Introduction to Scientific Programming with C++

Session 1: Control structure

Martin Uhrin

UCL

February 11-13th 2013

Table of Contents

1. Conditionals

2. Loops
   - while loops
   - do-while loops
   - for loops

3. Functions
   - Declaring functions
   - Default values
   - Overloading functions
   - Recursive functions

if, else

So far our programs have been linear: one start, one end, one path in between. Let's branch out:

if, else

So far our programs have been linear: one start, one end, one path in between. Let's branch out:

```
if(speed == 88){std::cout << "Great Scott!";}
```

the statement after the `if` condition will only be executed if the condition is true.

if, else

So far our programs have been linear: one start, one end, one path in between. Let's branch out:

```
if(speed == 88){std::cout << "Great Scott!";}
```

the statement after the `if` condition will only be executed if the condition is true.You can execute more than one statment within a { } block:

```
if(power >= 1.21)
{
  std::cout << "1.21 gigawatts!";
  std::cout << "1.21 gigawatts. Great Scott!\n";
}
```

if, else

So far our programs have been linear: one start, one end, one path in between. Let's branch out:

```
if(speed == 88){std::cout << "Great Scott!";}
```

the statement after the `if` condition will only be executed if the condition is true.You can execute more than one statment within a { } block:

```
if(power >= 1.21)
{
  std::cout << "1.21 gigawatts!";
  std::cout << "1.21 gigawatts. Great Scott!\n";
}
```

We can also specify what to do if the condition is not fulfilled:

```
bool haveEnoughPower;
if(power >= 1.21){
  haveEnoughPower = true;
}else{
  haveEnoughPower = false;
}
```

if, else

Lastly we can string together a series of conditionals, like so:

```
int width, height;
std::cin >> width >> height;

if(width > height){
  std::cout << "Fat rectangle\n";
}else if(height > width){
  std::cout << "Tall rectangle\n";
}else if(width == height){
  std::cout << "Square\n";
}else{
  std::cout << "IMPOSSIBLE!!";
}
```

These are evaluated one after another until one is found to be true, otherwise the final `else` statement is executed.

if, else

Lastly we can string together a series of conditionals, like so:

```cpp
int width, height;
std::cin >> width >> height;

if(width > height){
  std::cout << "Fat rectangle\n";
}else if(height > width){
  std::cout << "Tall rectangle\n";
}else if(width == height){
  std::cout << "Square\n";
}else{
  std::cout << "IMPOSSIBLE!!";
}
```

These are evaluated one after another until one is found to be true, otherwise the final `else` statement is executed.

Do

Keep conditionals simple, break them up if you have to. A huge proportion of programming errors come from conditional statements.

Loops

The purpose of a loop is to execute a set of statements until a condition is fulfilled.

Loops

The purpose of a loop is to execute a set of statements until a condition is fulfilled.

while loop

```
while(expression) statement
```

Example:

```
  int t = 10;
  while(t != 0)
  {
    std::cout << t << ", ";
    --t;
  }
  std::cout << "Blastoff!";
```

Loops

Go on, go on, go on, go on, go on...

do-while loop

```
do statement while(expression);
```

```
#include <iostream>

int main()
{
  bool wantACupOfTea;
  do
  {
    std::cout << "Cup of tea father? ";
    std::cin >> wantACupOfTea; // enter 0 or 1
    if(!wantACupOfTea)
      std::cout << "Go on\n";
  } while(!wantACupOfTea);

  return 0;
}
```

../code/1_control_structure/lectures/tea_father.cpp

Loops

for loop

```
for(initialisation; condition; increment) statement;
```

This works as follows:

Loops

for loop

```
for(initialisation; condition; increment) statement;
```

This works as follows:

1. `initialisation` is executed. Usually used to initialise a counter. This happens once.

Loops

for loop

```
for(initialisation; condition; increment) statement;
```

This works as follows:

1. `initialisation` is executed. Usually used to initialise a counter. This happens once.
2. `condition` is checked, if true the loop continues, otherwise the loop ends and is skipped.

