- What we’ve just seen:
  - Pointers
  - Arrays
  - Dynamic memory
  - Creating a simple data-type
- We’re now going to use them
  - Strings as data-types
  - Using pointers to create linked lists

Strings

Representing text

- Sequences of characters are special to humans
  - Our primary mode of communication is spoken language
  - Our primary form of persistence is written language

Text is a sequence of characters or bytes

- Sequences of bytes are special to computers
  - The primary form of communication is streams of bytes
  - The primary form of storage is sequences of bytes

The char type

C++ has built-in support for both characters and bytes

char

Smallest integer type supported in the machine

and/or

An integer type that can hold a character

Character literal values

The compiler can turn literal characters into numbers

```
char A       = 'A';
char space   = ' ';
char newline = '\n';
char tab     = '\t';
```

Character literals use ***single*** quotes

Characters are rare: usually you use see string literals

Text as sequences of `char`s

We could represent text as a vector of characters

```
vector<char> word={'H','e','l','l','o'};
```

Gives us basic methods for text

- Construct strings from individual characters
- Access characters in the middle using `text[.]`

Text as a `string`

- Text is special : we want more convenience
  - Concatenating strings using +
  - Reading/writing strings to cin/cout
- Text is special : we want to distinguish from bytes
  - When is `vector<char>` just bytes, and when is it text?

- The string type is a fancy version of `vector<char>`
  - Purpose 1 : we get extra functions and IO helpers
  - Purpose 2 : string type is explicitly text, not just bytes

The char type revisited

C++ has built-in support for both characters and bytes

char

Smallest integer type supported in the machine

and/or

An integer type that can hold a character

Smallest int is 8 bits

There are 256 characters

C++ is based on C : it is *old*

- The first version of C is from ***1972***
  - The longevity of C is a huge strength
  - Code from 30 years ago still works

- But: some assumptions and constraints are built in
  - 1972 : The only language is ~~English~~American
  - 1972 : Storage is extremely expensive - $1 per byte

ASCII code

- Codes which represent letters of the alphabet, punctuation marks, and other special characters, as well as numbers, are *alphanumeric* codes.

- The most widely used alphanumeric code is the American Standard Code for Information Inter-change (ASCII). The ASCII code (pronounced "askee") is a seven-bit code.

Character	Seven-Bit ASCII	Octal	Hex	Character	Seven-Bit ASCII	Octal	Hex
A	100 0001	101	41	Y	101 1001	131	59
B	100 0010	102	42	Z	101 1010	132	5A
C	100 0011	103	43	0	011 0000	060	30
D	100 0100	104	44	1	011 0001	061	31
E	100 0101	105	45	2	011 0010	062	32
F	100 0110	106	46	3	011 0011	063	33
G	100 0111	107	47	4	011 0100	064	34
H	100 1000	110	48	5	011 0101	065	35
I	100 1001	111	49	6	011 0110	066	36
J	100 1010	112	4A	7	011 0111	067	37
K	100 1011	113	4B	8	011 1000	070	38
L	100 1100	114	4C	9	011 1001	071	39
M	100 1101	115	4D	blank	010 0000	040	20
N	100 1110	116	4E	.	010 1110	056	2E
O	100 1111	117	4F	(	010 1000	050	28
P	101 0000	120	50	+	010 1011	053	2B
Q	101 0001	121	51	$	010 0100	044	24
R	101 0010	122	52	*	010 1010	052	2A
S	101 0011	123	53	)	010 1001	051	29
T	101 0100	124	54	—	010 1101	055	2D
U	101 0101	125	55	/	010 1111	057	2F
V	101 0110	126	56	,	010 1100	054	2C
W	101 0111	127	57	=	011 1101	075	3D
X	101 1000	130	58	⟨RETURN⟩	000 1101	015	0D
				⟨LINEFEED⟩	000 1010	012	0A

C++ and Unicode

- Human text should really be stored as Unicode
  - Unicode aims to represent all human characters
  - Each character needs 32 bits
  - But: the notion of “character as a number” is a bit false
- C++ has only limited support for Unicode
  - The type `wchar_t` represents a “wide” character
  - The type `wstring` represents a “wide” string
  - Library support for Unicode is very limited

