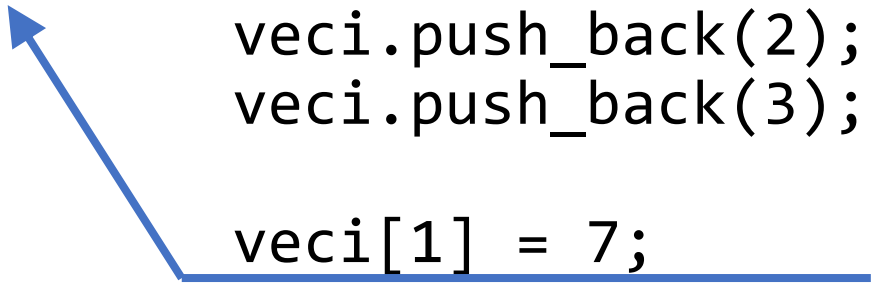
Accessing values

- Each element has an index
  - Starts at zero
- Access elements by index
  - Use *square* brackets

```
              0  1  2
veci == [1,7,3]
```

- Can also write values

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    veci[1] = 7;
}
```



Accessing values

- Each element has an index
  - Starts at zero
- Access elements by index
  - Use *square* brackets

```
              0  1  2
veci == [1,7,3]
```

- Can also write values
- Don't write invalid indices!

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    veci[1] = 7;
    veci[10] = 0;
}
```

Accessing values

- Each element has an index
  - Starts at zero
- Access elements by index
  - Use *square* brackets

```
              0  1  2
veci == [1,7,3] ?
```

- Can also write values
- Don't write invalid indices!

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    veci[1] = 7;
    veci[3] = 0;
}
```

Accessing values

- Each element has an index
  - Starts at zero
- Access elements by index
  - Use *square* brackets

```
              0  1  2
veci == [1,7,3] ?
```

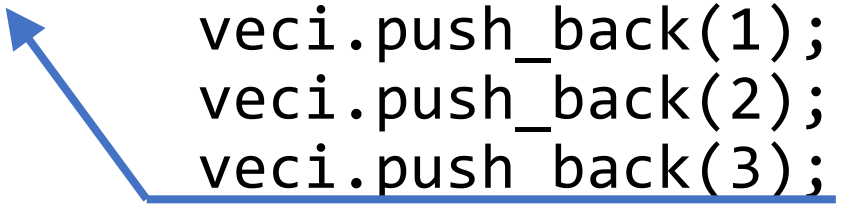
- Can also write values
- Don't write invalid indices!

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    veci[1] = 7;
    veci[3] = 0;
}
```

Resizing vectors

- Use `resize` to change size

```
            0  1  2
veci == [1,2,3]
```

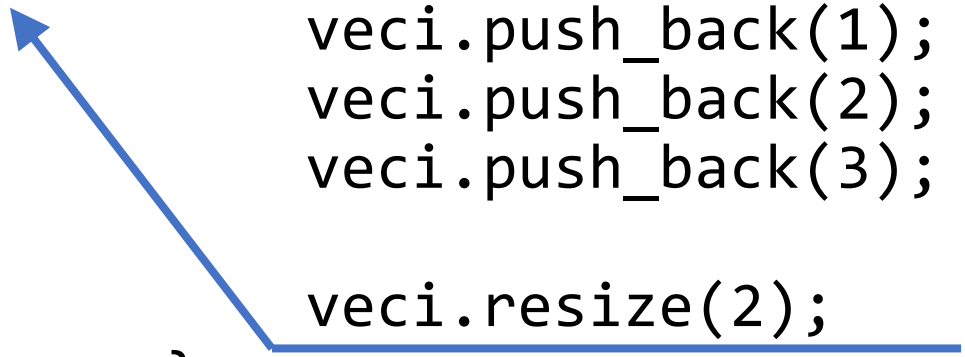


```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    veci.resize(2);
}
```

Resizing vectors

- Use `resize` to change size
- Can make it smaller
  - Values deleted from the end

```
          0  1
veci == [1,2]
```



```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    veci.resize(2);
}
```

