Admin

- Wiseflow test results returned on Friday
  - Including detailed feedback document on both tests
- Github registrations: 150 so far
  - About 10 pending to do today
- Portfolio repositories
  - Should be given access to them today
  - (Github didn't like it when I tried to make 600 repos)
- Q2 portfolio: apologies for the delay
  - We didn't anticipate the workload crunch after Xmas

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");
}
```

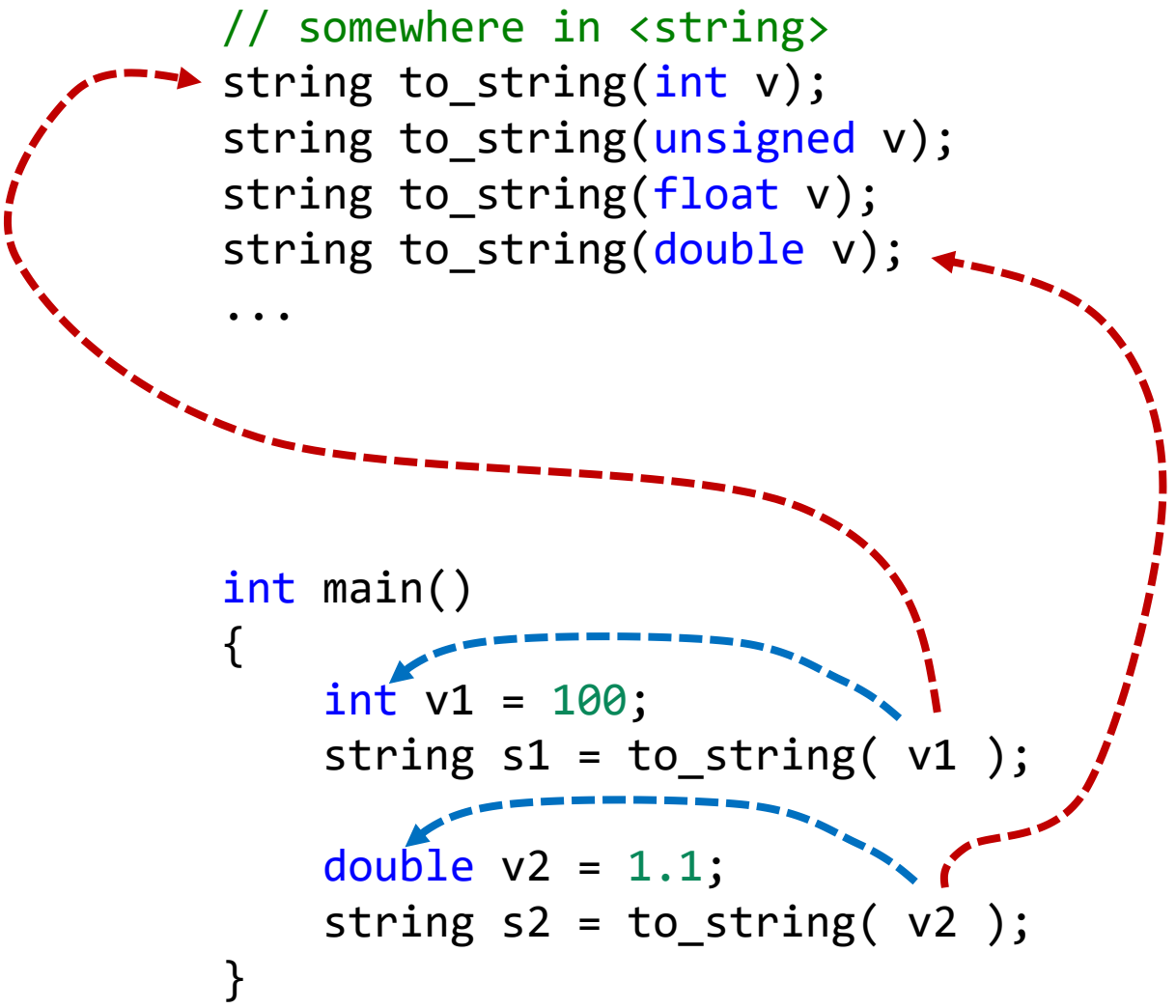
pow : input argument types?