On this course we use `string` to represent text

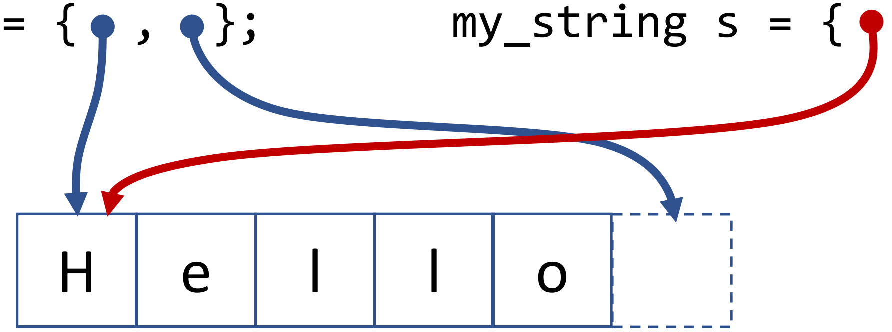
... while recognising it isn’t really good enough

C and null-terminated strings

The C++ type string is a bit like a vector

```
struct my_string
{
    char *begin;
    char *end;
};
my_string p = {  ,  };
```

```
struct my_string
{
    char *data;
    int size;
};
my_string s = {  , 5 };
```

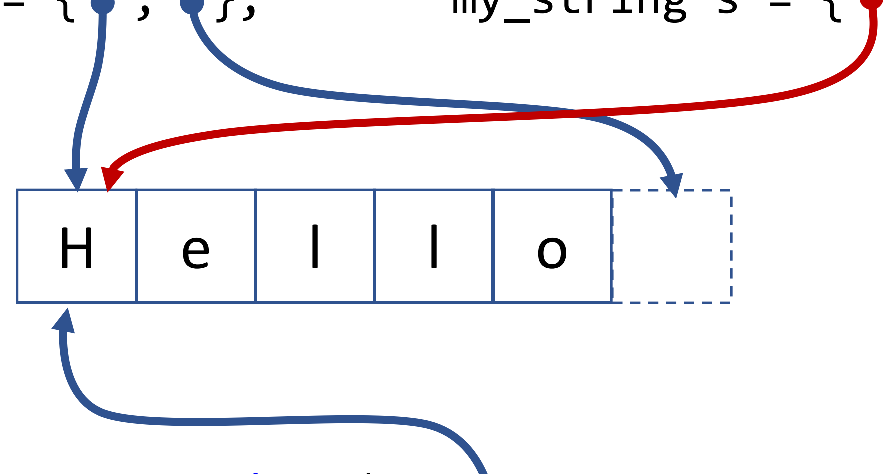


C and null-terminated strings

The C type for string is usually just a pointer

```
struct my_string
{
    char *begin;
    char *end;
};
my_string p = {  ,  };
```

```
struct my_string
{
    char *data;
    int size;
};
my_string s = {  , 5 };
```



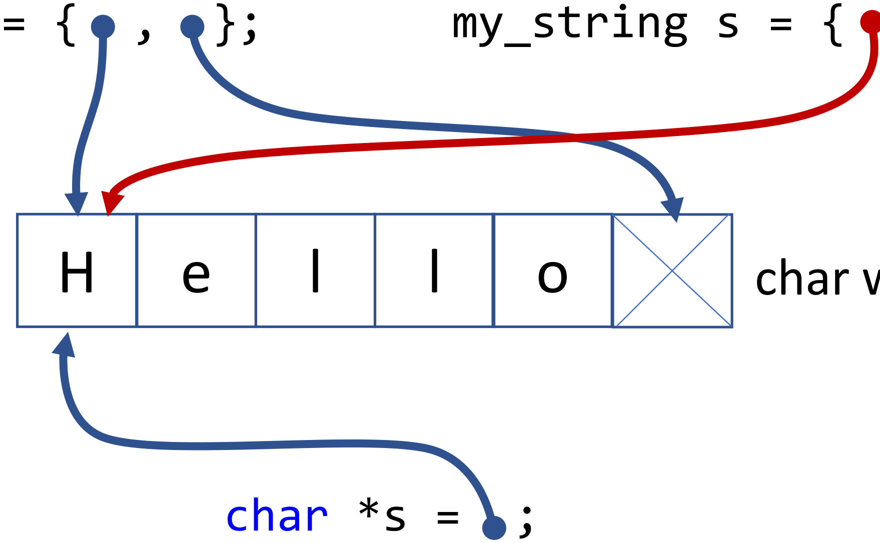

```
char *s =  ;
```

C and null-terminated strings

A C string must be null-terminated to define length

```
struct my_string
{
    char *begin;
    char *end;
};
my_string p = { , };
```

```
struct my_string
{
    char *data;
    int size;
};
my_string s = { , 5 };
```



char with numeric value 0

```
char *s = ;
```

Length-prefix vs null-terminated

- There are two main string styles:
  - Length prefixed: sequence of characters and a length
  - Null terminated: sequence of characters ending with null

```
int length(my_string *s)
{
  return s->size;
}
```

