Outline

- Standard library objects
  - Storing data in vectors
  - strings
  - algorithms
- More on functions
  - Overloading
  - Default arguments
  - Argument passing
- Memory and pointers
Vectors

We can store a list of numbers or objects in a `std::vector`.

- First we include the header: `#include <vector>`
- To declare a vector we specify the type of the variables it contains, and (optionally) the number of elements and their initial value, or a list of values:

```cpp
// vector of doubles, 5 elements, all initialised to 1.0
std::vector<double> terms(5, 1.0);

// vector of doubles, 5 elements, each specified (C++11)
std::vector<double> vec = {1.0, 1.5, 0.3, 2.7, 3.6};

// empty vector of integers (no elements)
std::vector<int> indices;
```
Vectors

- The syntax `[i]` accesses the `i`th element of the vector.
  
  \[
  \begin{align*}
  \text{terms[0]} &= 2.0; & \text{// set first element} \\
  \text{terms[4]} &= \text{terms[0]} + 1; & \text{// set last element}
  \end{align*}
  \]

- Note that for a vector of length `N`, `i` runs from `0` to `N - 1`.

- Accessing elements outside the range `0, \ldots, N - 1` is undefined behaviour: this likely causes a crash or incorrect results.

  \[
  \begin{align*}
  \text{// vector of doubles, 5 elements, each specified} \\
  \text{std::vector<double> vec = \{1.0, 1.5, 0.3, 2.7, 3.6\}};
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{std::cout} &\ll \text{vec[5]} \ll \text{std::end}; \text{// undefined behaviour!} \\
  \text{std::cout} &\ll \text{vec[-1]} \ll \text{std::end}; \text{// undefined behaviour!}
  \end{align*}
  \]
#include <vector> // need the <vector> header

int main()
{
    int numTerms = 20;

    // declare a vector of ints, with numTerms elements, 
    // and an initial value of 1 for all the elements  
    std::vector<int> fib(numTerms, 1);

    // loop from the third element (i=2)  
    // to the last element (i = nTerms-1)  
    for (int i=2; i<numTerms; i++)
        fib[i] = fib[i-1] + fib[i-2];

    return 0;
}
Vectors

For a vector `std::vector<int> vect(numTerms);`

- `vect.size()` returns the number of elements
- `vect.push_back(10);` increases the number of elements in `fib` by one, and sets the final element to 10
- `vect.resize(100);` sets the number of elements to 100,

```cpp
#include <vector>
#include <iostream>
int main()
{
    std::vector<double> vect(50);
    // vect.size() returns 50
    vect.push_back(4.0);
    // vect.size() returns 51
    vect.resize(25);
    // vect.size() returns 25
    return 0;
}
```
Standard library containers and algorithms

- `std::vector` is one of several data containers defined in the standard library. Others include
  - `std::list` (header `<list>`)  
  - `std::map` (header `<map>`)  
  - `std::set` (header `<set>`)  

- Various algorithms that operate on data in containers are defined in the `<algorithm>` header  
  - `std::sort` sorts a container  
  - `std::find` finds particular elements in a container  
  - `std::max_element`, `std::min_element` return maximum and minimum elements  

- These algorithms access data in containers through *iterators*

See *C++ Primer*, chapters 9, 10, 11
Standard library strings

The `std::string` type stores and manipulates text strings:

```cpp
#include <string>   // for std::string
#include <iostream> // for std::cout
int main()
{
    std::string s1("hello"), s2("world"); // define some strings
    std::string s3 = s1 + s2; // concatenate strings
    int length = s3.size(); // get length of string
    s3[0] = 'H'; // set first character to 'H'

    // find index of first vowel in string
    size_t index = s3.find_first_of("aeiou");

    if (index != std::string::npos) // if we found a vowel...
        std::cout << s3[index] << std::endl; // ...display it
    return 0;
}
```