```
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);
}
```

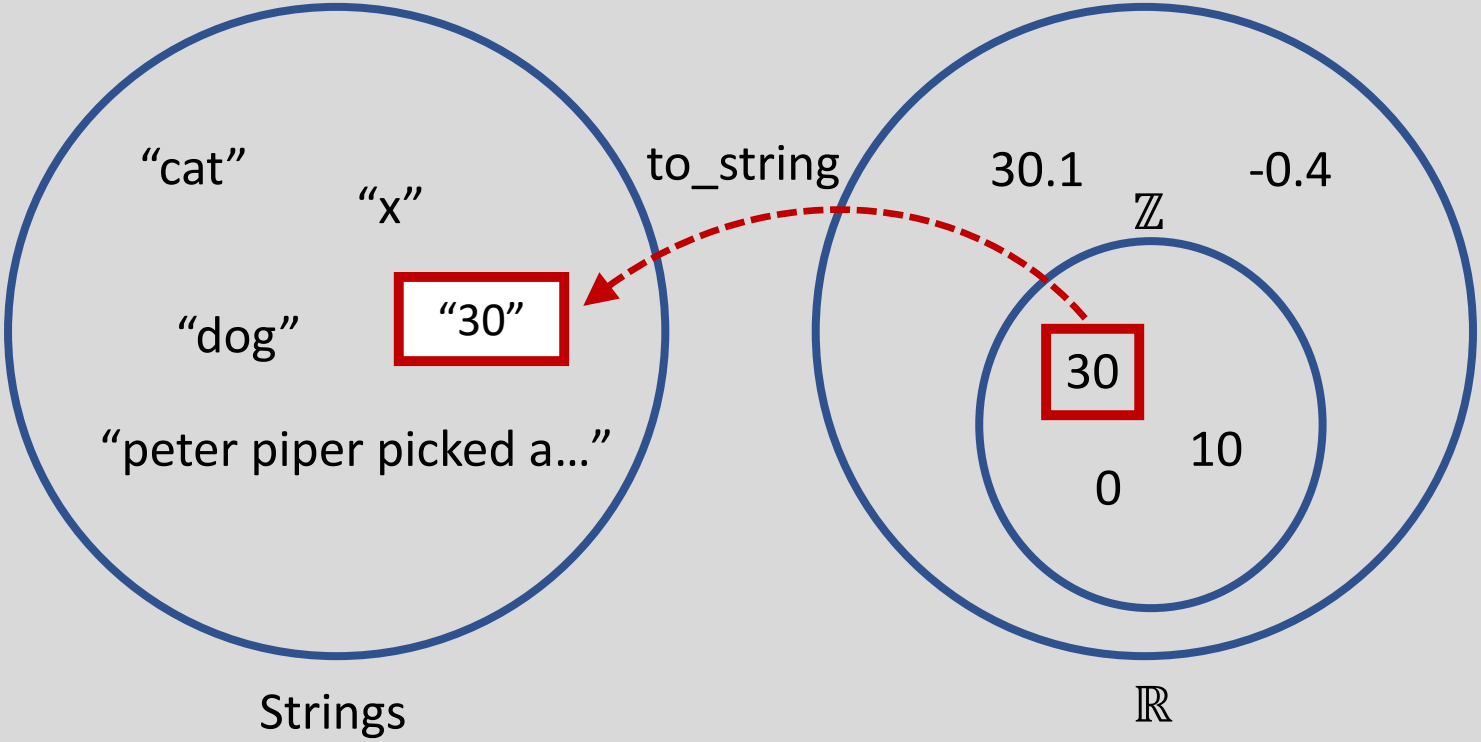
Functions can be *overloaded*

- 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