Null terminated is worse

- Slow
- Error prone
- A big security risk

```
int length(char *s)
{
    int count=0;
    while( *s != 0 ){
        count++;
        s++;
    }
    return count;
}
```

C style strings in practice

- Null terminated strings are (sadly) built in to C
- String literals are actually null terminated

```
int main()
{
    char *hello = "Hello";
    while(*hello){
        cout << *hello;
        hello++;
    }
}
```

Mixing C and C++ strings

- The string class treats `char*` as a string
  - Will automatically convert if assigned

```
int main()
{
    char *hello="Hello";

    string x=hello;
    x="Goodbye";
}
```

It is strongly suggested you always use string

C strings and program arguments

Our initial main function:

```c
int main()
{
    // Your code here
}
```

C strings and program arguments

Main function with explicit program return code

```c
int main()
{
    // Your code here
    return 13; // Return an exit code
}
```

C strings and program arguments

Main function with program arguments

```
int main(int argc, char **argv)
{
    // Your code here
    return 13; // Return an exit code
}
```

This prototype is inherited from C

- `argc` : The number of arguments passed
- `argv` : An array of argument values

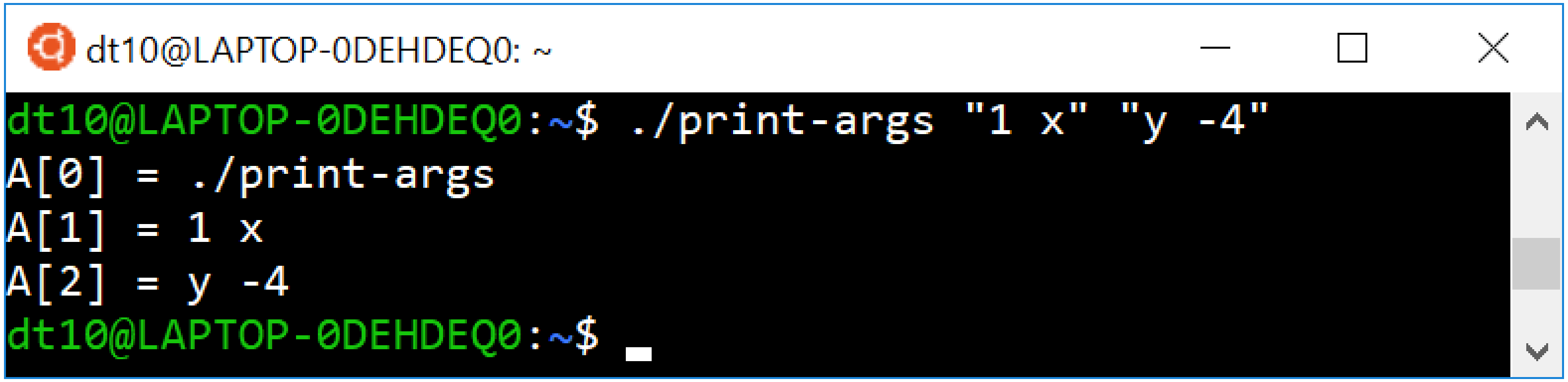
Passing arguments to a program

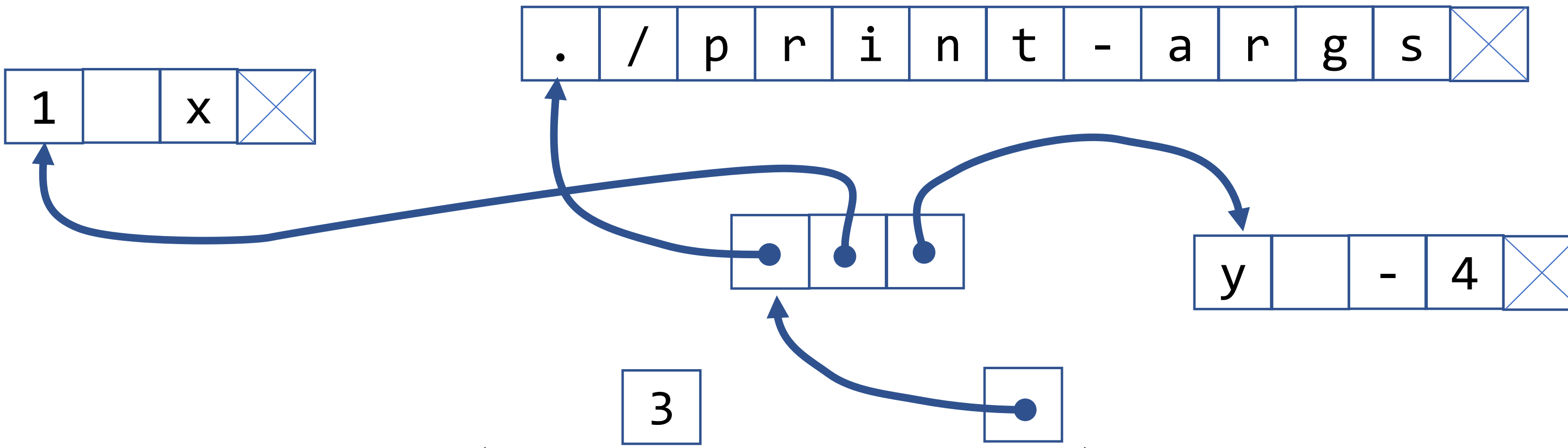
```
int main(int argc, char **argv)
{
  for(int i=0; i<argc; i++){
    string arg=argv[i];
    cout << "A["<<i<<"] = " << arg << endl;
  }
}
```

dt10@LAPTOP-0DEHDEQ0: ~

```
dt10@LAPTOP-0DEHDEQ0:~$ g++ print-args.cpp -o print-args
dt10@LAPTOP-0DEHDEQ0:~$ ./print-args
A[0] = ./print-args
dt10@LAPTOP-0DEHDEQ0:~$ ./print-args 1 x y
A[0] = ./print-args
A[1] = 1
A[2] = x
A[3] = y
dt10@LAPTOP-0DEHDEQ0:~$ ./print-args "1 x" "y -4"
A[0] = ./print-args
A[1] = 1 x
A[2] = y -4
dt10@LAPTOP-0DEHDEQ0:~$ _
```

```cpp
int main(int argc, char **argv)
{
  for(int i=0; i<argc; i++){
    string arg=argv[i];
    cout << "A["<<i<<"] = " << arg << endl;
  }
}
```






```
int main(int argc, char **argv)
{
  for(int i=0; i<argc; i++){
    string arg=argv[i];
    cout << "A["<<i<<"] = " << arg << endl;
  }
}
```

Passing arguments to a program

```
// Our main with nice arguments
int my_main(vector<string> argv)
{
    // ...
}

// Raw main with C arguments
int main(int argc, char **argv)
{
  vector<string> args;
  for(int i=0; i<argc; i++){
    args.push_back(argv[i]);
  }

  return my_main(args);
}
```

