Functions

Purpose of Functions

- We have learned about variables, loops, and conditions
- In principle, this is enough to do any kind of computation efficiently
- But what is missing?
- A good way to structure programs
- The most useful feature of C++ for structuring programs is **functions**

Why should we use functions?

History

- A programming project can involve millions of lines of source code
- Programming breakthrough in the 1950s:
  Use functions to divide a complex task into smaller, simpler tasks
- C and Fortran (1960s to 80s) massively use functions
- That's why they are called procedural programming languages (procedure=function)
- In C++, functions are still fundamental

Functions (Sub-Routines)

- A subroutine, also termed procedure, function, routine, method, or subprogram, is a part of source code within a larger computer program that performs a specific task and is relatively independent of the remaining code.
- As the name subprogram suggests, a subroutine behaves in much the same way as a computer program that is used as one step in a larger program or another subprogram.
- A subroutine is often coded so that it can be started (called) several times and/or from several places during one execution of the program, including from other subroutines, and then branch back (return) to the next instruction after the call once the subroutine's task is done.

Functions (Sub-Routines)

- A subroutine may be written so that it expects to obtain one or more data values from the calling program (its input arguments). It may also return a computed value to its caller (its return value), or provide various result values or output arguments.
- Subroutines are a powerful programming tool, and the syntax of many programming languages includes support for writing and using them.
- Judicious use of subroutines will often substantially reduce the cost of developing and maintaining a large program, while increasing its quality and reliability.
- Subroutines, often collected into libraries, are an important mechanism for sharing software.

Top Down

- We decompose a large problem in smaller sub-problems
- Each sub-problem is decomposed again into smaller ones
- In the end we have many small problems to solve
- Solving a small problem is easier than solving a big one
- Often, the same small problems appears in many larger problems, suggesting that it is worth to factor out the solution of these small problems into libraries

Function Head

```
returnType FunctionName(arguments)
```

- This is the head of a function with name ***`FunctionName`***. This is an identifier.
- The ***`arguements`*** consist of a comma separated list of variables declarations
- The ***`returnType`*** is the type of the value returned by the function
- If the function has no return value, the return type is **`void`**

Function Heads, Example

```
double pow(double a, double b)
```

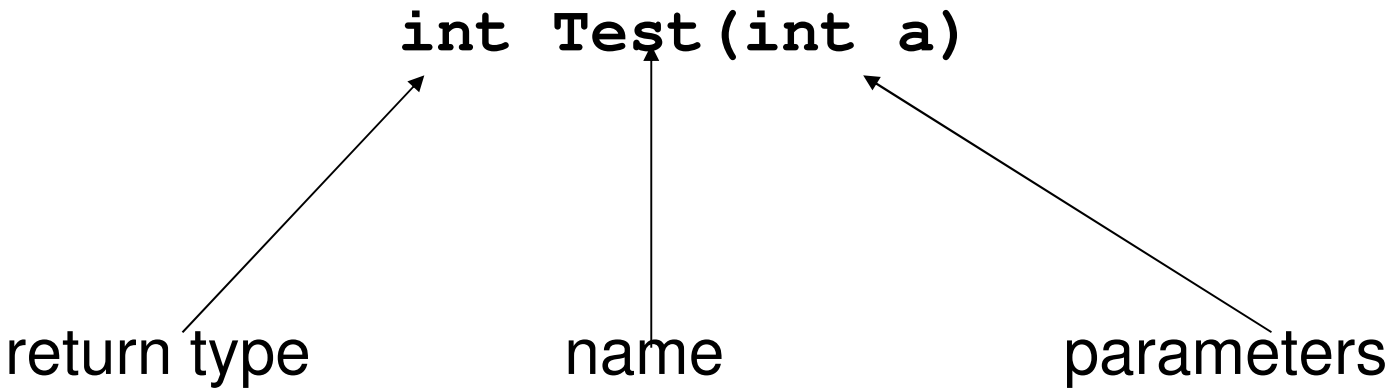
- Function head of the built-in function `pow`
- Has two input parameters of type double
- Returns a value of type double

Function Heads, Example

What is the head of a function with:

- Name: **`Test`**
- Input parameters: one variable **`a`** of type **`int`**
- Return value: of type **`int`**

Solution



Function Heads, Example

What is the head of a function with:

- Name: **`FormatDisk`**
- No input parameters
- No return value

```
void FormatDisk()
```

Function Heads, Example

What is the head of a function with:

- Name: **`BoxVolume`**
- 3 input parameter of type double
- Return type double

```
double BoxVolume(double a, double b, double c)
```

How to Call Functions

- We specify their name and the argument list enclosed in round brackets
- For instance, the function pow has head

```
double pow(double base, double exponent)
```

- and we can call it as follows:

```
double x = 3.5;

y = pow(x, 2.8);
```

- More on this topic later …

Which functions are available already?