  - More specifically: the ***types*** of the arguments

to_string : overload resolution

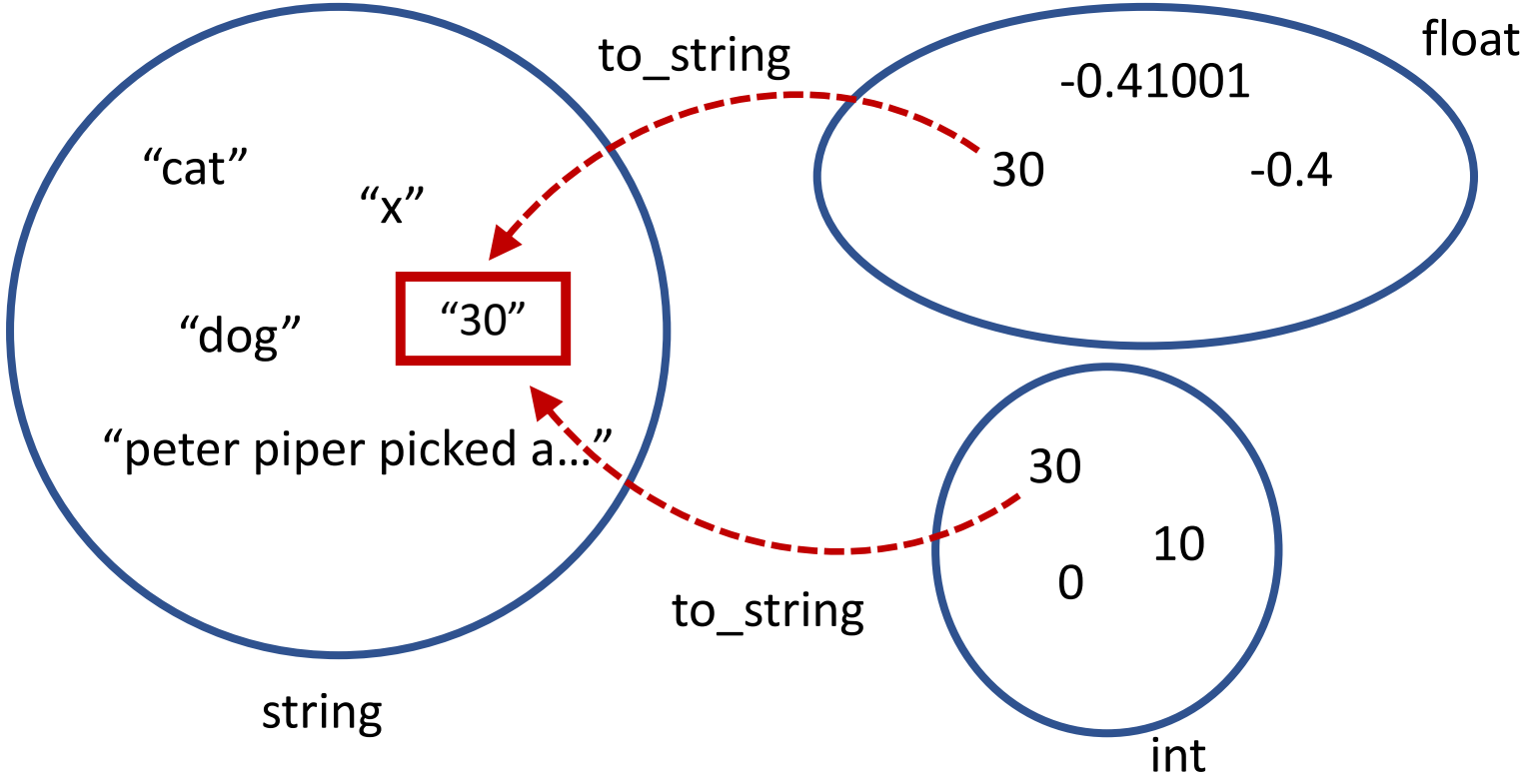


Recap: Mapping between types



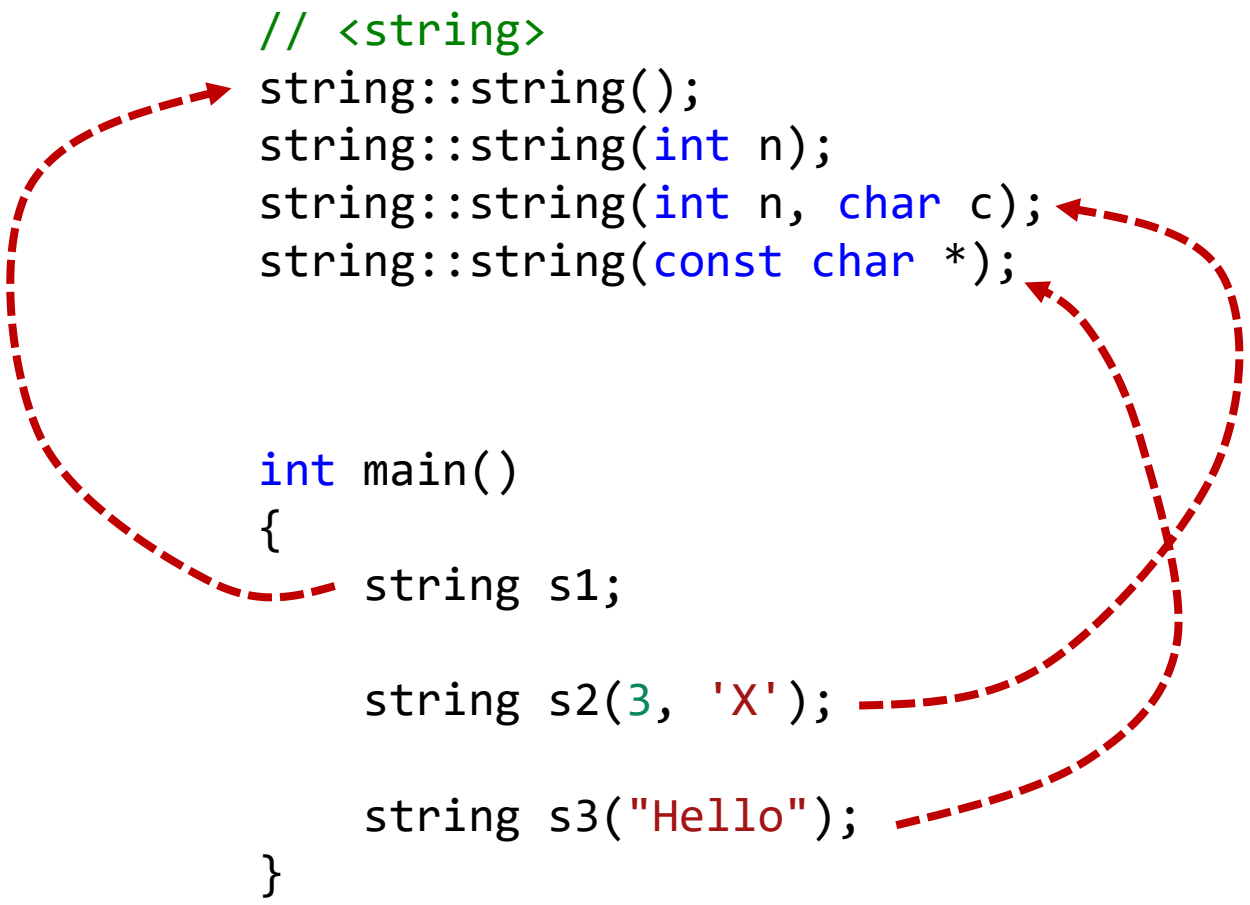