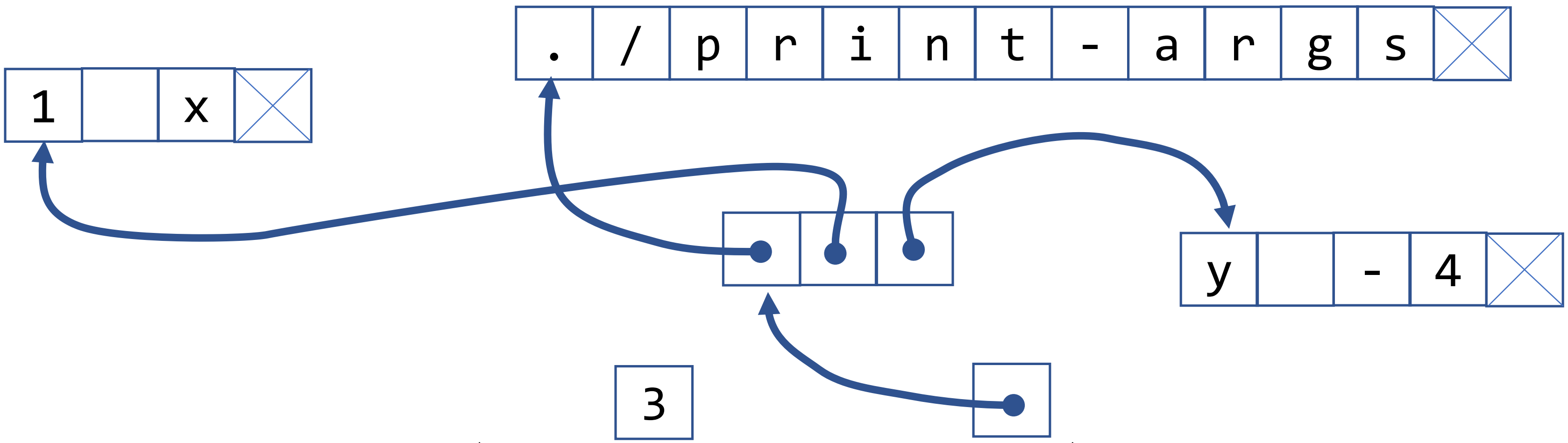
Can now write real commands

- Try to separate program inputs into two sets
  - Parameters: options changing the program behavior
  - Input data: data to be transformed or modified

- We now have four IO channels
  - Program parameters: affect program execution
  - Input data (stdin) : incoming data to be processed
  - Output data (stdout) : outgoing processed data
  - Diagnostics (stderr) : information *about* the processing
  - (Plus the program return code)

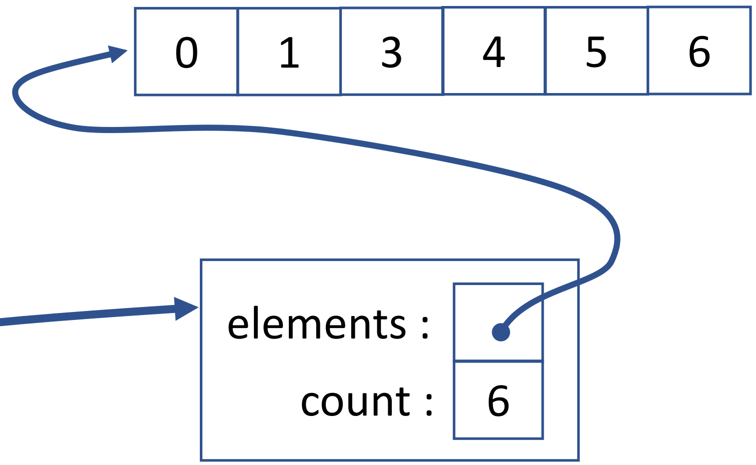
Linked data

Program arguments as linked data




```
int main(int argc, char **argv);
```

my_int_vec as linked data




```
struct my_int_vec{
    int *elements;
    int count;
};

int main()
{
    my_int_vec * v = iota(6);
}
```

Linked data: pointers link instances

- Some data is dense and regular
  - A matrix : mapping from (x,y) to number
  - An image : mapping from (x,y) to colour
  - Audio data : mapping from (t) to intensity

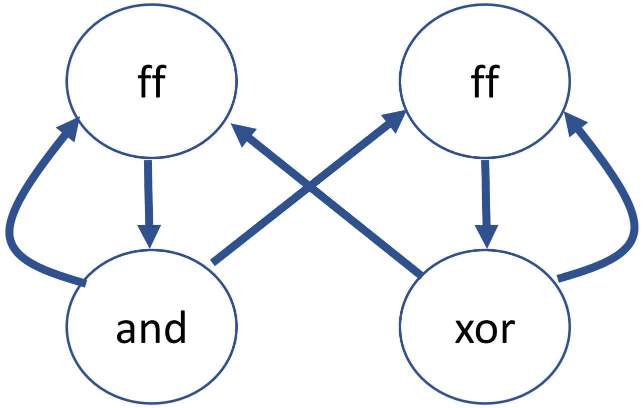
  What value is at a particular co-ordinate?

- Lots of data is sparse and irregular
  - A digital circuit : a set of gates and connections
  - An equation : a tree of operators and values
  - Social network : a set of people and interactions

  How are the parts linked to each other?

