Part IV

Functions

Outline

Functions

- A vital part of many programming languages is the use of functions
- These are fairly similar to mathematical functions
- Given a set of inputs, the function will produce a result
- The output of the function should not depend on any previous calls that have been made to it.
- The inputs to a function are called “parameters”
- The ouput from a function is called its return value.

Functions

- Every C++ function has a return-type
- This may be `void` implying nothing to return
- Functions cannot be declared inside another function.

```
double quadraticSoln(double a, double b, double c){
  return (-b + sqrt(b*b - 4*a*c)) / (2*a);
}

void printDouble(int x){
  std::cout << "Twice " << x << " is " << 2*x << std::endl;
}

int main(void){
  double x = quadraticSoln(1, -3, 2);
  printDouble(5);
  return 0;
}
```

Functions ctd

A function can have multiple return points:

```
double quadraticSoln(double a, double b, double c){
  if( fabs(a) < 1e-8 ){ // it's a linear eqn
    return -c/b;
  }
  else{
    return (-b + sqrt(b*b - 4*a*c)) / (2*a);
  }
}
```

and the function returns as soon as it reaches one.

Recursive functions

- Recursive functions are permitted:

```
int factorial(int n){
  if( n > 1 ){
    return n * factorial(n-1);
  }
  else{
    return 1;
  }
}
```

  is a valid function.
- A new copy of the function is placed on the stack for each iteration, with distinct local variables (if any).
- If infinite/high recursion occurs, then the computer may run out of stack space.
- The above is not a particularly good example of recursion use.

Outline

Passing by value

- Parameters are passed to functions by value by default

  ```
  int sum(int a, int b){
    a += b;
    return a;
  }
  int main(void){
    int x = 3, y = 5;
    std::cout << "sum=" << sum(x, y) << " ";
    std::cout << "x=" << x << " y=" << y << std::endl;
    return 0;
  }
  ```

  will print

  `sum=8 x=3 y=5`

  i.e. the value of `x` has not been changed by `sum()`.
- The value of `x` has been copied into the variable `a`
- This copy requires some work on the part of the computer.

Passing by reference

- Parameters can be made to be passed to functions by reference i.e. the function being called has a reference to the value that was passed, rather than a copy, potentially saving time.
- Passing `a` by reference from the previous example gives:

```
int sum(int &a, int b){
  a += b;
  return a;
}
int main(void){
  int x = 3, y = 5;
  std::cout << "sum=" << sum(x, y) << " ";
  std::cout << "x=" << x << " y=" << y << std::endl;
  return 0;
}
```

  will print

  `sum=8 x=8 y=5`

  i.e. the value of `x` has been changed by `sum()`.
- The variable `a` referred to the same location in memory as `x`

Pass-by-reference

Further example of passing by reference

```
void doubleMe(int& x){
  x *= 2;
}
int main(void){
  doubleMe(5); // Won't compile: can't take reference of "5"
  int g = 5;
  doubleMe(g); // Will compile
  std::cout << g << std::endl; // Will output 10
  return 0;
}
```

Notes on pass-by-reference

- Passing by value is usually preferred
- With pass-by-reference, a function call may alter the value of a variable, which is not evident from looking at the function-call alone
- Pass-by-reference can cause non-obvious aliasing:

```
int f(int &a, int &b){
  a += 1;
  b += 1;
  return a + b;
}

int x = 5;
int y = f(x, x); // y == 14
```

- Pass-by-reference is useful in two main cases:
  - When a function needs to return more than one value (e.g. its function is to update a variable and return an error-code)
  - When the object being passed is expensive to copy (in which case a `const int& x` should be used instead - see later).

l-values and r-values

- An l-value is roughly "something that exists in memory", i.e. it can have its reference taken
- The name l-value comes from being on the left-hand-side of an assignment
- An r-value is an expression that does not exist in a named memory location
  It may be a literal value (e.g. 5),
  or the result of an operation (e.g. `x + y`),
  or the result of a function (e.g. f(x))
- The name r-value comes from being on the right-hand-side of an assignment

Outline

Default function parameters

It is possible to have default values for function parameters:

```
int sumNos(int a, int b=2, int c=3){
  return a + b + c;
}

std::cout << sumNos(1, 3, 5) << sumNos(1,3) << sumNos(1) <<
    std::endl;
```

will output "976" because the non-specified values are filled in by default.

Default parameters ctd

- Default parameters must always come at the end of the parameter list
- If one default parameter is specified, then all default parameters before it must also be specified

```
int f(int a=0, double b, double c){ // Compile-error
}

double g(int a, double b=0, double c=1){
  // Code here
}

g(3, 2); // Always means a=3, b=2
g(3, 0, 2); // Only way to set c=2, even though b takes its
    default value
```

Function prototypes

- If we have mutually-dependent functions `f` and `g`, each of which may call the other, then we need to do the following:

```
int f(int); // Function declaration
int g(int); // Function declaration

// Function definition
int f(int x){
 // Computation requiring g to be called
}

// Function definition
int g(int y){
 // Computation requiring f to be called
}
```

- We assume that infinite recursion will not result
- In order that each function knows what sort of function the other is, each function has to have been declared before it is called, but not necesarily defined.