Built-in Functions

- Some functions are already provided in C++
- These are called built-in functions
- They are part of the library C++ inherited from C, which is called C Runtime library.
- To use them in C++, we need appropriate include statements, e.g. **#include<cmath>** for math functions (sin, cos, exp, log, …)
- Note that the original include statements in C was **#include <math.h>.** In C++ math.h still works, but it is better to use cmath, which is wrapped in the std namespace and does not pollute the global namespace (more on namespaces later)
- Most CRT libraries (not only math ones) are now wrapped in std namespace equivalent libraries

C Runtime Library

https://msdn.microsoft.com/en-us/library/2aza74he.aspx

The most commonly used functions from here are:

- Data Conversion
- Floating-Point Support (i.e. math)
- String Manipulation
- Buffer Manipulation
- Character Classification
- Memory Allocation
- System Calls
- Time Management

Problem (built-in functions)

- Output the values of sin(x)*cos(x) in the range 0-2pi at intervals equally spaced of size pi/10 (use the built-in functions **sin** and **cos**)
- Justify the output to the right and print exactly 4 decimals
- Full solution in: SinCos.cpp

```
// define is a compiler directive
#define _USE_MATH_DEFINES  // cause import of some macros like M_PI

#include <iostream>
#include <iomanip>  // needed to format the output
#include <cmath>    // needed for sin, cos and M_PI

// import only the symbols needed
using namespace std;

int main()
{
    const unsigned nPoints = 10;
    double step = M_PI / nPoints;  // here nPoints is lifted to double
    for (unsigned i = 0; i < 2*nPoints; ++i) {
        double x = i*step;
        cout << std::fixed << std::setprecision(3)
             << std::setw(8) << x << ", "
             << std::setw(8) << sin(x)*cos(x)
             << "\n";
    }
    cout << "\n";
}
```

Problem (built-in functions)

- Define an array of size 10 containing the sequence of characters “hello”
- Print the elements of the array as integers
- Print the array interpreting it as a null-terminated string
- Print the size of the array in bytes (use **sizeof**)
- Compute and print to the string the size of the null-terminated string contained in the array in two alternative ways
  - using the CRT function **strlen** (requires **#include <cstring>**) and print it to the string
  - use a loop to find the position of the first null character
- Full solution in: **CharArrayLength.cpp**

```
int main()
{
    const char msg[10] = "Hello";

    // print all elements of the array, converted to integers
    cout << "Binary content: ";
    for (int i = 0; i < sizeof(msg); ++i)
        cout << static_cast<int>(msg[i]) << " ";
    cout << endl;

    // print array interpreted as a null-terminated string
    cout << "String: " << msg << endl;

    // print size of the array in bytes
    cout << "Size in bytes: " << sizeof(msg) << endl;

    // print length of array computed with the CRT function strlen
    cout << "String length: " << strlen(msg) << endl;

    // print length of array computed with a loop
    for (int i = 0; i < sizeof(msg); ++i)
        if (msg[i] == 0) {
            cout << "String length: " << i << endl;
            break;
        }
}
```

User Defined Functions

- If we need functions which are not built-in, we need to write them ourselves
- Such functions are called user defined functions
- Most functions we use will be user defined (the set of built-in functions is minimal)

C++ Functions

- C++ functions are similar to mathematical functions (name, input parameters, return value). But:
- They can have a task aside from returning a value (e.g. launch a missile)
- They may have no input parameters
- They may have no return value

User Defined C++ function

Function Definition

- To create a user defined function, we must provide the necessary C++ commands
- This is called the function definition
- Function definition consists of a function head and a function body
- Function head contains information on function name, input parameters, and return value
- Function body contains the C++ commands which fulfill the purpose of the function

Function Definition and Function Body

```
returnType FunctionName(parameters) // function head
{
    // function body
    statements
}
```

- The function definition consists of the function head followed by the C++ statements which fulfill the purpose of the function
- These statements are enclosed in curly braces and are called the function body
- A function usually is defined at global scope, i.e. outside any other function (especially outside the `main` function)

Function Definition: Example

Write down the definition of a function that adds up three integers and returns the result.

Function Definition: Example

```
int Add(int a,int b,int c)
{
    return a+b+c;
}
```

Return type	`int`
Function name	`Add`
Parameters	`a,b,c,` all of type `int`
Function head	`int Add(int a,int b,int c)`
Function body	`{ return a+b+c; }`
Task	Add 3 integers and return result

Function Definition: Example

- Write down the definition of a function that adds up the entries in absolute value of an array with entries of type double and returns the result (i.e. we compute the norm-1 of the array)

Solution

```
// note the use of const and unsigned
// it is good practice to define variables types
// as precisely as possible
double norm1(const double *v, unsigned int n)
{
  double s = 0;
  for (unsigned int i = 0; i < n; ++i)
    s += std::abs(v[i]);  // we call abs
  return s;
};
```

Return Values Must be Returned

- If a function has return type **`void`**, it does not return a value
- In all other cases, we must make sure in the function body that a value of the correct type is returned under any circumstances
- For instance, if(condition) return x; usually will be wrong if we don't make sure that a value is returned also when the condition is false

Return Statements

`return` ***`expression`***`;`

- Functions return values by such return statements and terminates
- From the function head, we know the return type of the function. The ***`expression`*** must have same type
- A return statement immediately stops the execution of the function
- **`return;`** stops the execution of the function without returning a value (only allowed if the return type is `void`)
- **`return;`** is optional at the end of function of type **void**

Question: What is wrong with the following function?

```
1 int  Search()
2 {
3     for(int i=0;i<100;i++)
4        if(rand()%55==0)
5          return i;
6 }
```

This function has two possible exit points: if a random number divisible by 55 is found is found, the program flow hits return I and terminate, otherwise the program flow reaches the end of the function and terminate. A value of type int must be returned in both cases. Correct version (for instance):

```
1 int  Search()
2 {
3     for(int i=0;i<100;i++)
4        if(rand()%55==0)
5          return i;
6     return -1;   // result -1 means that
7                  // no number was divisible by 55
8 }
```

Question: What is wrong with the following function?

```
1 double  test()
2 {
3     cout << "hello" << endl;
4 }
```

The return type is double, so there must be a return statement which returns a double value. Correct version:

```
1 void  test()
2 {
3     cout << "hello" << endl;
4 }
```

Return type `void` means the functions does not return a value

Being the function of type void, we omitted **return;** at the end of the function

Problem

- Write down the definition of a function that checks if two integers have the same parity (both even or both odd) and returns the answer as a bool value

How to use a C++ function: function call

Using a Function = Function Call

- Functions which are defined already can be used (repeatedly if necessary)
- To use a function, we call the function
- This is called a functions call
- Function definition and function call are two completely different things

Most Functions Need Input Information

Examples:

sin(x): needs x

Input information is passed to functions through the function **parameters** (also called **arguments**)

How is Input Information Passed to a Function?