An example: modelling a circuit

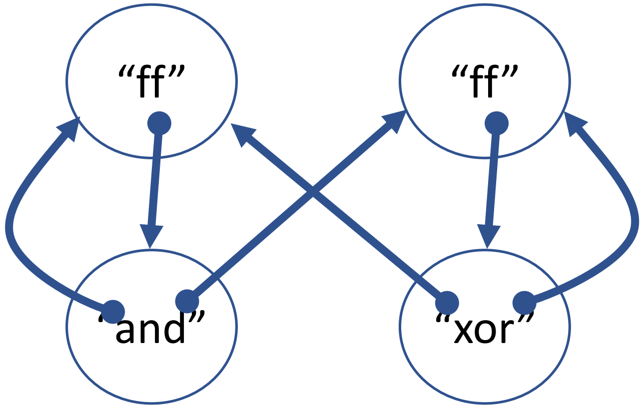
- We want to model a circuit containing logic gates
  - All logic gates have the same C++ type
  - Each gate has a function: “and”, “or”, “ff”, …
  - Each gate represents one output
  - Each gate has zero or more inputs




```
struct logic_gate
{
    string function;
    vector<logic_gate*> inputs;
};
```

An example: modelling a circuit

- We want to model a circuit containing logic gates
  - All logic gates have the same C++ type
  - Each gate has a function: “and”, “or”, “ff”, …
  - Each gate represents one output
  - Each gate has zero or more inputs




```
struct logic_gate
{
    string function;
    vector<logic_gate*> inputs;
};
```

An example : building a circuit

```
struct logic_gate
{
    string function;
    vector<logic_gate*> inputs;
};

int main()
{
    // Create flip-flops
    logic_gate ff1{"ff"};
    logic_gate ff2{"ff"};

    // Create logic gates and set inputs
    logic_gate xor1{"xor", {&ff1, &ff2} };
    logic_gate and1{"and", {&ff1, &ff2} };

    // Set the logic gate inputs
    ff1.inputs.push_back(&xor1);
    ff2.inputs.push_back(&and1);
}
```

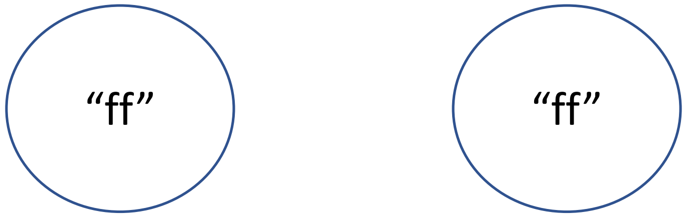
An example : building a circuit

```
struct logic_gate
{
    string function;
    vector<logic_gate*> inputs;
};

int main()
{
    // Create flip-flops
    logic_gate ff1{"ff"};
    logic_gate ff2{"ff"};

    // Create logic gates and set inputs
    logic_gate xor1{"xor", {&ff1, &ff2} };
    logic_gate and1{"and", {&ff1, &ff2} };

    // Set the logic gate inputs
    ff1.inputs.push_back(&xor1);
    ff2.inputs.push_back(&and1);
}
```



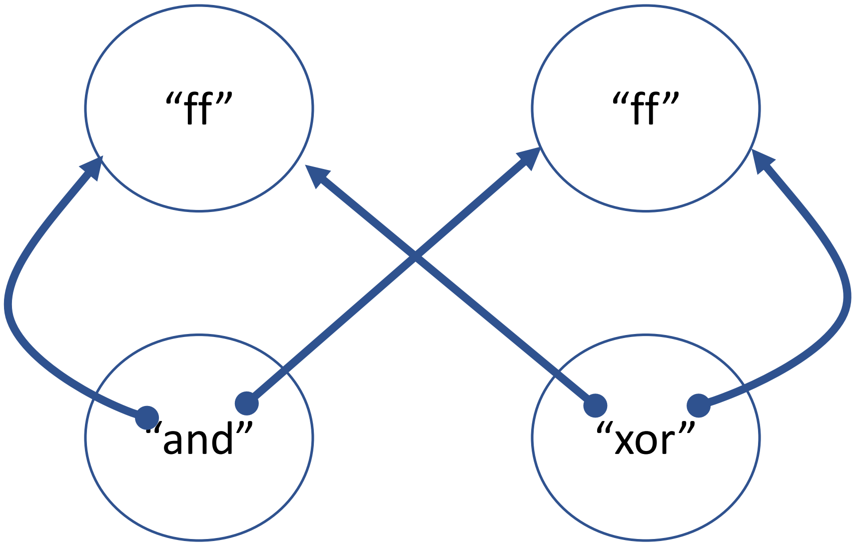
An example : building a circuit

```
struct logic_gate
{
    string function;
    vector<logic_gate*> inputs;
};

int main()
{
    // Create flip-flops
    logic_gate ff1{"ff"};
    logic_gate ff2{"ff"};

    // Create logic gates and set inputs
    logic_gate xor1{"xor", {&ff1, &ff2} };
    logic_gate and1{"and", {&ff1, &ff2} };

    // Set the logic gate inputs
    ff1.inputs.push_back(&xor1);
    ff2.inputs.push_back(&and1);
}
```



