Passing by value and reference

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    String(const String &source)
    {
        length=source.length;
        capacity=source.capacity;
        buffer=new char[capacity];
        for(int i=0; i<length; i++){
            buffer[i] = source.buffer[i];
        }
    }

    ~String();
};
```

We have two new things here:

- *const : a new keyword*
- *& : a new type modifier*

Passing by value vs by pointer

```c
// Return next prime larger than x
int next_prime_above(int x);
```

Passing by value vs by pointer

```
// Return next prime larger than x
int next_prime_above(int x);

int main()
{
    int p = 7;

    int pn = next_prime( p );

    // Prints "7 -> 11"
    cout << p << "->" << pn << endl;
}
```

Passing by value vs by pointer

This is guaranteed. There is *no way* for `next_prime_above` to modify p.

```
// Return next prime larger than x
int next_prime_above(int x);

int main()
{
    int p = 7;

    int pn = next_prime( p );

    assert( p == 7 );

    // Prints "7 -> 11"
    cout << p << "->" << pn << endl;
}
```

Passing by value vs by pointer

```
// Return next prime larger than x
int next_prime_above(int *x);
```

Passing by value vs by pointer

```
// Return next prime larger than x
int next_prime_above(int *x);

int main()
{
    int p = 7;

    int pn = next_prime( &p );

    assert( p == 7 );

    // Prints "7 -> 11" (hopefully)
    cout << p << "->" << pn << endl;
}
```

Passing by value vs by pointer

This is ***not*** guaranteed. It is possible for `next_prime_above` to modify p if it wants via the pointer.

```
// Return next prime larger than x
int next_prime_above(int *x);

int main()
{
    int p = 7;

    int pn = next_prime( &p );

    assert( p == 7 );

    // Prints "7 -> 11" (hopefully)
    cout << p << "->" << pn << endl;
}
```

Passing by value vs by pointer

```c
// Return next prime larger than x
int next_prime_above(int *x);

int next_prime_above(int *x)
{
    *x = *x+1;
    while(!is_prime(*x)){
        *x = *x+1;
    }
    return *x;
}
```

User: "*Why on earth did you decide to modify x?*"

Coder: "*you chose to give me a pointer... why shouldn't I?*"

Passing by value vs by pointer

```
// Return next prime larger than x
void next_prime_above(int *x);
```

Return type is void

↓

Only other way for information to come out is by pointer

↓

So the function ***must*** modify the instance that x points to

Passing by value vs by pointer

```
// Return next prime larger than x
void next_prime_above(int *x);

int main()
{
    int p = 7;

    int pn = p;
    next_prime( &pn );

    assert( p == 7 );

    // Prints "7 -> 11" (hopefully)
    cout << p << "->" << pn << endl;
}
```

Why pass inputs by pointer?

Can we not just refactor to avoid pointer parameters?

```
int next_prime_above(int x);
```

Better

```
void next_prime_above(int *x);
```

Worse

```
int next_prime_above(int *x);
```

Refactor = modify working code to improve quality & structure

retain *functional* correctness of the code
improve *non-functional* properties of code

Passing *objects* as parameters

```
// Turns word into plural form
String plural(String word)
{
    word.append('s');
    return word;
}
```

Passing *objects* as parameters

```
// Turns word into plural form
String plural(String word);
```

Return value will be *copied* back

Parameter value will be *copied* into function

```
// Copy constructor for String
String::String(const String &source)
{
    length=source.length;
    capacity=source.capacity;
    buffer=new char[capacity];
    for(int i=0; i<length; i++){
        buffer[i] = source.buffer[i];
    }
}
```

Copy time is proportional to length of string

Passing ***objects*** as parameters

```
// Turns word into plural form
String plural(String word);
```

Return value will be *copied* back

Parameter value will be *copied* into function in time proportional to length

```
String plural(String word)
{
    word.append('s');
    return word;
}
```

Append time is ***constant*** (on average)

Passing *objects* as parameters

```
// Turns word into plural form
void plural(String *word);
```

Pointer value is copied into the function

The thing it ***points to*** is never copied

Time to copy a pointer is ***constant***

```
void plural(String *word)
{
  word->append('s');
}
```

Time of plural is ***constant***

Append time is ***constant*** (on average)

Tradeoffs : value versus reference

Passing by value : values are copied in and out of functions

```
String plural(String v);
```

No ambiguity
Matches math. view of "function"
Feels natural

Copies can be expensive
Difficult to return multiple values

Always cheap: no value copies
Provides flexibility

Very easy to get confused
Instances can be modified by mistake
What happens if the pointer is null?
Who owns the memory and calls delete?