As a comma separated list in parenthesis after the function name.

Examples of function calls:

- `IsPrime(1001)`
- `AreEqual(v,w)`

Attention: if we use *variables* as parameters in a function call, we must declare and initialize them first!

Function Call

A function call consists of the function name followed by a comma separated list of values for the parameters enclosed in parenthesis

Purpose:

- Passes the values of the parameters to the function
- Executes the function with this input

Function Call: Example

Function contained in `<cmath>`

`pow(a,b)`: takes two parameters of type double and returns `a` to the power `b` in type double

Possible function call: `pow(2.0,31.0)`

(function name followed by a comma separated list of values for the parameters)

What Happens in a Function Call?

The program reaches `pow(2.0,31.0)`
What happens?

- The parameter values 2.0 and 31.0 are passed to the function and copied into the function's arguments
- The function is executed with this input and computes the result 2147483648
- The function call is replaced by the result (!)
- Hence, `cout << pow(2.0,31.0);` has the same effect as `cout << 2147483648;`

Rules for Function Calls

- Treat a function call like a value (except if it is of type void)
- Functions with return type void:
  cannot treat function call as a value – functions call must be single command, not inside any expression
- The values of function parameters can be specified by any expressions of the correct type
- Unlike function definitions, function calls are done inside other functions (often inside the main function)

Function Call: Example

Task:

- Show that we can use any expression with return value of type double as parameters in a call of the function `pow`

- Show that with a function call of the function `pow`, we can do exactly the same things as with a value of type double:
  - print to the screen,
  - use in other expressions,
  - write to a file etc.

Solution

```
1  // double literals can be used as parameters:
2  cout << pow(2.0,5.0) << endl;
3  double x = 2.0;
4  double y= 5.0;
5  // variables can be used as parameters:
6  cout << pow(x,y) << endl;
7  // more complicated expressions, too:
8  cout << pow((x-y)*5,x*x*y) << endl;
9  // even a result of a function call can be a parameter:
10 cout << pow(pow(2.0,5.0),3.0) << endl;
11
12 // a function call can be the right hand side of assignment:
13 double z = pow(2.0,5);
14 // result of a function call can be printed
15 cout << pow(2.0,20) << endl;
16 // a function call can be part of an expression:
17 cout << z*100/pow(10.0,5) << endl;
```

Problem

Write and test a function

double RandVector(int n, const double *v1, const double *v2)

that returns the scalar product of two vectors.

Problem

Write and test a function

void RandVector(int n, double *result, double a, double b)

that fills the array result with random numbers in the interval [a,b].

By default, a vector of random numbers in [0,1] should be returned.

Hint: A random number x in [a,b] can be created by

double x, max = RAND_MAX;

x = (rand()/max)*(b-a)+a;

(needs #include<cstdlib>)

Function Declarations

- They are function headers, terminated by a semicolon, without body
- Parameters names are optional. Example the following two lines are equivalent:

```
int foo(int);
int foo(int x);
```

- A function declarations (or a definition) must be available before any function call, otherwise we get a compiler error

```
int foo(int);  // declare foo
// some other code . . .
y = foo(3);    // use foo
```

Function Declaration - Example

```
// definition of foo
void foo(int x)
{
   cout << x << endl;
}

int main()
{
   foo(3);  // use foo
   return 0;
}
```

This is ok because, when I use the function *foo* its **definition** is available

Function Declaration - Example

```
int main()
{
   foo(3);  // use foo
   return 0;
}

// definition of foo
void foo(int x)
{
   cout << x << endl;
}
```

Here we moved the definition below the place in the code where it is used.

This is NOT ok because, when I use the function *foo* neither its definition nor its declaration are available

We will get a compiler error: unknown identifier

Function Declaration - Example

```
void foo(int); // declare foo

int main()
{
   foo(3);  // use foo
   return 0;
}

// definition of foo
void foo(int x)
{
   cout << x << endl;
}
```

This is ok because, when I use the function *foo* its declaration is available

Function Declaration - Example

```
void foo(int); // declare foo

int main()
{
   foo(3);  // use foo
   return 0;
}
```

This is ok because, when I use the function *foo* its declaration is available

Note that *foo* is not defined in this file

Compilation of this file will succeed, but if we want linking of the final program to succeed too, we need to provide the definition of *foo* in another file or library

Move Useful Functions in Separate File

- The split between function declaration and definition allows us to define functions in separate files
- This allows to:
  - Have many small files, instead of one big files with lot of lines of code, which makes maintenance easier
  - Sharing useful functions across multiple programs

.h and .cpp

LIBRARY FILES

```
// foo.h

// functions' declarations

int foo(int);
```

```
// foo.cpp

// functions' implementations

int foo(int)
{
   // do something
}
```