Loops

for loop

```
for(initialisation; condition; increment) statement;
```

This works as follows:

1. `initialisation` is executed. Usually used to initialise a counter. This happens once.
2. `condition` is checked, if true the loop continues, otherwise the loop ends and is skipped.
3. `statement` is executed.

Loops

for loop

```
for(initialisation; condition; increment) statement;
```

This works as follows:

1. `initialisation` is executed. Usually used to initialise a counter. This happens once.
2. `condition` is checked, if true the loop continues, otherwise the loop ends and is skipped.
3. `statement` is executed.
4. `increment` is executed and we go back to step 2.

Loops

for loop

```
for(initialisation; condition; increment) statement;
```

This works as follows:

1. `initialisation` is executed. Usually used to initialise a counter. This happens once.
2. `condition` is checked, if true the loop continues, otherwise the loop ends and is skipped.
3. `statement` is executed.
4. `increment` is executed and we go back to step 2.

Here's our blastoff example with a for loop:

```
for(int t = 10; t != 0; --t)
{
  std::cout << t << ", ";
}
std::cout << "Blastoff!";
```

Functions

How would we function without them?

Functions provide a way of structuring a program in a more modular way, by grouping sets of statement so they can be easily reused.

Functions

How would we function without them?

Functions provide a way of structuring a program in a more modular way, by grouping sets of statement so they can be easily reused.
Consider: what if we want to calculate gravitational force between multiple bodies?

```
forceEarthSun = G * massEarth * massSun /
  (rEarthSun * rEarthSun);
forceEarthMoon = G * massEarth * massMoon /
  (rEarthMoon * rEarthMoon);
forceEarthMars = G * massEarth * massMars /
  (rEarthMars * rEarthMars);
```

Functions

How would we function without them?

Functions provide a way of structuring a program in a more modular way, by grouping sets of statement so they can be easily reused.
Consider: what if we want to calculate gravitational force between multiple bodies?

```
forceEarthSun = G * massEarth * massSun /
  (rEarthSun * rEarthSun);
forceEarthMoon = G * massEarth * massMoon /
  (rEarthMoon * rEarthMoon);
forceEarthMars = G * massEarth * massMars /
  (rEarthMars * rEarthMars);
```

wouldn't it be better to be able to write:

```
forceEarthSun = force(massEarth,  massSun, rEarthSun);
forceEarthMoon = force(massEarth, massMoon, rEarthMoon);
forceEarthMars = force(massEarth, massMars, rEarthMars);
```

Through the magic of functions, we can!

Functions

Format of a function:

```
type name(parameter1, parameter2, ...) { statements }
```

where:

Functions

Format of a function:

```
type name(parameter1, parameter2, ...) { statements }
```

where:

- `type` is the data type that is returned to you by the function.

Functions

Format of a function:

```
type name(parameter1, parameter2, ...) { statements }
```

where:

- type is the data type that is returned to you by the function.
- name is the unique identifier (name) of the function.

Functions

Format of a function:

```
type name(parameter1, parameter2, ...) { statements }
```

where:

- `type` is the data type that is returned to you by the function.
- `name` is the unique identifier (name) of the function.
- `parameter`, each of these is just like a variable definition (e.g. `double mass`).

Functions

Format of a function:

```
type name(parameter1, parameter2, ...) { statements }
```

where:

- type is the data type that is returned to you by the function.
- name is the unique identifier (name) of the function.
- parameter, each of these is just like a variable definition (e.g. `double mass`).
- statements are the function's *body*. These statements will execute when the function is called.

Functions

Format of a function:

```
type name(parameter1, parameter2, ...) { statements }
```

where:

- `type` is the data type that is returned to you by the function.
- `name` is the unique identifier (name) of the function.
- `parameter`, each of these is just like a variable definition (e.g. `double mass`).
- `statements` are the function's *body*. These statements will execute when the function is called.