```
String *plural(String *v);
```

Passing by reference : instance locations are passed into functions

Solutions in C++

1. The `const` modifier
   - Communicates about what is supposed to be modified
   - Protects against accidental changes

2. References "&" : a variation on pointers
   - A pointer that is guaranteed to never be null
   - Lets us create new names for an existing instance

`const` : protecting against change

```
// Return next prime larger than x
int next_prime_above(const int *x);
```

The function is making a promise:

"*I will not change the instance you are passing me*"
or
"*I will not change the instance pointed to by x*."

This is also enforced by the compiler

const : protecting against change

```
// Return next prime larger than x
int next_prime_above(const int *x);

int next_prime_above(const int *x)
{
    *x = *x+1;
    while(!is_prime(*x)){
        *x = *x+1;
    }
    return *x;
}
```

prime.cpp: In function '**int next_prime_above(const int*)**':
prime.cpp:3:8: **error:** assignment of read-only location '*** x**'
3 | ***x = *x +1**;

References : stricter pointers

- Pointer types are quite low-level and vague:

```
void f(T *ptr);
```

- The parameter `ptr` has a type which means that it:
  - might point at an instance of T; or
  - might point at an array of instances of T; or
  - might point part-way through an array of instances of T; or
  - might point at nothing; or
  - might point at neither something nor nothing."

References : stricter pointers

- Reference types are more high-level and strict

```
void f(T &ref);
```

- The parameter ref has a type which means that:
  - it refers to exactly one instance

References "look like" the instance

```
void f(T *ptr)
{
  T x = *ptr;
  int z = ptr->y;
  ptr->y += 1;
}
```

```
void f(T &ref)
{
  T x = ref;
  int z = ref.y;
  ref.y += 1;
}
```

```
void f(T copy)
{
  T x = copy;
  int z = copy.y;
  copy.y += 1;
}
```

- A pointer needs `*` or `->` to access the instance
- A reference is accessed directly or using `.`

References "look like" the instance

```
void f(T *ptr)
{
  T x = *ptr;
  int z = ptr->y;
  ptr->y += 1;
}
```

```
void f(T &ref)
{
  T x = ref;
  int z = ref.y;
  ref.y += 1;
}
```

```
void f(T copy)
{
  T x = copy;
  int z = copy.y;
  copy.y += 1;
}
```

- A pointer needs `*` or `->` to access the instance
- A reference is accessed directly or using `.`

References "look like" the instance

```
void f(T *ptr)
{
  T x = *ptr;
  int z = ptr->y;
  ptr->y += 1;
}
```

```
void f(T &ref)
{
  T x = ref;
  int z = ref.y;
  ref.y += 1;
}
```

```
void f(T copy)
{
  T x = copy;
  int z = copy.y;
  copy.y += 1;
}
```

- A pointer needs `*` or `->` to access the instance
- A reference is accessed directly or using `.`

References "look like" the instance

```
void f(T *ptr)
{
  T x = *ptr;
  int z = ptr->y;
  ptr->y += 1;
}
```

```
void f(T &ref)
{
  T x = ref;
  int z = ref.y;
  ref.y += 1;
}
```

```
void f(T copy)
{
  T x = copy;
  int z = copy.y;
  copy.y += 1;
}
```

- A pointer needs * or -> to access the instance
- A reference is accessed directly or using .

References "look like" the instance

```
void f(T *ptr)
{
  T x = *ptr;
  int z = ptr->y;
  ptr->y += 1;
}
```

```
void f(T &ref)
{
  T x = ref;
  int z = ref.y;
  ref.y += 1;
}
```

```
void f(T copy)
{
  T x = copy;
  int z = copy.y;
  copy.y += 1;
}
```

- A pointer needs `*` or `->` to access the instance
- A reference is accessed directly or using `.`

References "look like" the instance

```
void f(T *ptr)
{
  T x = *ptr;
  int z = ptr->y;
  ptr->y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( &t );

  assert( t.x==6 );
}
```

```
void f(T &ref)
{
  T x = ref;
  int z = ref.y;
  ref.y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( t );

  assert( t.x==6 );
}
```

```
void f(T copy)
{
  T x = copy;
  int z = copy.y;
  copy.y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( &t );

  assert( t.x==5 );
}
```

References "look like" the instance

```
void f(T *ptr)
{
  T x = *ptr;
  int z = ptr->y;
  ptr->y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( &t );

  assert( t.x==6 );
}
```

```
void f(T &ref)
{
  T x = ref;
  int z = ref.y;
  ref.y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( t );

  assert( t.x==6 );
}
```

```
void f(T copy)
{
  T x = copy;
  int z = copy.y;
  copy.y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( &t );

  assert( t.x==5 );
}
```

References "look like" the instance

```
void f(T *ptr)
{
  T x = *ptr;
  int z = ptr->y;
  ptr->y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( &t );