PROGRAM FILES

```
// main.cpp

// import library
#include "foo.h"


int main()
{
    // use library
    int x = foo(3);
    // ...
}
```

- *foo.cpp* and *main.cpp* are compiled indepdendently into *.obj* files (*.o* files in Linux), then the linker join them to produce an executable

Default Parameters

- Some function parameters are only used rarely
- In such a case, a default value for the parameter can suffice
- Default values can be specified in the function declaration or definition
- Default values cannot be re-specified in the function declaration or definition
- Default values are only used by the function if no values are substituted for them in the function call
- There can be several default parameters, but they must be at the end of the list of parameters

Default Parameters Examples

```
1  #include<iostream>
2  #include<iomanip>
3  using namespace std;
4
5  void PrintDouble(double x, int prec=10)
6  {
7        cout << setprecision(prec) << x << endl;
8        // prints x with a precision of prec digits
9  }
10
11 int main()
12 {
13       PrintDouble(0.123456789123);
14       // prints 10 digits; default value 10 is used for prec
15
16       PrintDouble(0.123456789012,3);
17       // prints 3 digits; default value is overridden
18
19       system("pause");
20 }
```

Default Parameters Examples

```
1  #include<iostream>
2  #include<iomanip>
3  using namespace std;
4
5  void PrintDouble(double x, int prec=10);
6
7  int main()
8  {
9      PrintDouble(0.123456789123);
10
11 }
12
13 void PrintDouble(double x, int prec=10)   // error
14 {
15     cout << setprecision(prec) << x << endl;
16 }
```

Compiler error – default parameters cannot be re-specified, even with the same value

Modes for Passing Arguments

- By Values
  - A copy of the variable is created
  - Any modification of the variable inside the function have no effect on the value of the original variable
- By Reference
  - A reference to the original variable is created (i.e. an alias)
  - Any modification of the variable inside the function happens also on the original variable
- By Pointer
  - A pointer to the original variable is passed
  - Any modification to the pointed object inside the function happens also on the original variable
  - Access to the pointed variable must happen using pointer syntax

Example - Pass Argument By Value

```
int foo(int x)   // pass x by value, it is copied
{
    return ++x;  // x is modified
}
int main()
{
    int y = 5;
    cout << y << "\n";
    cout << foo(y) << "\n";
    cout << y << "\n";
}
```

- If x was a large object instead of being an int, then passing it by value would be expensive. Better to pass it by **const reference**

Example - Pass Argument By Reference

```
int foo(int& x)   // pass x by reference
{
    return ++x;  // x is modified
}
int main()
{
    int y = 5;
    cout << y << "\n";
    cout << foo(y) << "\n";
    cout << y << "\n";
}
```

Example - Pass Argument By Pointer

```
int foo(int *x)   // pass x by pointer
{
    return ++*x;  // x is modified
}
int main()
{
    int y = 5;
    cout << y << "\n";
    cout << foo(&y) << "\n";
    cout << y << "\n";
}
```

Arguments of Type struct

- The argument of a function can also have type struct
- If passed by value, the struct is copied into the argument, which might be costly for large structures
- Generally we pass struct arguments by:
  - **reference**, if we intend to modify them inside the function
  - **const reference** otherwise

Example - struct arguments

- Create a function which multiplies two complex numbers

```
struct ComplexNumber
{
    double r, i;
};

// we pass the complex numbers by const reference, so we avoid the cost of copying them
// we specify them as const, to make it clear that they will not be modified in the function
ComplexNumber mult(const ComplexNumber& a, const ComplexNumber& b)
{
    ComplexNumber c;
    c.r = a.r * b.r - a.i * b.i;  // compute real part
    c.i = a.r * b.i + a.i * b.r;  // compute imaginary part
    return c;
}

int main()
{
   ComplexNumber x = {1, 2},
                 y = {2, -3},
                 z = mult(x, y);
   std::cout << z.r << "+" << z.i << "i\n";
}
```

Problem

- Write and test a function which computes the real roots of a second order polynomial ax^2+bx+c, using as argument type the struct *PolyCoeff* and as return type the struct *Roots*, which we define as:

```
struct Roots { double r1, r2; };
struct PolyCoeff { double a, b, c; };
```

- Ignore for simplicity the possibility that the roots may be complex
- Hint: the formula is: $x = (-b \pm \sqrt{b^2 - 4ac}) / (2a)$

Scope and Visibility

- The scope of function arguments is the function itself, i.e. function arguments are **local** variables
- Variables declared in the calling function are in scope (i.e. they still exist) during the execution of the function, but they are not accessible, unless they were passed via a pointer or via reference to the function
- Variables defined outside of functions are global.

Scope and Visibility Example

```
// this is a global variable
int x = 20;

void foo(int& z)
{
    // here we can access x, which is in global
    // we can access y, which has been passed via z
    // k cannot be accessed, however it does exist
    cout << ++x << " " << ++z << endl;
}

int main()
{
    // This are all local variables.
    // The variable x masks the global variable x
    int x = 5, y = 1, k = 3;
    cout << x << " " << y << " " << k << endl;
    foo(y);
    cout << x << " " << y << " " << k << endl;
}
```

Do not Re-Declare Function Parameters

```
int Multiply(int x, double y)
{
    int x; // wrong!
    return x*y;
}
```

- Function parameters are declared in function head
- Must not be re-declared in function body!
- int x; in functions body “shadows” the parameter x

Good Practice

- Do not use global variables
- They cause serious problems in multi-threading applications (parallelization)
- The only safe use is with constants with simple types (e.g. int, double)

```
const double pi = 3.14;  // this is ok
```