The function `to_string` maps the set of ints to the set of strings

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



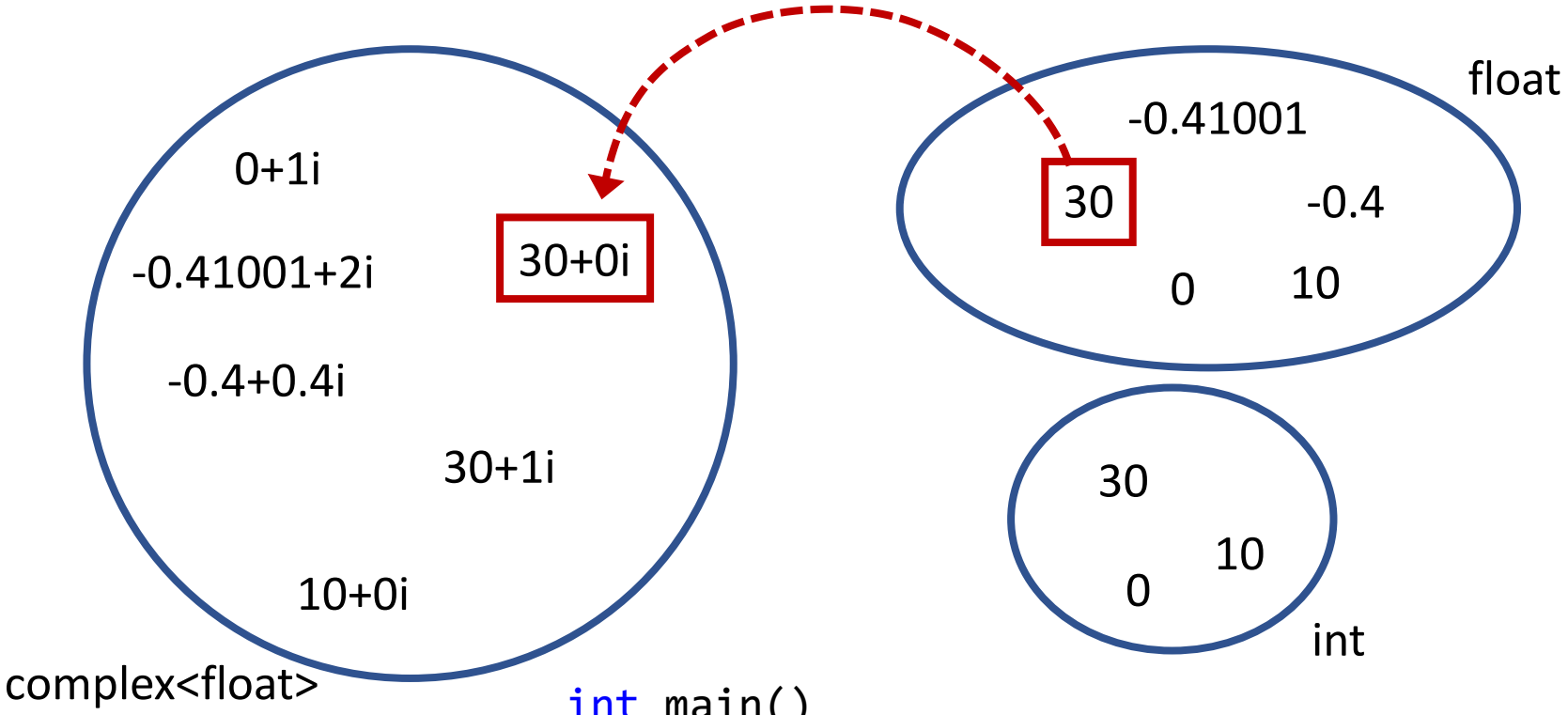
```
// <cmath>
float pow(float x, float y);
double pow(double x, double y);
...