  assert( t.x==6 );
}
```

```
void f(T &ref)
{
  T x = ref;
  int z = ref.y;
  ref.y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( t );

  assert( t.x==6 );
}
```

```
void f(T copy)
{
  T x = copy;
  int z = copy.y;
  copy.y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( t );

  assert( t.x==5 );
}
```

References "look like" the instance

```
void f(T *ptr)
{
  T x = *ptr;
  int z = ptr->y;
  ptr->y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( &t );

  assert( t.x==6 );
}
```

```
void f(T &ref)
{
  T x = ref;
  int z = ref.y;
  ref.y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( t );

  assert( t.x==6 );
}
```

```
void f(T copy)
{
  T x = copy;
  int z = copy.y;
  copy.y += 1;
}

int main()
{
  T t;
  t.y = 5;

  f( t );

  assert( t.x==5 );
}
```

References are fixed to one instance

- A reference always refers to the same instance
  - You cannot change what it refers to
  - You cannot have an uninitialised reference

```
int main(int argc, char **argc)
{
    int a1 = atoi(argv[1]);
    int &r;
}
```

```
source.cpp: In function 'int main()':
source.cpp:4:10: error: 'r' declared as
reference but not initialized
  4 | int &r;
    |      ^
```

References are fixed to one instance

- A reference always refers to the same instance
  - You cannot change what it refers to
  - You cannot have an uninitialised reference

```
int main(int argc, char **argc)
{
    int a1 = atoi(argv[1]);
    int &r = a1;
}
```

References are fixed to one instance

- A reference always refers to the same instance
  - You cannot change what it refers to
  - You cannot have an uninitialised reference

```
int main(int argc, char **argc)
{
    int a1 = atoi(argv[1]);
    int &r = a1;

    r = 10;
}
```

References are fixed to one instance

- A reference always refers to the same instance
  - You cannot change what it refers to
  - You cannot have an uninitialised reference
  - Any modifications to reference ***also*** change the instance

```
int main(int argc, char **argc)
{
    int a1 = atoi(argv[1]);
    int &r = a1;

    r = 10;

    assert( a1 == 10 );
}
```

References are fixed to one instance

- A reference always refers to the same instance
  - You cannot change what it refers to
  - You cannot have an uninitialised reference
  - Any modifications to reference ***also*** change the instance
  - Addresses of reference and original are the same

```
int main(int argc, char **argc)
{
    int a1 = atoi(argv[1]);
    int &r = a1;

    r = 10;

    assert( a1 == 10 );
    assert( &a1 == &r );
}
```

Pointers vs references

- Pointers create a *temporary link* to an instance
  - We can create a link using &
  - We can move the link around using arithmetic
  - We can change the link to a different instance
- References create a *permanent alias* to an instance
  - The reference is a new name for the original instance
  - Once a reference is created it cannot be changed
  - Changes to the reference change the original instance
  - The alias exists for as long as the reference does

Basic type variants

T x	const T x
T *x	const T *x
T &x	const T &x

References provider clearer APIs

Modifies original string with no copies.
Parameter to plural ***will*** be modified.

```
void plural(String &word);
```

Returns a new string, requiring one copy.
Parameter to plural will ***not*** be modified

```
String plural(const String &word);
```

Different meaning

Better

Worse

```
String plural(String word);
```

Returns a new string, requiring ***two*** copies.
Parameter to plural will ***not*** be modified

Note: in more recent C++ revisions certain copies will be removed, but the principle is still the same

Guidance for references

- Only use references for function parameters
  - You *can* use them elsewhere, but avoid for now
- Scenario 1 : passing "expensive" objects as input
  - e.g. passing a string or vector into a function
  - Any variable sizes type that takes time to copy
  - Don't use for primitive types like int, float, ...
- Scenario 2 : passing an instance to be modified
  - Prefer passing a reference over a pointer to one object
- Scenario 3 : using a parameter as an output
  - Prefer passing a reference over a pointer

Example : Copy constructor

Read-only view of the String we want to copy

```
String::String(const String &source)
{
    // Copy the length and capacity verbatim
    length=source.length;
    capacity=source.capacity;

    // Create a new buffer just for us
    buffer=new char[capacity];