Return Value as an Argument

- Issue: When we return an object, it is copied, which may be expensive
- Solution: we create the variable outside the function and pass it to it as a reference, thus avoiding the copy.
- The syntax however is less convenient: we can no longer write `z=mult(x,y)`
- Sometimes the compiler will do this for you under the hood, as an optimization (see https://en.wikipedia.org/wiki/Return_value_optimization)!

```
// we pass the complex numbers by reference, so we avoid the cost of copying them
// we specify them as const, to make it clear that they will not be modified in the function
void mult(const ComplexNumber& a, const ComplexNumber& b, ComplexNumber& c)
{
    c.r = a.r * b.r - a.i * b.i;  // compute real part
    c.i = a.r * b.i + a.i * b.r;  // compute imaginary part
}

int main()
{
   ComplexNumber x = {1, 2},
                 y = {2, -3},
                 z;
   mult(x, y, z);
   std::cout << z.r << "+" << z.i << "i\n";
}
```

Function Overloading

- Functions with the same name can be used if they have different kinds of parameters
- “Different” means different in the type or in the number of parameters
- When the function is called, the compiler determines through the parameter values which function has to be used
- If type conversion are involved, ambiguities have to be avoided

Example Functions can be Overloaded

```
void foo(int x)
{
   cout << "foo called with an int: " << x << endl;
}

void foo(double x)
{
   cout << "foo called with a double: " << x << endl;
}

void foo(int x, int y)
{
   cout << "foo called with 2 int: " << x << " " << y << endl;
}

int main()
{
   int xi = 2;
   double xd = 2;
   foo(xi);
   foo(xd);
   foo(xi,xi);
   return 0;
}
```

Example Functions can be Overloaded

- What's wrong?

```
void foo(int x)
{
   cout << "foo called with an int: " << x << endl;
}

void foo(int x, int y = 3)
{
   cout << "foo called with 2 int: " << x << " " << y << endl;
}

int main()
{
   int xi = 2;
   foo(xi);    // compile error: which version of foo should be called?
   return 0;
}
```

- There is ambiguity: it is not clear if we want to call the implementation of *foo* which takes one single int argument, or the one which takes two arguments with a default value for the second one

Example Functions can be Overloaded

- What's wrong?

```
void foo(int x, double y)
{
    cout << "foo called with an int and double: " << x << " " << y << endl;
}

void foo(double x, int y)
{
    cout << "foo called with an double and int: " << x << " " << y << endl;
}

int main()
{
    unsigned xi = 2, yi = 3;
    foo(xi, yi);    // compile error: which version of foo should be called?
    return 0;
}
```

- There is ambiguity: an int can be automatically lifed up to double, but here it is not clear which implementation of *foo* we want to call: should the first or the second argument be lifted up to double?

Operators are Functions

- They can be **overloaded** for our custom types

```
struct ComplexNumber
{
    double r, i;
};

// we pass the complex numbers by reference, so we avoid the cost of copying them
// we specify them as const, to make it clear that they will not be modified in the function
ComplexNumber operator *(const ComplexNumber& a, const ComplexNumber& b)
{
    ComplexNumber c;
    c.r = a.r * b.r - a.i * b.i;  // compute real part
    c.i = a.r * b.i + a.i * b.r;  // compute imaginary part
    return c;
}

int main()
{
    ComplexNumber x = { 1, 2 },
                  y = { 2, -3 },
                  z = x * y;
    std::cout << z.r << "+" << z.i << "i\n";
}
```

Exercise

- Let's refactor **InputNumberArray.cpp** using function
- We define 4 functions
  - addNewNumber
  - printArray
  - growVector
  - getInteger
- Now the body of the **main** function is much easier to read.
- The functions defined are small pieces of code which simple functionality and are easier to debug.
- Full solution: **InputNumberArrayFun.cpp**

Move Reusable Function to Headers

- The *getInteger* function we just wrote, might come handy also for other projects
- It may be worth reusing it
- We move the GetInteger function into a separate cpp file: **GetInteger.cpp**
- In **InputNumberArrayFun2.cpp**, we just declare the head of the function
- Full solution:
  - InputNumberArrayFun2.cpp
  - GetInteger.cpp
- It may be convenient to move the head of the GetInteger function into a separate header file, which is then imported by the cpp file (see GetInteger.h)

Using the Debugger

- Live demo specific to Visual Studio …
  - Step Into
  - Step Over
  - Step Out
  - Go Until (Ctrl-F10)
  - Breakpoints
  - Conditional breakpoints
  - Inspect variables values
    - Mouse hovering
    - QuickWatch
    - Watches

Recursion

- When a function directly or indirectly calls itself
- Recursive functions are defined in terms of two parts: recursion and exit condition
- Recursion is equivalent to iteration (anything can be done either in recursive or iterative form)
- Some languages favor one of the two programming styles
- Certain algorithms are more naturally described in one way rather than the other

Recursion

- Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself.
- For example, we can define the operation "find your way home" as:
  - 1. If you are at home, stop moving.
  - 2. Take one step toward home.
  - 3. "find your way home".
- Here the solution to finding your way home is two steps (three steps). First, we don't go home if we are already home. Secondly, we do a very simple action that makes our situation simpler to solve. Finally, we redo the entire algorithm (recursion).

n Factorial

Recursive

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

$$n! = \begin{cases} 1 & \text{if } n < 2 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Iterative

```
unsigned factorial(unsigned n)
{
    unsigned f = 1;
    while(n > 1)
        f *= n--;
    return f;
}
```