// <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);
}
```

a has type float
must "upgrade" to complex<float>

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

Implicit conversion: constructors

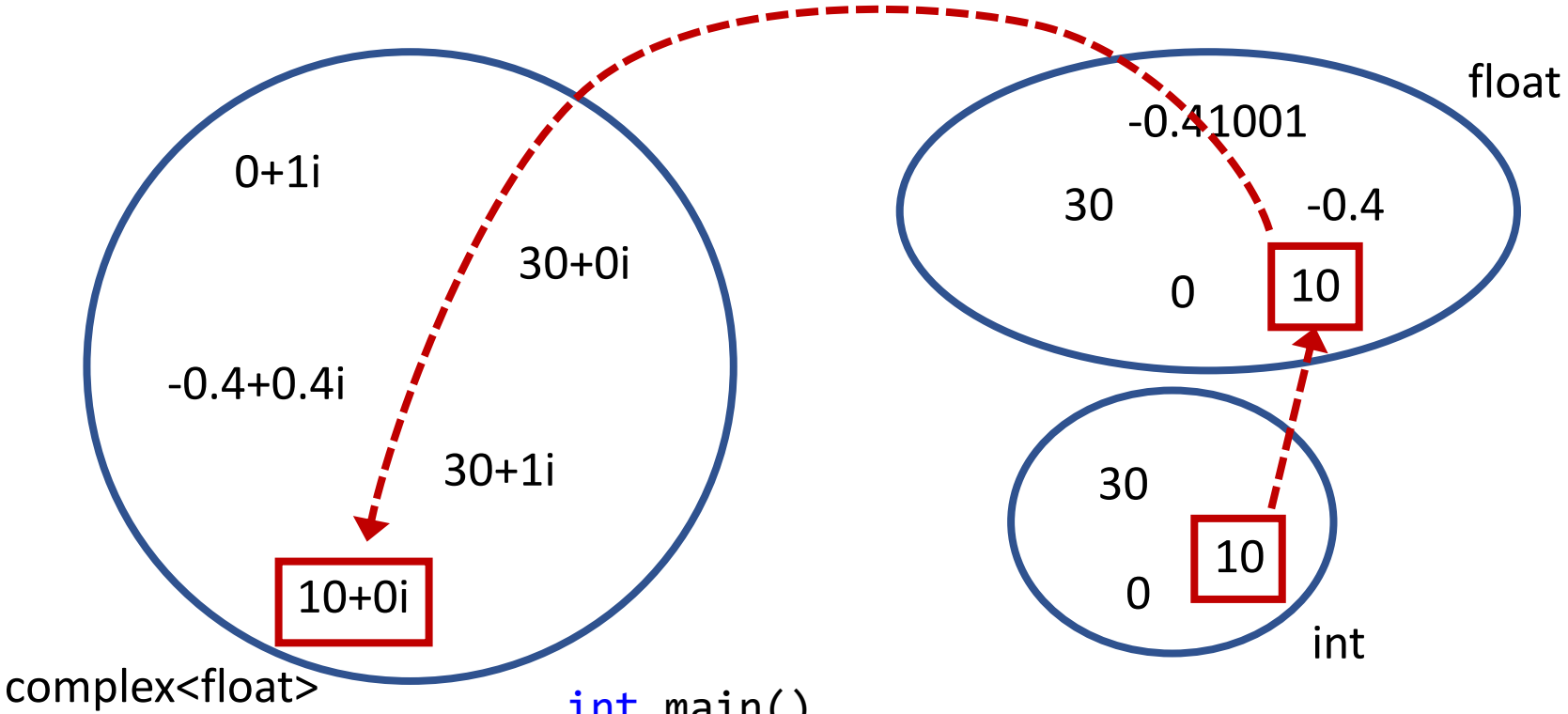
```
complex<float>::complex<float>(float x);
```