Let's try:

```
double force(const double mass1, const double mass2,
  const double r)
{
  return G * mass1 * mass2 / (r * r);
}
```

Functions

Let's try a full example:

```
#include <iostream>

double force(const double mass1, const double mass2, const double r)
{
  const double G = 3.96402e-14;
  return G * mass1 * mass2 / (r * r);
}

int main() {
  // Astronomical units
  const double massSun = 1.0, massEarth = 3.003e-6, massMars = 0.323e-6;

  const double rSunEarth = 1.0, rSunMars = 1.523;

  const double forceSunEarth = force(massSun, massEarth, rSunEarth);
  const double forceSunMars = force(massSun, massMars, rSunMars);

  std::cout << "Forces:\n";
  std::cout << "Sun-Earth:  " << forceSunEarth << "\n";
  std::cout << "Sun-Mars: " << forceSunMars << "\n";
}
```

../code/1_control_structure/lectures/grav_force.cpp

Function nomenclature

Definition

caller A point in code that *calls* a function. A value is said to be returned to the caller when the function finishes.

Functions with no type

What if we want to have a function that doesn't return anything?

Functions with no type

What if we want to have a function that doesn't return anything?

Use `void`:

```
void printForce(const std::string object1,
  const std::string object2,
  const double force)
{
  std::cout << object1 << " " << object2 << ":" << force <<
      "\n";
}
```

In C++ `void` denotes the absence of a type.

Declaring functions

I do declare

All identifiers have to be declared before they are used. So how are we to code something like:

```
const bool ON = true, OFF = false
const bool OPEN = true, CLOSED = false;

void setMicrowaveState(const bool newState) {
  if(newState == ON)
  {
    setMicrowaveDoor(CLOSED); // <- ERROR: Don't know about
    state = ON;               // setMicrowaveDoor
  }
  else
        state = OFF;
}
void setMicrowaveDoor(const bool newState) {
  if(newState == OPEN)
    setMicrowaveState(OFF);

  doorState = newState;
}
```

Declaring functions

I do declare

To get around this declare the setMicrowaveDoor function before setMicrowaveState like this:

```
void setMicrowaveDoor(const bool newState);

void setMicrowaveState(const bool newState) {
  if(newState == ON)
  {
    setMicrowaveDoor(CLOSED); // Happy: you've told me about
    state = ON;               // setMicrowaveDoor
  }
  ...
}
void setMicrowaveDoor(const bool newState) { /*as before*/ }
```

The first line tells the compiler: expect to see a setMicrowaveDoor function somewhere further down, and this is what it will look like.

Declaring functions

Definition

function prototype The header of a function without any of the body. Used in function declarations, e.g.:

```
void setMicrowaveDoor(const bool newState);
```

Default values

Definition

default value A value that is used as a function parameter if the caller doesn't supply one.

An example:

```cpp
const int MONDAY = 0;
const int TEA = 0, COFFEE = 1;
void dispenseDrink(
  const int drinkType = COFFEE)
{
  std::cout << "Dispensing: ";
  if(drinkType == COFFEE)
    std::cout << "coffee...\n";
  else
    std::cout << "tea...\n";
}
```

```cpp
int main()
{
  unsigned int dayOfWeek;
  // Enter number from 0 to 6
  std::cin >> dayOfWeek;

  if(dayOfWeek == MONDAY)
    dispenseDrink();
  else
    dispenseDrink(TEA);
}
```

../code/1_control_structure/lectures/dispense_drink.cpp

Default values

Default values have to be at the end of the parameter list, e.g.:

```
void dispenseDrink(int size,
  int drinkType = COFFEE,
  bool withMilk = false) { ... } // Good, can call:

dispenseDrink(1);      // or ...
dispenseDrink(3, TEA); // or ...
dispenseDrink(2, COFFEE, true)
```

Default values

Default values have to be at the end of the parameter list, e.g.:

```
void dispenseDrink(int size,
  int drinkType = COFFEE,
  bool withMilk = false) { ... } // Good, can call:

dispenseDrink(1);      // or ...
dispenseDrink(3, TEA); // or ...
dispenseDrink(2, COFFEE, true)
```

Bad:

```
void dispenseDrink(int drinkType = COFFEE,
  int size,
  bool withMilk = false) { ... }

dispenseDrink(/*what goes here?*/, 2); // Error!
```

Default values

Do

- Use default values to automate commonly used parameter values or provide the user with optional parameters.