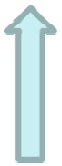
$$n! = \begin{cases} 1 & \text{if } n < 2 \\ \prod_{i=2}^{n} i & \text{otherwise} \end{cases}$$

- Recursive and iterative implementations are both fairly simple
- Example in: *factorial.cpp*

Factorial Execution

Recursive

```
factorial( 3 )
  return 3 * factorial( 2 )
      return 2 * factorial( 1 )
         return 1
```



This is the **call stack**:
let's look at it with the debugger

Iterative

```
factorial( 3 )
    f = 1
    f = f*n = 3
    n = n-1 = 2
    f = f*n = 6
    n = n-1 = 1
    return f
```

Recursion

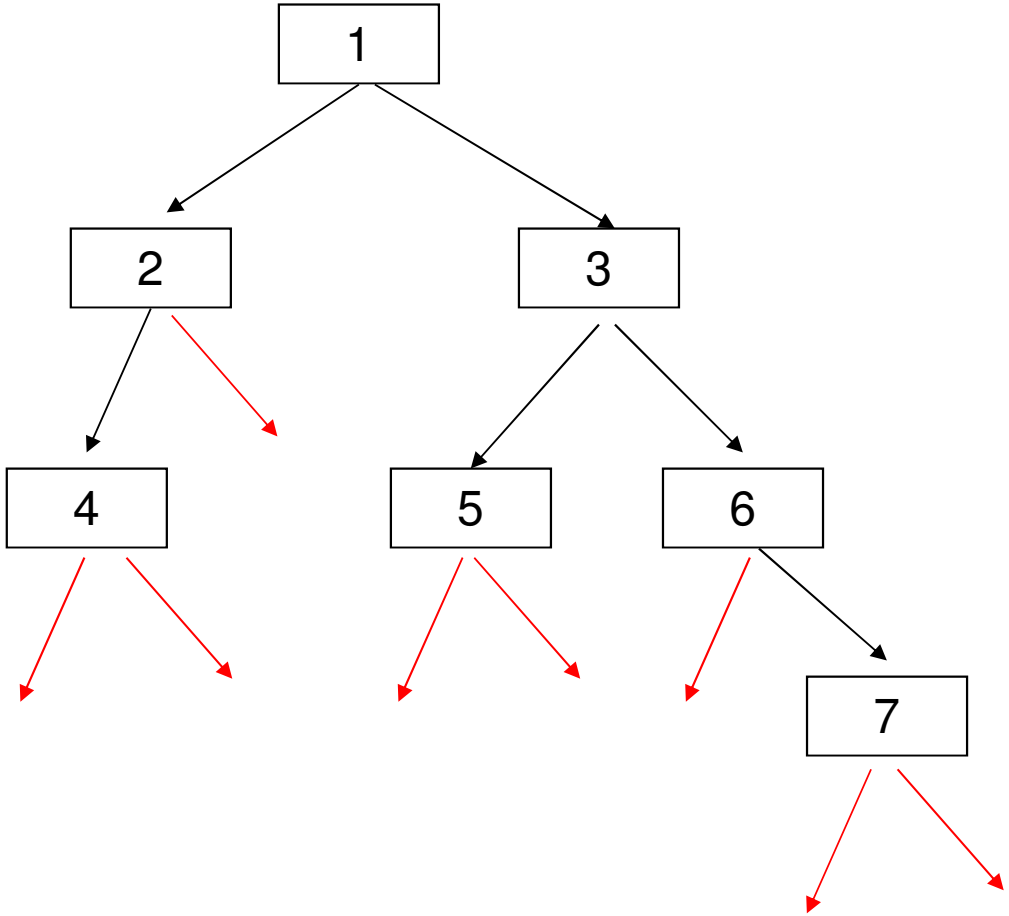
- In a recursive algorithm we have three components:
  - The condition to stop the recursion
  - The small action
  - The solution of a smaller problem structurally identical to the original problem (the recursion)

```
unsigned factorial(unsigned n)
{
     if (n == 0)  {  // condition to stop the recursion
         return 1;   // we are not invoking factorial again
     }
     else  {         // recursion
         // compute the solution of a smaller problem
         unsigned smallerProblemSolution = factorial(n-1);  // recursion
          // small action: multiply the solution of the smaller problem by n;
         return n * smallerProblemSolution;
     }
}
```

Problem:
Visit a Binary Tree in Pre-Order

- Define a binary tree structure where nodes contains an integer number
- Every node contains one value, and possible connections to a left node and a right node (a pointer)
- A null pointer means there is no connection
- Consider an algorithm which prints the content of a binary tree of arbitrary depth in pre-order (always visiting the left node first)

Problem: Visit a Binary Tree in Pre-Order



We want the tree to be visited in order: 1,2,4,3,5,6,7

Problem: Visit a Binary Tree in Pre-Order

- First we need to define a suitable data structure

```
struct Node
{
   int value;   // the value of the node
   Node *left;  // the node to the left
   Node *right; // the node to the right
};
```

- And an access point to the data structure. We can use the top node, because from it we can navigate to any other node. We can use a pointer variable of type.

```
Node *treeTop = NULL;  // a pointer to a node
```

Problem: Visit a Binary Tree in Pre-Order

- Then, let's construct manually a tree like the one in the picture

```
Node *buildTree()
{
    Node *n1, *n2, *n3;

    n1 = createNode(4, NULL, NULL);
    n1 = createNode(2, n1, NULL);
    n2 = createNode(7, NULL, NULL);
    n2 = createNode(6, NULL, n2);
    n3 = createNode(5, NULL, NULL);
    n3 = createNode(3, n3, n2);
    n1 = createNode(1, n1, n3);
    return n1;
}
```

Try and follow this step by step...