```
int main()
{
    float a = 30.0;
    complex<float> c = a;
}
```

Implicit conversion: constructors

```
complex<float>::complex<float>(float x);
```




```
int main()
{
    int a = 10;
    complex<float> c = a;
}
```

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. Use constructors to try to adapt types
  5. 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 first; or
  - Cast arguments to chose type explicitly

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;
}
```

Things can be printed – how?

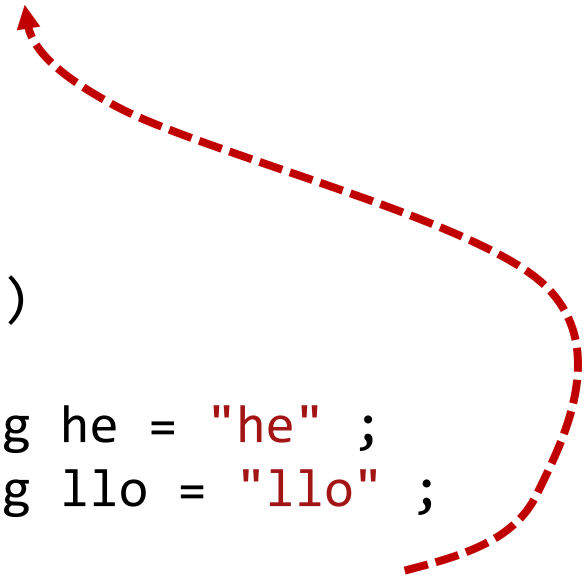
```
int main()
{
    string hello("hello");

    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);
```

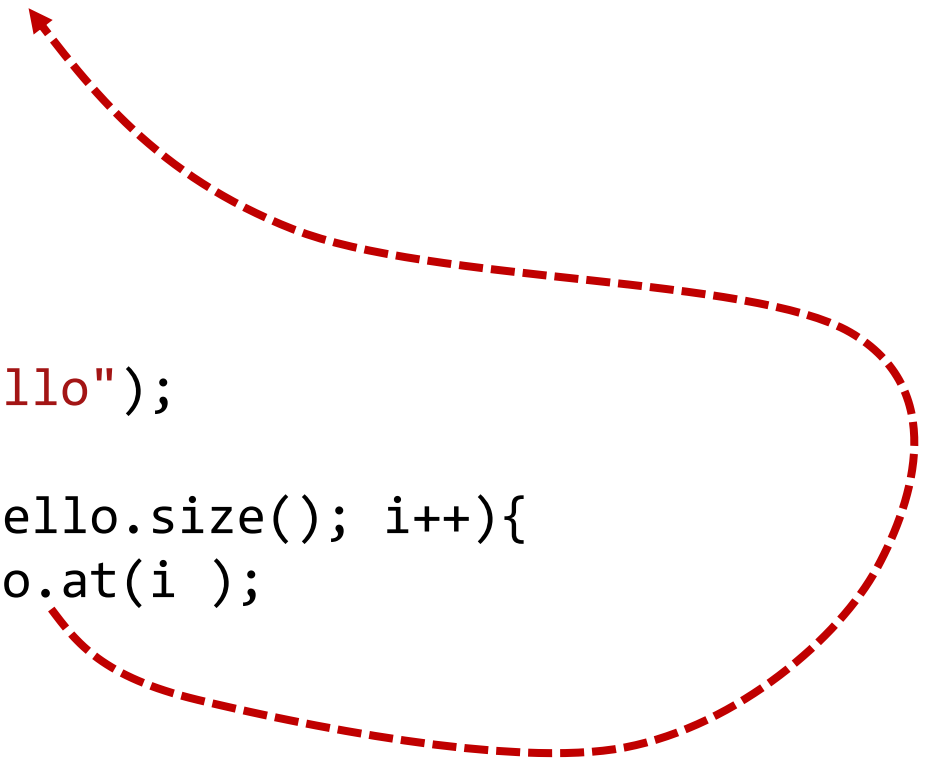


```
int main()
{
    string he = "he" ;
    string llo = "llo" ;

    string hello = add( he , llo );

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

Strings *could* be indexed via `at`

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

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

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



Recap: Results of operations

Many operations are *closed*

the result is the same type as the inputs

Multiplication: $a \times b$

Addition: $a + b$

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

"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

$$+ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

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

$$+ : \texttt{string} \times \texttt{string} \rightarrow \texttt{string}$$