    // Copy the other string's data in
    for(int i=0; i<length; i++){
        buffer[i] = source.buffer[i];
    }
}
```

FAQ : so... is a reference a pointer?

We are not treating pointers as addresses or numbers here because it is a "dangerous" way of thinking

But... yes. They are numbers in the types of computer you are currently using

So are references the same as pointers?

Mostly. In the types of computer you are currently using they are often implemented as pointers

But: *there exist computers where pointers are not just numbers, and where references are not the same as pointers*

`const` : member functions

- How methods affect state is important for users
  - Some methods **change** the state of an object
  - Some methods *read* the state of an object

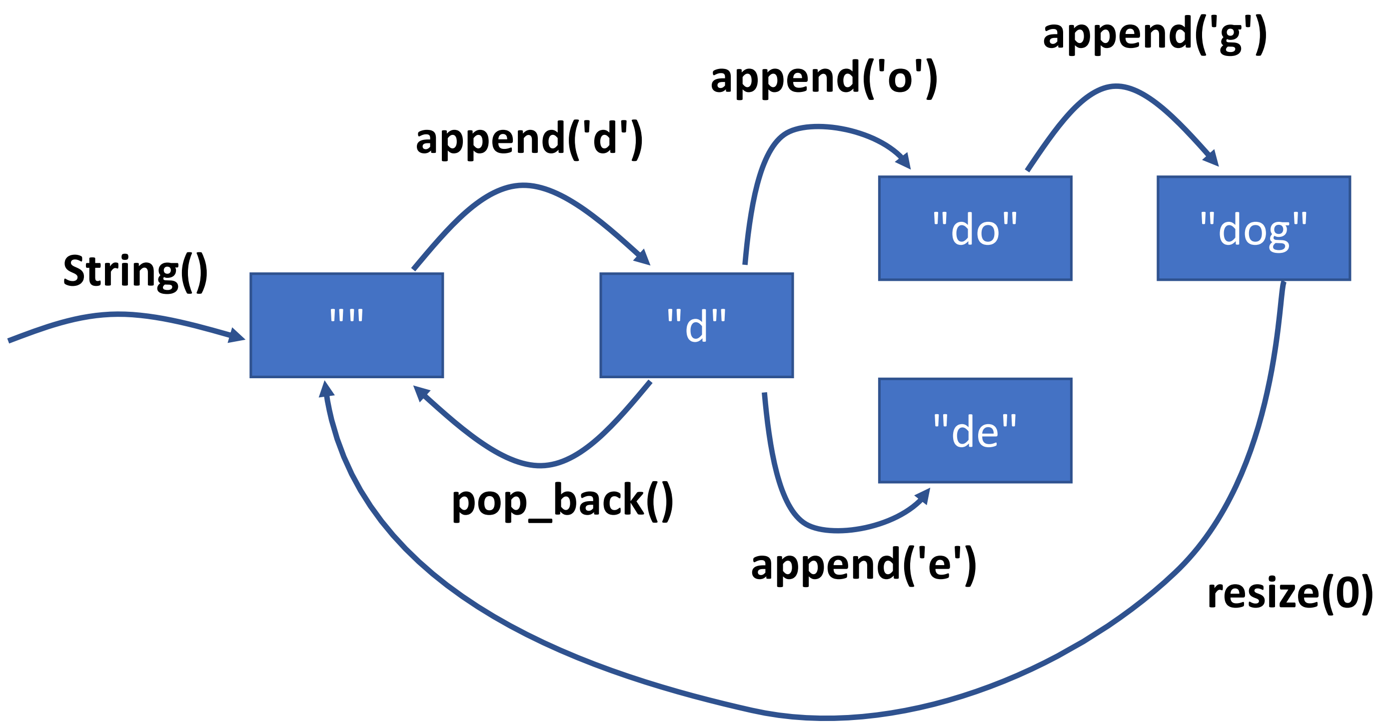
- `const` indicates that a parameter does not change
  - Methods have a very important parameter : `this`

```c
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *Str_create();
void    Str_destroy(String *s);

int  Str_size(String *s);
void Str_resize(String *s, int n);
char Str_at(String *s);
void Str_append(char String *s, char c);
```

```cpp
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
  String();
  ~String();