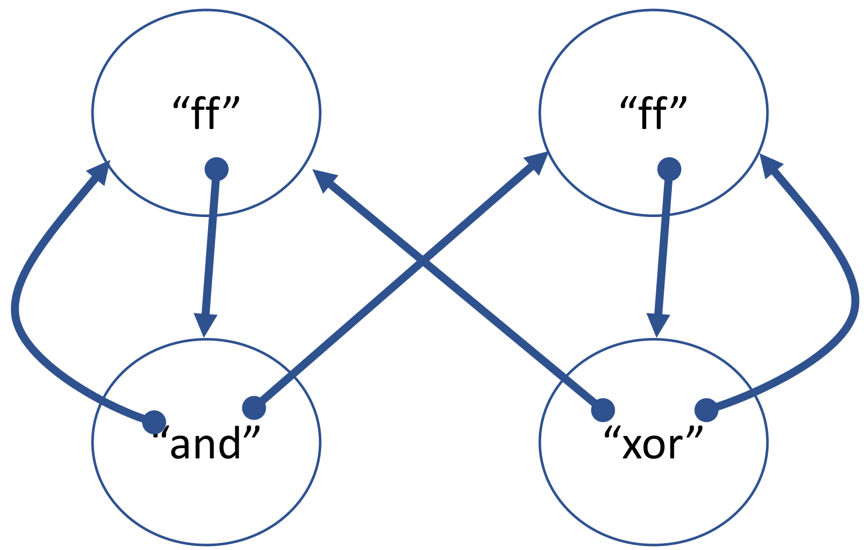
An example : building a circuit

```
struct logic_gate
{
    string function;
    vector<logic_gate*> inputs;
};

int main()
{
    // Create flip-flops
    logic_gate ff1{"ff"};
    logic_gate ff2{"ff"};

    // Create logic gates and set inputs
    logic_gate xor1{"xor", {&ff1, &ff2} };
    logic_gate and1{"and", {&ff1, &ff2} };

    // Set the logic gate inputs
    ff1.inputs.push_back(&xor1);
    ff2.inputs.push_back(&and1);
}
```



Choices in data model design

Linking to instances to versus containing instances

```
struct logic_gate
{
    string function;
    vector<logic_gate*> inputs;
};

logic_gate ff1{"ff"};
logic_gate ff2{"ff"};

logic_gate xor1{"xor", {&ff1, &ff2} };
logic_gate and1{"and", {&ff1, &ff2} };

ff1.inputs.push_back(&xor1);
ff2.inputs.push_back(&and1);
```

```
struct logic_gate
{
    string function;
    vector<logic_gate> inputs;
};

logic_gate ff1{"ff"};
logic_gate ff2{"ff"};

logic_gate xor1{"xor", {ff1, ff2} };
logic_gate and1{"and", {ff1, ff2} };

ff1.inputs.push_back(xor1);
ff2.inputs.push_back(and1);
```

Data models the world

- We need to create representations of “stuff”
  - We can only compute transformations of data
  - So everything needs to be turned into data

- Modelling a problem is the key to solving it
  - The data model can make it easy or impossible

- You know enough already to build complex models
  - The limit is just experience of *how* to design models
  - Things like OOP make this easier, but are not required

Linked Lists : intro

The problems with vector

Vector has worked well for us so far

- Dynamically sized
- Can contain any type
- Convenient access functions

But there are some things it can't (easily) do

We have `push_back` but where is `push_front`?