Outline

Const-ness

- Variables can also be declared `const`
  i.e. they cannot be changed after their creation.

```
const int a = 5;
a = 6; // Will cause compile failure
const int b; //  Will cause compile failure — uninitialized
```

- Useful for guarding against programmer-error
- Note that `const int x = 8;` means that `x` is an l-value, but is not modifiable because it is `const`

Const function parameters

- One of the reasons for using pass-by-reference was that copying an object was expensive.
- In order to reduce the potential for unintended altering of such an object, the following is common:

```
int f(const MyLargeObject& a){
 // Use a but don't alter it
}
```

- Thus, the expense of copy is (usually) avoided, and the function is prevented from changing it.
- The compiler *may* make a copy if it wishes, but need not do so.
- This also indicates to another programmer that the function *does not* change non-static members of `a` at all (without having to read the function in detail).

Static variables

- Ideally, the result of a function should be independent of any previous calls to that function
- However, there are cases when the internal workings of a function might depend on previous calls
- The `static` keyword causes the variable's value to persist between function calls
- The variable is only initialised the first time it is seen.
- (Care should be taken for threaded applications, static is not thread-safe).

Static variable example

```
double f(double x){ // x assumed +ve
  static double previousX = -1;
  static double previousAnswer = 0;
  if( previousX == x ){
    return previousAnswer;
  }
  // Expensive calculation
  previousX = x;
  previousAnswer = result;
  return result;
}
```

- The above code implements a simple caching optimization.
- If the same value is passed to the function multiple times, the calculation is only carried out the first time.
- This does not contradict the idea that a function should be independent of its parameters; the result remains the same, even if the computation performed is different.

Outline

Enums

- Suppose you have three options for an ODE solver: Euler, RK2, and RK4.
- How can you store this choice in a variable?
- Could use a coding scheme: $0 \implies$ Euler, $1 \implies$ RK2, $2 \implies$ RK4
- This relies on you (and later users of your code) remembering the scheme and sticking to it
- Could define global variables `int Euler = 0` etc.
- None of these prevent `int myScheme = 3;` though

Enums ctd.

- Solution is to use an enumeration:
  Similar to a coding scheme, but now enforced by compiler.

```
enum class ODEsolver {Euler, RK2, RK4};

ODEsolver mySolver = ODEsolver::Euler;
mySolver = 2; // Compile-time error

void useSolver(ODEsolver, double*);
useSolver( mySolver, data );
```

- `ODEsolver` is now a new type, with all C++ type-safety features attached.
- i.e. an `int` cannot be converted to an `enum`.

Outline

Vectors - `std::vectors`

- C++ has a `std::vector` (use `#include <vector>`)

```
std::vector<int> a(100);
for(size_t i=0 ; i < a.size() ; i++){
  a[i] = i;
}
std::vector<int> b(200);
for(size_t i=0 ; i < b.size() ; i++){
  b[i] = a[i/2] + i;
}
```

- A `std::vector` is always indexed from zero up to `size()-1`.
- Memory is freed automatically when vector goes out of scope.
- Vectors can be passed to functions with little cost; data is not explicitly copied if passed constant-by-reference (see later)
- However, bounds checking is not done for [] access:

```
a[100] = 0; // Will likely cause hard-to-trace error
```

- The following is available (at a slight computational cost):

```
a.at(100) = 0; // Will produce useful error at run-time.
```

C++ arrays

- For small, fixed-size arrays, C++ has an explicit `std::array` type:

```
#include <array>
std::array<int,5> a = {1,4,9,16,25};
std::cout << a[0] << a[4] << std::endl;
a[3] = 5;
```

- Arrays are always indexed from zero.
- Array-size must be known at compile-time
- Arrays cannot be resized at compile-time (unlike a `vector`)
- No bounds checking is done
- Anything requiring large arrays or dynamic resizing should be done with heap allocation or `std::vector`

Arrays ctd

- No bounds checking is done for arrays

```
std::array<int, 5> a;
a[0] = 6; // valid
a[3] = 3; // valid
a[-1] = 8; // invalid
a[5] = 9; // invalid
```

- The third and fourth lines above will lead to either a seg-fault or undefined results.
- Arrays cannot have their size defined at run-time; the '5' must be known to the compiler.

C-style arrays - a horrible warning

- C++ inherited arrays from C

```
int a[5] = {1, 4, 9, 16, 25};
```

- You will often see these used in legacy C++ code, but you are strongly advised not to use them yourself.
- There are potentially confusing issues with passing this type of array to functions, the difference between this array and pointers, and with determining the size of the array at run-time.

Passing arrays to functions

- The syntax to pass an array to a function is:

```
int sum(std::array<int,10> a){
 // Code here
}

std::array<int,10> b;
int total = sum(b);
```

- but this causes the data to be copied.
- To allow the contents of the array to be altered within the function:

```
void f(std::array<int,4>& b){
  b[0] = 5;
}

std::array<int,4> a = {1,2,3,4};
f(a);
std::cout << a[0]; // Prints 5
```

- The function parameter could be declared `const`, which would prevent this.

Two-dimensional arrays

- Two-dimensional arrays can be achieved by:

```
std::array<std::array<int, 3>, 3> a;
a[2][1] = 1;
```

- This will produce a constant-size 2D array in a contiguous block of memory.

Matrices

- As may already be obvious, C++ arrays are not easy to use for matrix operations.
- Possible libraries include:
  - BLAS - Standardised C/Fortran specification for basic linear algebra functions
  - LAPACK - Standardised C/Fortran interface for solving systems of linear equations, finding eigenvectors, etc.
  - Armadillo `arma.sourceforge.net` - C++ interface for matrix operations - uses LAPACK as a back-end
  - Eigen `eigen.tuxfamily.org` - C++ interface for matrix operations - uses LAPACK as a back-end
- I have not had much experience with any of these, but believe they are fairly stable and should give good performance in general.
- A good implementation of BLAS/LAPACK should be used for preference, preferably tuned for your system, e.g. ATLAS.