  void size();
  void resize(int n);
  char at(int index);
  void append(char c);
};
```

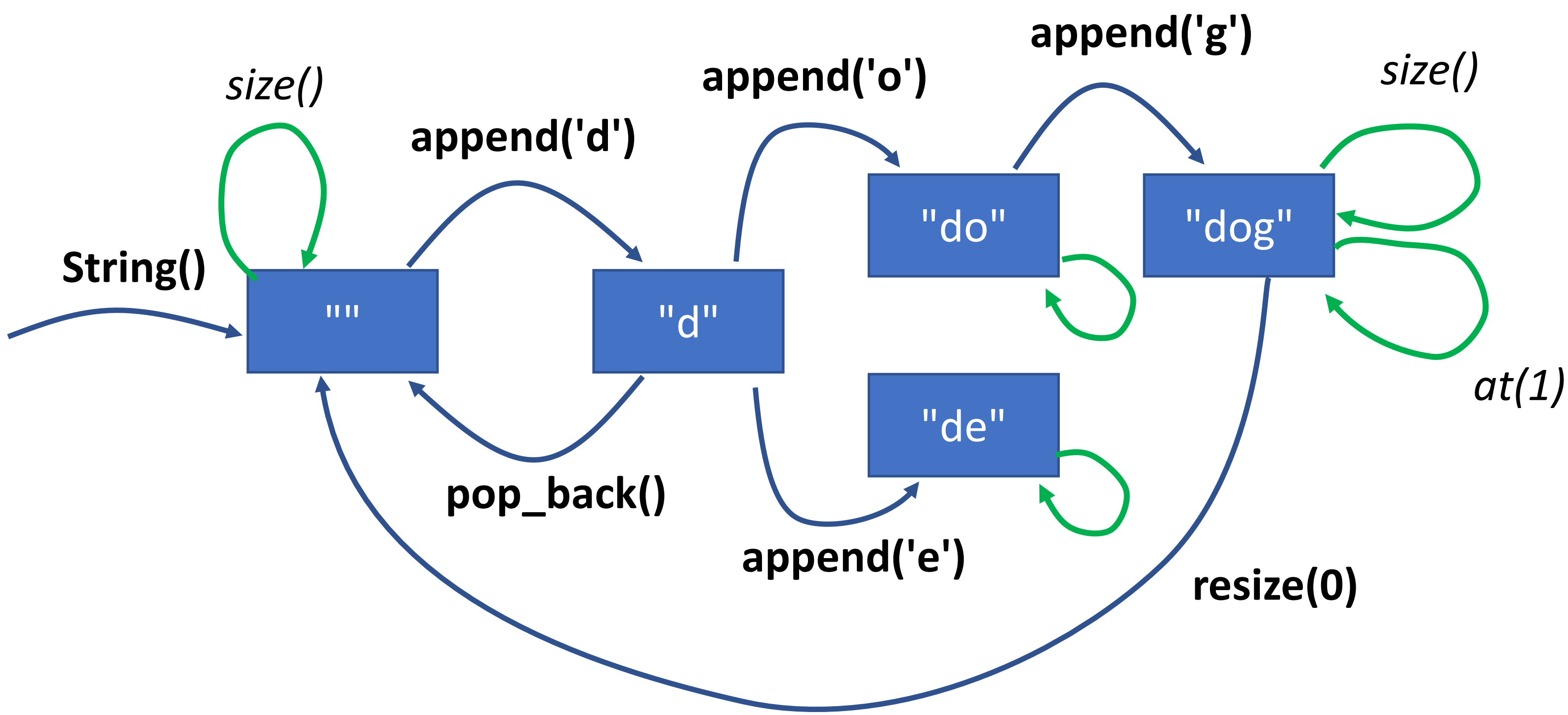
`const` : member functions

- How methods affect state is important for users
  - Some methods **change** the state of an object
  - Some methods *read* the state of an object



const : member functions

- How methods affect state is important for users
  - Some methods **change** the state of an object
  - Some methods *read* the state of an object



```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *Str_create();
void    Str_destroy(String *s);

int  Str_size(String *s);
void Str_resize(String *s, int n);
char Str_at(String *s);
void Str_append(char String *s, char c);
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
  String();
  ~String();

  void size();
  void resize(int n);
  char at(int index);
  void append(char c);
};
```

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *Str_create();
void    Str_destroy(String *s);

int  Str_size(const String *s);
void Str_resize(String *s, int n);
char Str_at(const String *s);
void Str_append(char String *s, char c);
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
  String();
  ~String();

  void size();
  void resize(int n);
  char at(int index);
  void append(char c);
};
```

```c
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *Str_create();
void    Str_destroy(String *s);

int  Str_size(const String *s);
void Str_resize(String *s, int n);
char Str_at(const String *s);
void Str_append(char String *s, char c);
```

```cpp
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
  String();
  ~String();

  void size() const;
  void resize(int n);
  char at(int index) const;
  void append(char c);
};
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    char at(int index) const
    {
        return buffer[index];
    }

};
```

```cpp
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    char at(int index) const;
};

char String::at(int index) const;
{
    return buffer[index];
}
```

Function Overloading

to_string : what is its input type?

```
int main()
{
    int v1 = 100;
    string s1 = to_string( v1 );

    double v2 = 1.1;
    string s2 = to_string( v2 );
}
```

`string` : constructor input type?

```
int main()
{
	string s1;

	string s2(3, 'X');

	string s3("Hello");
}
```

pow : input argument types?

```
int main()
{
    float a = 2.2;
    complex<float> b{3.1,0.2};

    float aa =           pow(a,a);

    complex<float> ab = pow(a,b);

    complex<float> bb = pow(b,b);
}
```