See C++ Primer, Chapter 3
Recall the syntax for a function:

```c
double power(double v, int n)
{
    double r = 1;
    for (int i = 0; i < n; i++) r *= v;
    return r;
}
```

- **Name** (power)
- **Parameters** (double v, int n)
- **Return value** (double)
- **Body** ({ ... })

Function is executed ('called') with

```c
power(3.5, 2);
```
Function overloading

Functions can share a name, if their arguments differ

- Computers usually evaluate $a^b$ as $\exp(b \times \log(a))$
- But if $b$ is an integer, on some CPUs $a^b$ may be found more efficiently by repeated multiplication

Consequently, C++ has more than one function\(^1\) for calculating $a^b$:

```cpp
double pow(double a, double b); // slow algorithm for general b
double pow(double a, int b); // faster algorithm for integer b
```

When calling these functions, C++ selects the appropriate\(^2\) function based on the parameters passed:

```cpp
pow(1.5, 2); // calls pow(double, int)
pow(1.5, 2.5); // calls pow(double, double)
```

\(^1\) in the `<cmath>` header

Default arguments

Function arguments can take default values

```cpp
// function to return the log (base b) of n
// b has a default value of 10.0
double logn(double a, double b=10.0)
{
    return std::log(a)/std::log(b);
}
```

logn(5.5, 2.1); // evaluates to log of 5.5 base 2.1
logn(5.5); // evaluates to log of 5.5 base 10

- To use the default value, the parameter is omitted when the function is called
- Default arguments must come last in the argument list
- Defaults should be specified in the function declaration only, if the declaration is separate from the definition
Passing arguments by value

The function parameters we have seen so far are passed by value:

- Copies of each parameter are made when the function is called
- Code in function body can only access these copies
- The copies are destroyed when the function returns

```c
void func(int i)
{
    // i is a copy of p. Nothing we do to i here can affect p
}
int main()
{
    int p = 5;
    func(p);
}
```

Passing by value means that:

- Large parameters are copied and destroyed, which takes time
- We cannot modify parameters outside the function
Passing arguments by reference

Sometimes we might want a function to:

- Return multiple values by altering function parameters
- Take a large object as a parameter without copying it

We can do this by passing parameters by reference:

```c
void func(int &i) // note the &
{
    i = 0; // i here is a reference to p
}
int main()
{
    int p = 5;
    func(p);
    // p is now set to zero!
}
```
Passing arguments by reference – examples

‘Returning’ more than one value from a function by altering parameters

```cpp
// A function to normalise the components of a 2-D vector
void Normalise(double &x, double &y)
{
    double d = std::sqrt(x*x + y*y); // calculate norm
    x /= d; // divide x component by norm
    y /= d; // divide y component by norm
}

int main()
{
    double vx = 3.0, vy = 4.0;
    Normalise(vx, vy);
    // now vx is 0.6 and vy is 0.8

    Normalise(3.0, 4.0); // error: we can't modify "3.0"
}
```
Passing arguments by reference – examples

Passing a large object by reference to avoid copying:

```cpp
// A function to 'zero out' small components of an n-D vector
void Threshold(vector<double> &vect, double threshold)
{
    double d = 0.0;
    for (int i=0; i<vect.size(); i++)
    {
        if (vect[i] < threshold && vect[i] > -threshold)
            vect[i] = 0;
    }
}

int main()
{
    vector<double> v(1000000, 2.0); // v is ~8 megabytes
    Threshold(v, 1.5); // Alter v 'in place' without copying it
}
```
Passing arguments by const reference

- Often we need to pass a large object to a function, but this object does not need to be modified.
- In this case we use a *const reference*: the parameter is not copied, and the compiler ensures that no changes are made to the parameter within the function.

```cpp
double Norm(const vector<double> &vect) // note: const and &
{
    // we cannot modify vect in this function
}
```

Summary:
- For ‘input’ parameters that are not modified:
  - Pass small objects by value
  - Pass large objects by const reference
- For ‘output’ parameters that are modified: pass by reference
Memory – a recap

- All the variables in our program are stored in memory (RAM).
- Each variable occupies a separate location in memory, identified by its address.
- Memory is conceptually like a set of numbered pigeon-holes, each able to hold one byte (8 bits) of information.