```
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
{
    ...
};

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

bool 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 usually needed in practice
  - Assignment + Comparison
  - Needed to get containers and algorithms to work

Overloads: input/output

The magic of `cin` + `cout`

- We've used `cin` and `cout` extensively
  - Used `<<` to print things
  - Used `>>` to read things
- We can now see that `cin` and `cout` are objects
  - e.g. we can call `cin.fail()`
- Some *non-primitive* types print automatically: how?

```
int main()
{
    string msg="Value is ";
    complex<float> c=(4.0, 1.0);
    cout << msg << c << endl;
}
```

Could print things via an object

Type1 cout;

Some unspecified type that represents a stream of characters

```
void print(Type1 &output, const string &s);


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

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

Could print things via an object

```
Type1 cout;
Type2 endl;

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


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

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

`istream` and `ostream`

Input and outputs are sequences of characters/bytes

`istream`: input sequence `ostream`: output sequence

```
class istream
{
public:
  bool fail() const;

  // look at next character,
  // but don't read it yet
  int peek();

  // read one character
  int get();
};
```

```
class ostream
{
public:
  bool fail() const;

  // Write one character
  // to output
  void put(char c);

  // ...
};
```

Overloading : printing

Pass by non-const reference:
writing to output will change it

Pass by const reference:
printing does not change it

```
String
{
    char size() const;
    char at(int index);
};

void print(ostream &output, const String &s)
{
    for(int i=0; i<s.size(); i++){
        output.put( s.at(i) );
    }
}
```

We really don't care *how* `ostream` *is implemented*
It is something we can write characters too

Overloading : printing

```
String
{
    char at(int index);
};

void print(ostream &output, const String &s)
{
   ...
}

ostream &operator <<(ostream &output, const String &s)
{
    print(output, s);
    return output;
}
```

Why return a reference?

```
void print(ostream &output, const String &s);

ostream &operator <<(ostream &output, const String &s)
{
    print(output, s);
    return output;
}

int main()
{
    String a("a");
    String b("b");
    print(cout, a);
    print(cout, b);
}
```

Why return a reference?

```
void print(ostream &output, const String &s);

ostream &operator <<(ostream &output, const String &s)
{
    print(output, s);
    return output;
}

int main()
{
    String a("a");
    String b("b");
    cout << a;
    cout << b;
}
```

Why return a reference?

```
void print(ostream &output, const String &s);

ostream &operator <<(ostream &output, const String &s)
{
    print(output, s);
    return output;
}

int main()
{
    String a("a");
    String b("b");
    cout << a << b;
}
```

Why return a reference?

```
void print(ostream &output, const String &s);

ostream &operator <<(ostream &output, const String &s)
{
    print(output, s);
    return output;
}

int main()
{
    String a("a");
    String b("b");
    ( cout << a ) << b;
}
```

Why return a reference?

```
ostream &print(ostream &output, const String &s);

ostream &operator <<(ostream &output, const String &s)
{
    print(output, s);
    return output;
}

int main()
{
    String a("a");
    String b("b");
    print( print( cout , a ) , b );
}
```

Overloading : reading

```
String
{
    void push_back(char c);
};

String read(istream &input)
{
    String result;
    while( input.peek()!=0 && !isspace(input.peek()) ){
        result.push_back( input.get() );
    }
    return result;
}
```

Overloading : reading

```
String
{
    void push_back(char c);
};

String read(istream &input)
{
    ...
}

istream &operator >>(ostream &input, String &s)
{
    s = read(input);
    return input;
}
```

When to overload IO: “value” classes

- Some classes represent values
  - Their state completely captures them
  - They can be copied and duplicated
  - *Examples*: `int`, `complex`, `string`, bitmaps, audio data
- Reading and writing allows us to move values
  - *Through space*: write on one machine, read on another
  - *Through time*: write in the past, read in the future
  - It often makes sense to overload << and >>

When to overload IO: “thing” classes

- Some classes represent actual things in the world
  - Their state identifies something in the world
  - They cannot be copied and duplicated
  - *Examples*: cout, cin, a motor, a sensor, a display
- Overloading << may make sense for debug
  - *Robotic arm:* print the current angle and position
  - *Temperature sensor*: print sensor location + temperature
- Overloading >> is likely to be confusing

```
void f(RobotArm &arm)
{
  // What does this mean?
  cin >> arm;
}
```

```
void f(RobotArm &arm)
{
  arm.load_constraints(cin);
}
```

Overloading Assignment

Construction vs assignment

```
class String
{
private:
    int length;
    int capacity;
    char *data;
public:
    String();
    String(const char *s);
    String(const String &s);
};

int main()
{
    String a;
    String b("x");
    String c(b);
    String d=c;

    a = c;
}
```

A fresh string instance is being created
The constructor is called to initialise it.