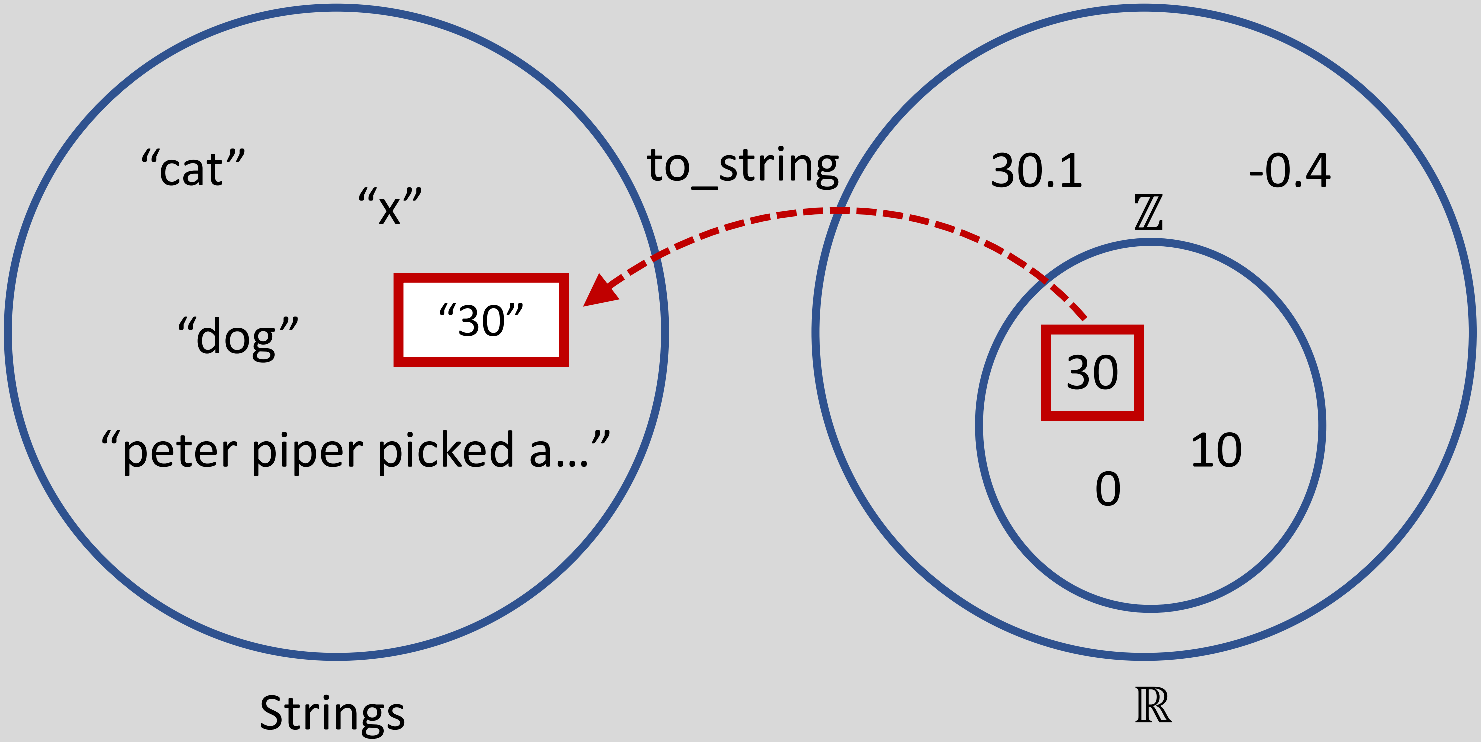
Functions can be *overloaded*

- A given functions can have multiple definitions
  - As long as each definition has different input types
- The compiler will pick the correct version
  - It will pick based on the arguments to the function