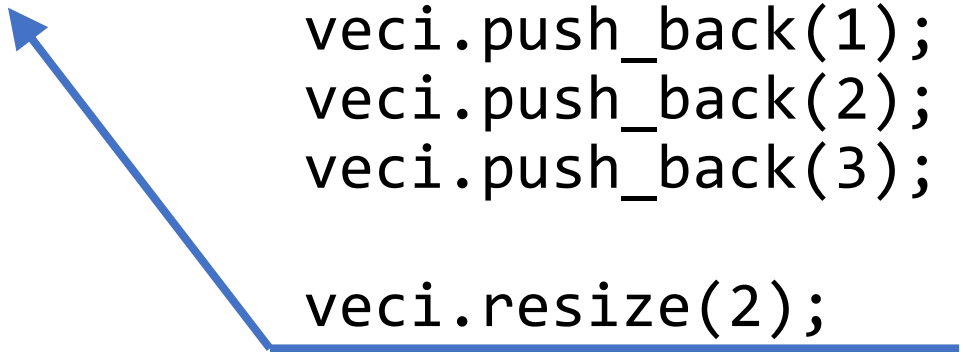
Resizing vectors

- Use `resize` to change size
- Can make it smaller
  - Values deleted from the end

```
          0  1
veci == [1,2]
```

- Can make it bigger

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    veci.resize(2);
    veci.resize(4);
}
```



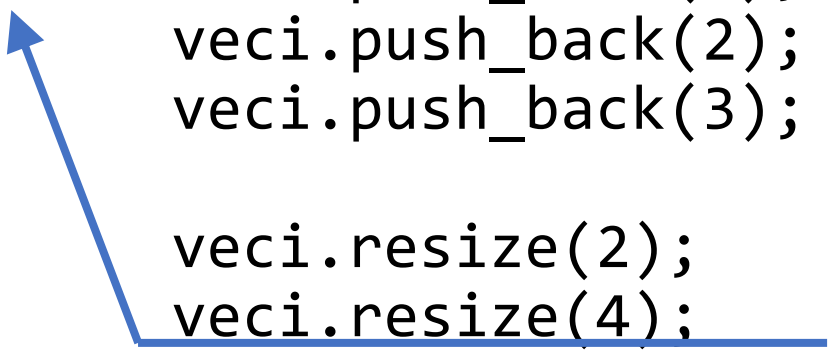
Resizing vectors

- Use `resize` to change size
- Can make it smaller
  - Values deleted from the end

```
            0  1  2  3
veci == [1,2,0,0]
```

- Can make it bigger
  - Will pad with some default
  - e.g. 0, false, “”

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> veci;

    veci.push_back(1);
    veci.push_back(2);
    veci.push_back(3);

    veci.resize(2);
    veci.resize(4);
}
```



Printing values

- Cannot directly print it
  - What should it look like?

```
int main()
{
  vector<int> veci;
  veci.push_back(1);
  veci.push_back(2);
  veci.push_back(3);

  cout << veci;
}
```

Printing values

- Cannot directly print it
  - What should it look like?
  - Must print each element

```
int main()
{
  vector<int> veci;
  veci.push_back(1);
  veci.push_back(2);
  veci.push_back(3);


}
```

Printing values

- Cannot directly print it
  - What should it look like?
  - Must print each element
- Find out the current size

```
int main()
{
  vector<int> veci;
  veci.push_back(1);
  veci.push_back(2);
  veci.push_back(3);

  int n = veci.size();

}
```

Printing values

- Cannot directly print it
  - What should it look like?
  - Must print each element
- Find out the current size
- Loop over the elements

```
int main()
{
  vector<int> veci;
  veci.push_back(1);
  veci.push_back(2);
  veci.push_back(3);

  int n = veci.size();

  for(int i=0; i<n; i++){
    cout << veci[i] ;
    cout << " ";
  }
}
```

`vector` : a general sequence type

- There are more operations : see documentation

- Vectors fit naturally with loops
  - Loop: sequence of statements; count varies at run-time
  - Vector: sequence of elements; count varies at run-time

- You might use either type of loop
  - `while` : pushing values onto a vector
  - `for` : enumerating elements in a vector

Where we are

- Variables and Types
  - Scalar types : int, float, …
  - Non-scalar types : vector, string
- Expressions and calculation
- Control and statements
  - Conditionals
  - Loops
- Input and output

Well done : you are Turing complete!

You can now write ***any*** computable program

Nothing you see from now on (in this course or any other) will ever fundamentally add to this [1]

Everything from here on falls into

1. Low-level understanding of implementation
2. High-level abstractions to make you more efficient
3. Methods and techniques to support practise

2 – Except maybe quantum computing