An ***existing*** instance a is being assigned the value of the c. This is not construction.

Assignment in practise

```
struct MyStringVec
{
private:
    int length;
    int capacity;
    String *data;
public:
    void write(int index, const String &s)
    {
        data[index] = s;
    }
};
```

Overloading assignment : v1

```
class String
{
private:
  int length;
  int capacity;
  char *data;
public:
  String &operator=(const String &s)
  {
    length = s.length;
    capacity = s.capacity;

    data = new char[capacity];

    for(int i=0; i<length; i++){
      data[i] = s.data[i];
    }
    return *this;
  }
};
```

```
int f(String &x)
{
  String y;
  x = y;
}
```

Overloading assignment : v2

```
class String
{
private:
  int length;
  int capacity;
  char *data;
public:
  String &operator=(const String &s)
  {
    delete []data;

    length = s.length;
    capacity = s.capacity;
    data = new char[capacity];
    for(int i=0; i<length; i++){
      data[i] = s.data[i];
    }

    return *this;
  }
};
```

```
int f(String &x)
{
  x = x;
}
```

Overloading assignment : v3

```
class String
{
private:
  int length;
  int capacity;
  char *data;
public:
  String &operator=(const String &s)
  {
    if(this != &s){
      delete []data;
      length = s.length;
      capacity = s.capacity;
      data = new char[capacity];
      for(int i=0; i<length; i++){
        data[i] = s.data[i];
      }
    }
    return *this;
  }
};
```

Overloading assignment : v4

```
class String
{
private:
  vector<char> data;
public:
  String &operator=(const String &s)
  {
    data = s.data;
  }
};
```

Overloading assignment : v5

```
class String
{
private:
  vector<char> data;
public:
  // Use compiler generated
  // default assignment. Let the
  // vector class handle it.
};
```

Making a “full” type

- Certain operations should *always* be considered

```
T::T()
T::T(const T &x)
const T &T::operator=(const T &x)
```

  - These make an object look “normal”
  - Often the compiler default behaviour is fine
- Some operations make objects much more useful

```
bool T::operator<(const T &o) const;
bool T::operator==(const T &o) const;
```

  - Allows us to sort and order objects
  - Not meaningful for all classes
- Everything else depends on context

Making a restricted type

- Sometimes you don't want people copying your type
  - E.g. Objects representing low-level resources; or
  - Trying to represent the idea of uniqueness
- *Simple*: make copy constructor and assignment private

- You are unlikely to want to do it in this course

You may want to do it in other situations

…don't do what you're about to see

```
/* WARNING: direct access to memory-map of
TR1 ZX80. Do not use on a desktop, it will crash.

DO NOT COPY THIS CLASS ONCE CONSTRUCTED
*/
class RawMotor
{
private:
  volatile int *m_peripheral;
public:
  RawMotor()
  {
    // WARNING: Deeply unsafe. This will only
    // work on the TR1 ZX80 robot. Demons here!
    m_peripheral = (int*) 0x80001000;
  }

  void set_speed(int degrees_per_second)
  {
    // WARNING: directly from TR1 ZX80 datasheet
    m_peripheral[1] = degrees_per_second;
  }
};
```

```
/* WARNING: direct access to memory-map of
TR1 ZX80. Do not use on a desktop, it will crash.*/
class RawMotor
{
private:
  volatile int *m_peripheral;

  RawMotor(const RawMotor &);
  const RawMotor &operator=(const RawMotor &);
public:
  RawMotor()
  {
    // WARNING: Deeply unsafe. This will only
    // work on the TR1 ZX80 robot. Demons here!
    m_peripheral = (int*) 0x80001000;
  }

  void set_speed(int degrees_per_second)
  {
    // WARNING: directly from TR1 ZX80 datasheet
    m_peripheral[1] = degrees_per_second;
  }
};
```

Overloading : summary

- Overloading allows multiple functions with same name
  - Each overload must have different input types
- We have two types of overloading
  - Function overloading : multiple defn. of “normal” functions
  - Operator overload: adding new meanings to operators
- Constructor and assignment overloading are important
  - Needed to avoid surprises when working with raw pointers
- Overloading can be powerful, but don’t overuse it
  - Overloading should always “make sense”
  - Don’t overload `to_string` to return a float
  - Don’t overload `*` to mean divide