to_string : overload resolution

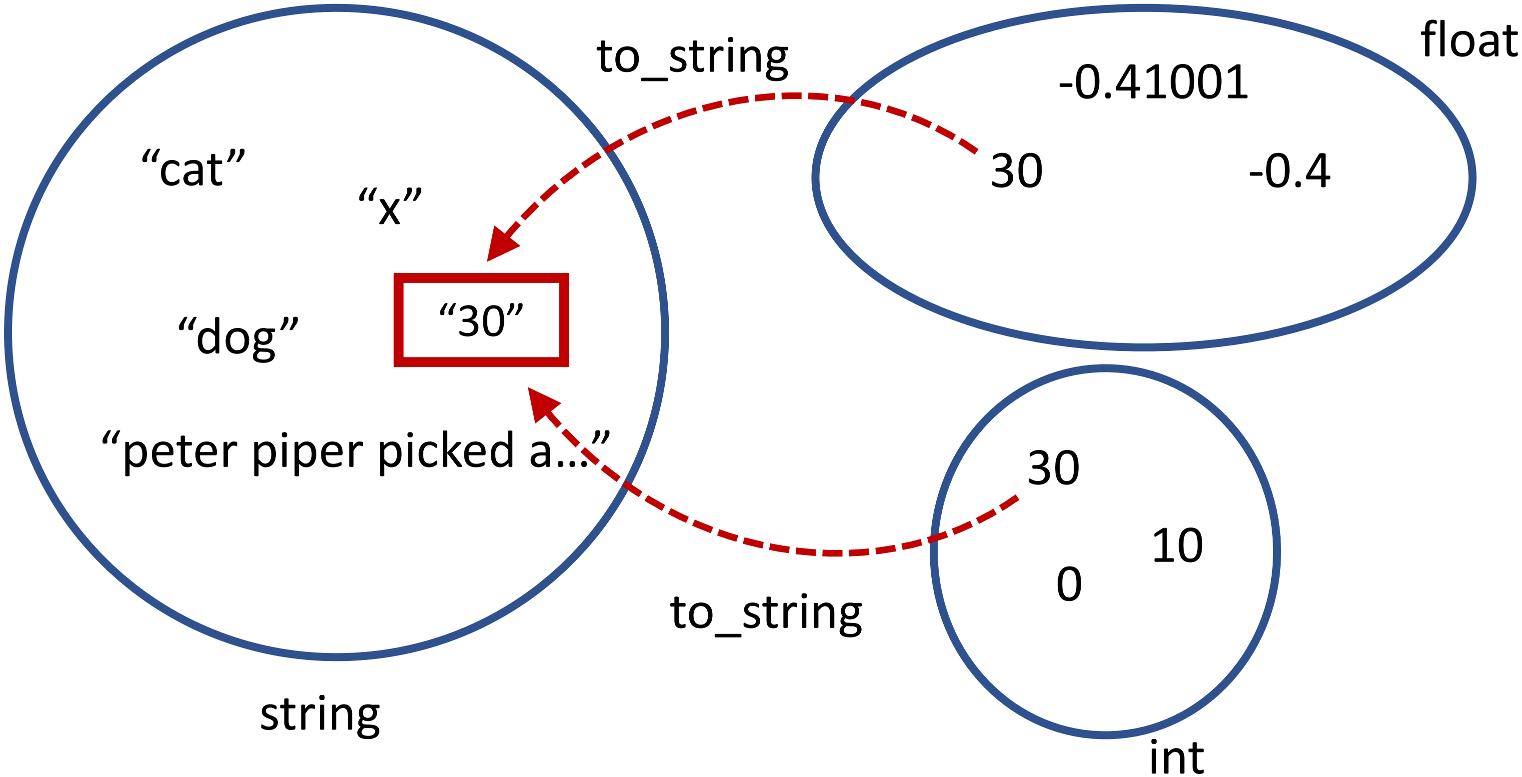
```
// somewhere in <string>
string to_string(int v);
string to_string(unsigned v);
string to_string(float v);
string to_string(double v);
...


int main()
{
    int v1 = 100;
    string s1 = to_string( v1 );

    double v2 = 1.1;
    string s2 = to_string( v2 );
}
```

Recap: Mapping between types



The function `to_string` maps the set of ints to the set of strings

Mapping between types : overloads



The function `to_string` maps the type `int` to the type `string`
The function `to_string` maps the type `float` to the type `string`
...

`string` : constructor selection

```
// <string>
string::string();
string::string(int n);
string::string(int n, char c);
string::string(const char *);


int main()
{
    string s1;

    string s2(3, 'X');

    string s3("Hello");
}
```

```
// <cmath>
float pow(float x, float y);
double pow(double x, double y);
...

// <complex>
complex<float> pow(complex<float> x, complex<float> y);
complex<double> pow(complex<double> x, complex<double> y);
...


int main()
{
    float a = 2.2;
    complex<float> b{3.1,0.2};

    float aa =           pow(a,a);

    complex<float> ab = pow(a,b);

    complex<float> bb = pow(b,b);
}
```

This is a slight lie for complex. Relies on templates, which are still to come.