Default values

Do

- Use default values to automate commonly used parameter values or provide the user with optional parameters.
- To give an indication of what a reasonable parameter value might be.

Overloading functions

What if you want to create a dot product function that works for both integers and doubles? Could write:

```cpp
int dotInt(int x0, int y0, int x1, int y1)
{ return x0 * x1 + y0 * y1; }
double dotDouble(double x0, double y0, double x1, double y1)
{ return x0 * x1 + y0 * y1; }

dotInt(fromX, fromY, toX, toY);
```

Have to look up which dot function to call based on my number type. Ideally I'd like to call `dot` and let the compiler choose the right one.

Overloading functions

What if you want to create a dot product function that works for both integers and doubles? Could write:

```
int dotInt(int x0, int y0, int x1, int y1)
{ return x0 * x1 + y0 * y1; }
double dotDouble(double x0, double y0, double x1, double y1)
{ return x0 * x1 + y0 * y1; }

dotInt(fromX, fromY, toX, toY);
```

Have to look up which dot function to call based on my number type. Ideally I'd like to call dot and let the compiler choose the right one. No problem, use:

```
int dot(int x0, int y0, int x1, int y1)
{ return x0 * x1 + y0 * y1; }
double dot(double x0, double y0, double x1, double y1)
{ return x0 * x1 + y0 * y1; }

dot(fromX, fromY, toX, toY); // Compiler will choose
                             // correct version based on
                             // from/to number types
```

Overloading functions

Definition

overloaded functions two or more functions with the same name but different parameter types and/or numbers of parameters.

Overloading functions

Definition

overloaded functions two or more functions with the same name but different parameter types and/or numbers of parameters.

Examples:

```
// Sum two or three integers
int sum(int n1, int n2);
int sum(int n1, int n2, int n3);

// Add together any integer/double
int add(int n1, int n2);
double add(int n1, double n2);
double add(double n1, int n2);
double add(double n1, double n2);
```

Math functions

C numerics library

As scientists we're going to want to manipulate numbers. Here are some commonly used functions that are available as part of the `cmath` header:

`abs`	absolute value
`sin, cos, tan`	Warning: take angle in radians!
`exp, log, log10`	raise *e* to power, natural log and base 10 log
`sqrt`	
`pow(double base, double exp)`	raise based to power exp

See [1] for a full list.

[1] `http://www.cplusplus.com/reference/clibrary/cmath/`

Math functions

C numerics library

As scientists we're going to want to manipulate numbers. Here are some commonly used functions that are available as part of the `cmath` header:

`abs`	absolute value
`sin, cos, tan`	Warning: take angle in radians!
`exp, log, log10`	raise e to power, natural log and base 10 log
`sqrt`	
`pow(double base, double exp)`	raise based to power exp

See [1] for a full list.

To use the math functions include the cmath header by writing:

```
#include <cmath>
```

at the start of you program.

[1] `http://www.cplusplus.com/reference/clibrary/cmath/`

Recursivity

Definition

recursive function a function that calls itself.

This is similar to a recurrence relation in mathematics e.g.:

$$b_n = nb_{(n-1)}, b_0 = 1$$

which gives the factorial of a number (n!).

Recursivity

Definition

recursive function a function that calls itself.

This is similar to a recurrence relation in mathematics e.g.:

$$b_n = nb_{(n-1)}, b_0 = 1$$

which gives the factorial of a number ($n!$). And in C++:

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

../code/1_control_structure/lectures/factorial.cpp