Problem: Visit a Binary Tree in Pre-Order

- We constructed the tree manually. We could do this because we knew the exact shape of the tree at the time we were writing the program (i.e. at ***compile time***).
- Now we get to the core part of this exercise. We want to write a function which takes as an argument the pointer to the top of the tree and visit its nodes in pre-order
- This function should be generic, i.e. it does not know in advance the exact shape of the tree it will receive

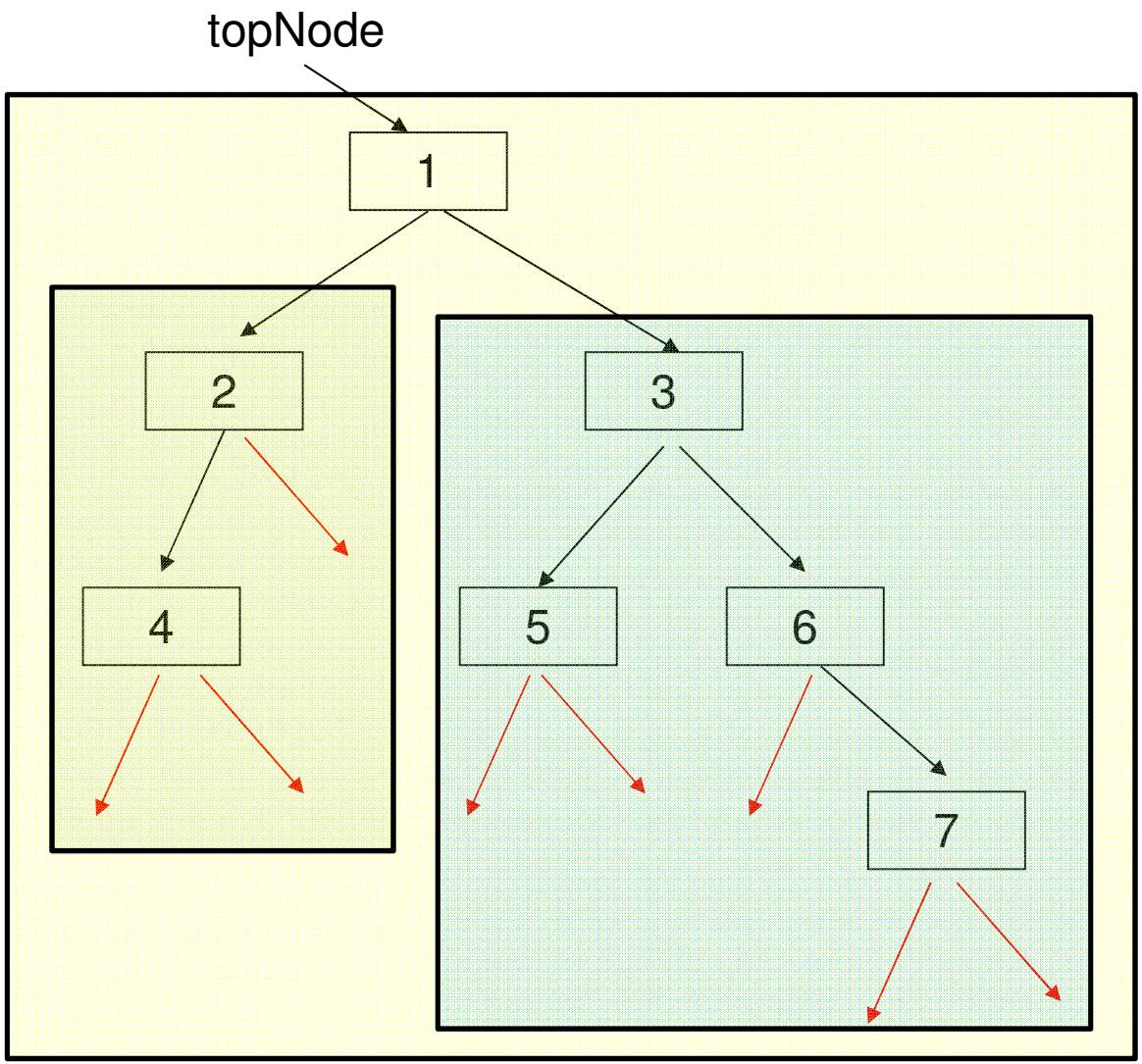
Problem: Visit a Binary Tree in Pre-Order

- Our function has the signature

```
void printTreeInPreorder(Node *node);
```

- Writing such a function using iterative programming is possible but not trivial. We need to emulate a call stack. Try if you do not believe me.
- In recursive programming there is a nice and elegant solution

Problem: Visit a Binary Tree in Pre-Order



- Visiting the yellow tree starting from node 1, is the same problem as visiting the green subtree starting from node 2 or visiting the blue tree starting from node 3
- Because it is the same problem, we can use the same function to solve them

Problem: Visit a Binary Tree in Pre-Order

- This leads to the solution:

```
void printTreeInPreorder(Node *node)
{
    if (node != NULL) {
        cout << node->value << endl;  // print value
        printTreeInPreOrder(node->left);  // visit left
        printTreeInPreOrder(node->right); // visit right
    }
}
```