A naive push_front

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

A naive push_front

4	3	8	1	4	3	8	1	7

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

A naive push_front

4	3	8	1	4	3	8	1	7	0

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

We're ignoring the cost of resize here: see last weeks lab for discussion of efficiency

A naive push_front

4	3	8	1	4	3	8	1	7	7

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

A naive push_front

4	3	8	1	4	3	8	1	1	7

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

A naive push_front

4	3	8	1	4	3	8	8	1	7

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

A naive push_front

4	4	3	8	1	4	3	8	1	7

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

A naive push_front

6	4	3	8	1	4	3	8	1	7

```c
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```cpp
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

Functionality versus performance

- The ***functionality*** of push_front is fine
  - It does exactly what we want
- The ***performance*** of push_front is terrible
  - It takes n operations to push_front 1 item into a vector
  - It takes n^2 operations to push_front n items into a vector
- The API of `vector<T>` is carefully designed
  - Exposes everything the vector is good at
  - Tries to hide or make difficult the weak operations

An alternative : `list<T>`

- The C++ library contains multiple ***containers***
  - Each container provides different functionality
  - Each container provides different performance

- An example is `list<T>` : manages a sequence of `T`
  - Has an efficient implementation of `push_front` built in
  - ***But***: there is no array-like indexing through `[.]`

- Selecting the right container can be important
  - Use vector<T> -> program takes one year
  - Use list<T> -> program takes one second

  Or for another application it could be the opposite

Implementation of `list<T>`

Internally list<T> is implemented as a ***linked list***

A linked list consists of nodes, where each node has:

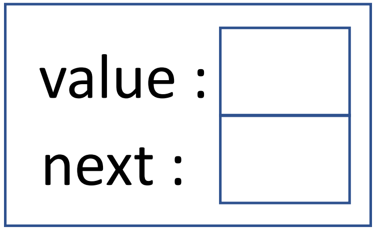
- A value
- A pointer to the next node

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

Technically list<T> is a doubly linked list, but that's not relevant here.

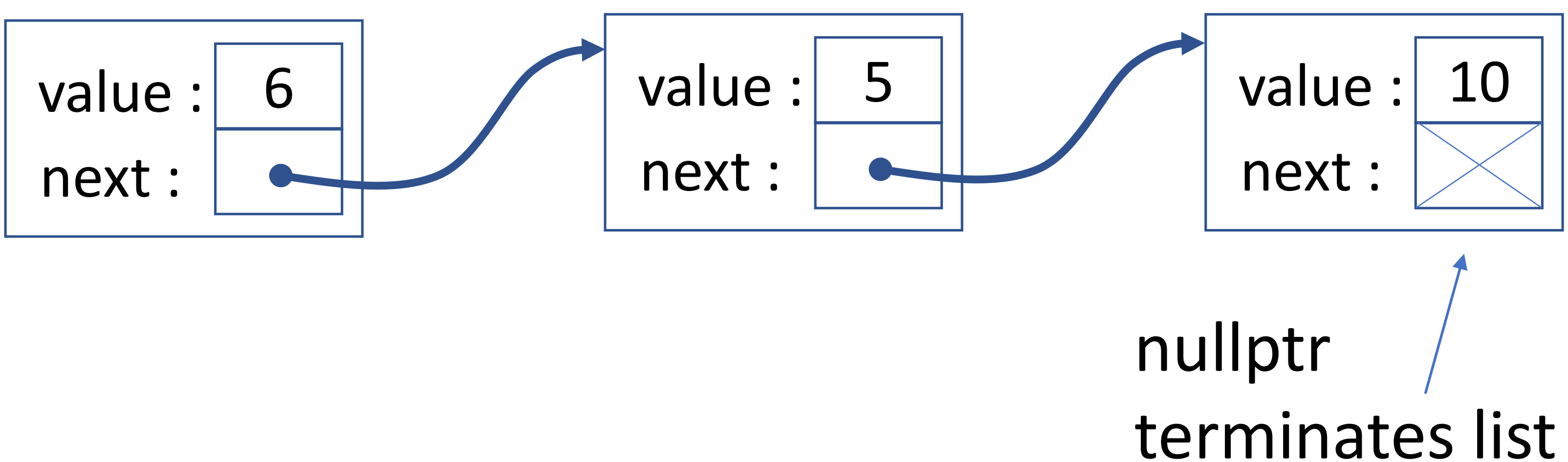
Linked lists : chains of nodes

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```



Linked lists : chains of nodes

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```



Next time

- Linked lists continued
- Lists as a model for source control
- Trees