Overloads are *mostly* quite simple

- You can have multiple function overloads
  - They must differ in ***number*** of parameters; and/or
  - Differ in the ***type*** of parameters.
  - You cannot define the *same* declaration twice
- When you call a function, the compiler will:
  1. Find the set of function overloads with that name
  2. Filter out overloads with different parameter count
  3. Filter out overloads where a type doesn't match
  4. Possible outcomes:
     1. No overloads are left: *compiler error*
     2. More than one overload is left: *compiler error*
     3. One overload is left: *use that function*

Overloads are sometimes complex

- The overload resolution is quite technical
  - It gets fiddly around what is the *best* overload
  - I still get confused, with 30 years of C++ experience
- Your job is *not* to be a C++ details expert
  - Your job is to get stuff done using C++
- Most of the time, things will just work
- If they *don't*, explicitly choose argument types
  - Put arguments into variables; or
  - Cast arguments to chose type

Operator Overloading

Strings can be added – how?

```
int main()
{
    string he = "he" ;
    string llo = "llo" ;

    string hello = he + llo;

    cout << hello << endl;
}
```

string is not a built-in language type, it is just a class

Strings can be indexed – how?

```
int main()
{
    string hello("hello");

    for(int i=0; i<hello.size(); i++){
        cout << hello[i];
    }
    cout << endl;
}
```

Strings can be indexed – how?

```
int main()
{
    string hello("hello");

    for(int i=0; i<hello.size(); i++){
        cout << hello[i];
    }
    cout << endl;
}
```

Things can be printed – how?

```
int main()
{
    string hello("hello");

    for(int i=0; i<hello.size(); i++){
        cout << hello[i];
    }
    cout << endl;
}
```

We've been doing this so long it isn't odd, but:

- `x << s` is actually the left-shift operator
- `cout` must be an object, but where is it?

Strings *could* add via a function

```
string add(const string &a, const string &b);



int main()
{
    string he = "he" ;
    string llo = "llo" ;

    string hello = add( he , llo );

    cout << hello << endl;
}
```

Strings *could* be indexed via **at**

```
char string::string(int index) const;



int main()
{
    string hello("hello");

    for(int i=0; i<hello.size(); i++){
        cout << at( hello, i );
    }
    cout << endl;
}
```

Could print things via an object

```
Type1 cout;


void print(Type1 &stream, const string &s);



int main()
{
    string hello("hello");

    for(int i=0; i<hello.size(); i++){
        print( cout , at( hello, i ) );
    }
    cout << endl;
}
```

Could print things via an object

```
Type1 cout;
Type2 endl;

void print(Type1 &stream, const string &s);
void print(Type1 &stream, const Type2 &e);


int main()
{
    string hello("hello");

    for(int i=0; i<hello.size(); i++){
        print( cout , at( hello, i ) );
    }
    print( cout , endl );
}
```

Recap: Results of operations

Many operations are *closed*

the result is the same type as the inputs

Multiplication: $a \times b$

Addition: $a + b$

$$a \in \mathbb{Z} \wedge b \in \mathbb{Z} \Rightarrow (a \times b) \in \mathbb{Z}$$

"If a is an integer; **and** b is an integer; **then** a times b is an integer"

Function declarations

Function declarations specify the function prototype

$$\exp : \mathbb{R} \to \mathbb{R}$$

```
float exp( float x );
```

Declarations are not *required* to name parameters
Just like in maths, it is the types that matter
However, it often helps users to have names

Addition as a function

Function declarations specify the function prototype

$$+ : \mathbb{R} \text{ x } \mathbb{R} \rightarrow \mathbb{R}$$

```
float add( float x, float y );
```

$$+ : \texttt{string} \text{ x } \texttt{string} \rightarrow \texttt{string}$$

```
string add(const string &x, const string &y );
```

Overloading : addition

```
String
{
    ...
};

String add(const String &a, const String &b)
{
   ...
}
```

Overloading : addition

```
String
{
    ...
};

String add(const String &a, const String &b)
{
   ...
}

String operator +(const String &a, const String &b)
{
    return add(a,b);
}
```

Overloading : equality

```
String
{
     ...
};

String equals(const String &a, const String &b)
{
    ...
}

String operator ==(const String &a, const String &b)
{
    return equals(a,b);
}
```

Overloading : comparison

```
String
{
    ...
};

String less_than(const String &a, const String &b)
{
   ...
}

String operator <(const String &a, const String &b)
{
    return less_than(a,b);
}
```

You can overload most operators

- We've seen the type `complex<float>`
  - We can do normal maths on it : +, -, *, /
  - We can compare it with others: ==, <, <=, ...
  - We can mix it with other types like `float` and `double`
- Overloading lets us create new math. types
  - Matrices, vectors, rationals, ...
  - Infinite size integers
  - Aribtrary precision floating point numbers
- These can then be used liked "normal" numbers

Operator overloading in practice

- Operator overloading can be mis-used
  - It should only be used where it makes sense
  - Don't make + mean "print"
- You are mainly expected to *use* overloaded operators
  - Explained so that you can understand what is going on
  - You won't be assessed on most of it
  - *Some* of you may use it in later years
- Only a subset are actually needed in practice
  - Assignment + Comparison
  - Needed to get containers and algorithms to work
  - We will delve more into these next : templates!