- full implementation in *BinTree1.cpp*

Function Pointers

- Pointers can point also to function
- The syntax is ugly:

```
int foo(int);      // foo is a function name
int (*foo)(int);   // foo is a function pointer
```

Example: Function Pointers

```
void foo(int x)
{
   cout << x << endl;
}

void bar(int x)
{
   cout << x+2 << endl;
}

int main()
{
  // p is a pointer variable and points to
  // the function foo
  void (*p)(int) = &foo;
  (*p)(3);  // equivalent to a call to foo(x)
  p = &bar; // now p points to bar
  (*p)(3);  // equivalent to a call to bar(x)
   return 0;
}
```

Callback

- Because a function pointer is like any other variable, it can be passed as an argument to a function
- This technique is called **callback** (an advanced construct of the C language)
- Definition: a callback is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time

Example: Callback

```
int addTwo(int x)
{
    return x + 2;
}

int subTwo(int x)
{
    return x - 2;
}

void printf_fun_x(int x, int(*f)(int))  // f is a function pointer
{
    cout << (*f)(x) << endl;            // invoke the function stored in f
}

int main()
{
    printf_fun_x(10, &addTwo);      // pass a pointer to addTwo as argument
    printf_fun_x(10, &subTwo);      // pass a pointer to subTwo as argument
    return 0;
}
```

Example: Callback Applied To Binary Tree Pre-Order Visit

- We wrote an algorithm to visit a binary tree in pre-order and print the value of each node
- Now we generalize that function to take as an argument a function pointer, which will perform an action (callback)
- To make it more flexible, we want that the callback function we perform on the node to be parametric, i.e. to accept some extra arguments in addition to the node itself
- For example, if we have a function which increment the value of the node, we want to have an extra argument to specify by ‘how much’
- We achieve this by adding to the callback signature a pointer argument of type void *, which can be used to pass a pointer to any arguments to the callback. This is because any pointer can be converted to a void *.
- The calback function will have to know what to do with this void* pointer
- Full code in *BinTree2.cpp*

Example: Function Pointers Applied To Binary Tree Pre-Order Visit

```
void visitTreeInPreorder(Node *node, void (*action)(Node*, void *), void *args )
{
     if (node != NULL) {
      // perform action on the node
      // note that we are passing extra arguments for perusal of the callback
      (*action)(node->value, args);

      visitTreeInPreOrder(node->left, action, args);  // visit left
      visitTreeInPreOrder(node->right, action, args); // visit right
   }
}
```

- Before the action *print the node* was hard-coded inside the function to visit the tree
- Now we can re-use the same function to visit the tree for different actions
- We do not need to know how the callback function will use *args*

Example: Function Pointers Applied To Binary Tree Pre-Order Visit

```
// here the void * argument is not used, so we do not even bother giving it a name
void print(Node *node, void *)
{
   cout << node->value << "";
}

// here we assume that arg points to a memory location containing an int
// this is dangerous: if this is not the case, the behavior of the program is undefined
void increment(Node *node, void *arg)
{
    int *p = reinterpret_cast<int *>(arg);  // we convert arg type from void* to int*
    node->value += *p;  // we increment by the amount pointed by p
}

int main()
{
   // . . .
   visitTreeInPreorder(treeTop, &print, NULL);   // print nodes (some dummy pointer for arg)
   int x = 3;
   visitTreeInPreorder(treeTop, &increment, &x); // increment nodes (a pointer to x as argument)
   visitTreeInPreorder(treeTop, &print, NULL);   // print nodes (some dummy pointer for arg)
}
```

Additional Practice Problems

Problem

- Write and test a function with function head below, which returns true if the argument is odd, false otherwise.

```
bool isOdd(unsigned n)
```

Problem

Use the C++ built-in mathematical functions to find out which of the following identities are correct and which are incorrect. "Find out" means finding out *by experiment*; no mathematical proof is sought!

$$2\,(\cos(x))^2 - 1 = \cos(2\,x)$$

$$(\sin(x))^4 + 2\,(\cos(x))^2 - 2\,(\sin(x))^2 - \cos(2\,x) = (\cos(x))^4$$

$$4\,(\cos(x))^3 + 3\,\cos(x) = \cos(3\,x)$$

$$\pi = \sum_{k=0}^{\infty} \frac{2(-1)^k 3^{-k+1/2}}{2k+1}.$$

Hints

- note that floating point operations are rounded. Therefore you cannot just check for equality as it is almost always false. You need to check for the result of the LHS and RHS to be *close enough* in relative or absolute terms, e.g. *abs(lhs-rhs)<eps*
- infinity can be approximated experimentally with same very large number

Problem

- Write and test a function with function head below, that prints the *n* entries of the array *v* separated by spaces

```
void printArray(const int *v, unsigned n)
```

Problem

- Write and test a function with function head below, that returns *true* if the elements of the array are in increasing order, *false* otherwise

```
bool isOrdered(const int *v, unsigned n)
```

Problem

- Write and test a function with function head below, which returns the n-th prime number. For instance nthPrime(4) should return 7, which is the 4-th prime number (2,3,5,7,...)

```
unsigned nthPrime(unsigned n)
```

Problem (Euclidean Algorithm)

- Informally, the Euclidean algorithm for finding the gcd of two positive integers m,n can be described as follows.

  If m=n then return m.
  If m<n use gcd(m,n)=gcd(m,n mod m),
  if m>n use gcd(m,n)=gcd(m mod n,n)
  to recursively compute the gcd with the convention gcd(0,a)=gcd(a,0)=a for all positive integers a.

- Implement this algorithm as a recursive C++ function.