What happens when we declare a variable?

```java
double a = 100.0;
```

The compiler:

- finds 8 consecutive free bytes of memory and allocates them
- sets this allocated memory to the bits corresponding to the double value 100.0
Memory

- We can find the address of the (first of the 8 bytes of) memory allocated for \(a\) by using the \textit{address-of} operator, \&:

  ```cpp
double a = 100.0;
std::cout << &a << std::endl;
```

- The output of this program is a address (addresses displayed in hexadecimal)

  0x7652755c1528

- In C++ we can store memory addresses in \textit{pointers}:

  ```cpp
double a = 100.0; // declare a double "a"
double *pa = &a;  // declare a pointer to a double,
                    // initialised with the address of "a"
```

- The type \texttt{double*} stores just a memory address, but the type indicates that the data at this address should be interpreted as a \texttt{double}.
Pointers

• We can evaluate the value at a memory address using the dereference or “value-at” operator `*`:

```c++
double a = 100.0; // declare a double "a"
double* pa = &a; // declare a pointer to a double, // initialised with the address of "a"
std::cout << (*pa); // output the "value at" pa
```

• Here `pa` stores the address where the variable `a` is stored. Evaluating the “value at” this address using `*pa` gives us back the value of `a`, 100.0.
Pointers

• We define two variables, with a pointer to each of them:

```c
double a = 1.0, b = 2.0;
double *pa = &a, *pb = &b;
```

• What is the result of the expression:

```c
pa = pb;
```
This sets the address stored in `pa` to the address stored in `pb`. The result is that both `pa` and `pb` contain the address of `b` (we say they both point to `b`).

• What is the result of:

```c
*pa = *pb;
```
Recall that `*` means “value at” so we read this as “set the value at address `pa` to the value at address `pb`”. This sets `a` to `2.0`. 
When are pointers used?

- Interfacing with C code:
  - C does not have references, so functions can only pass by value. Passing a pointer by value is similar to passing by reference.
  - C does not have containers like `std::vector` for storing multiple objects: instead, a region of memory is allocated manually and accessed through a pointer.

- Implementing data structures and algorithms:
  - E.g. a `linked list` container (implemented in `std::list`), which can rapidly delete and insert large elements into the middle of the list (unlike a `std::vector`).
  - Successive elements of a linked list can be scattered throughout memory. Each element contains a pointer to the next element and a pointer to the previous element.
Further reading about pointers and memory

- More on pointers and *arrays*
  - see *C++ Primer, Chapter 4.3*

- Dynamic memory allocation (*new* and *delete*):
  - see *C++ Primer, Sections 4.3.1 and 5.11*
  - See in particular the box “Caution: Managing dynamic memory is error-prone”

- Dynamic memory in modern C++
  - Current best practice is to use C++11 ‘smart pointers’
    (std::unique_ptr, std::shared_ptr, std::weak_ptr)
  - see *Effective Modern C++ (S. Meyers), Chapter 4*
Summary

- **std::vectors** (C++ Primer §3.3)
  - Used for storing multiple numbers/objects
  - One of several containers in the standard library
- Overloaded functions (C++ Primer §7.8)
- Default function arguments (C++ Primer §7.4)
- Function arguments: (C++ Primer §7.2)
  - by value (makes a copy of the parameter; used for small objects; can’t modify parameter outside function)
  - by reference (for ‘output parameters’)
  - by const reference (for large objects, when we don’t want to modify parameter)
- Pointers (C++ Primer §4.2)
  - Pointers store the address of a variable
  - Indirection operators: ‘address of’ & and ‘value at’ *