21. Function Prototypes

Objectives

- Write function prototypes
- Write header files and build a multifile project

Function Prototypes

Exercise. What's wrong with the program. You have 2 seconds.

```
#include <iostream>


int main()
{
    f(1);
    return 0;
}


void f(int x)
{
    std::cout << x << std::endl;
}
```

Correct it. Verify with C++.

Now try this:

```
#include <iostream>


void f(int);


int main()
{
    f(1);
    return 0;
}


void f(int x)
{
    std::cout << x << std::endl;
    return;
}
```

Function prototype for `f()`

A **function prototype** basically tells C++ that the function does exist. The information contained in a function prototype includes:

- **name** of the function
- the **return type** and
- the **types of the parameters**.

The function prototype is the minimal amount of information needed to get your C++ compiler to work with your C++ file.

Why?

Well C++ needs to compile `main()` to work which means from the side of `main()`, the `main()` function needs to know what to push onto the stack to communication with `f()` and also what value (if any) is expected from the return of calling `f()`. As far as `main()` is concerned, all `main()` needs is the “communication infrastructure” of `f()`.`main()` doesn't need to know how `f()` carries out its duty. We'll come back to this again later.

The format of a function prototype is exactly the same as the header of a function except for the following:

- a function prototype must end with a semicolon
- the parameter names can be left out. (You can retain them if you wish).

Here's another example. First the program without prototype:

```
int sum(int start, int end, int step)
{
    int sum = 0;
    if (step > 0)
    {
        for (int i = start; i <= end; i += step)
        {
            sum += i;
        }
    }
    else if (step < 0)
    {
        for (int i = start; i >= end; i += step)
        {
            sum += i;
        }
    }
    return sum;
}

int main()
{
    std::cout << sum(1, 10, 1) << std::endl;
    std::cout << sum(2, 10, 4) << std::endl;
    return 0;
}
```

And now the program with prototype:

```
int sum(int, int, int);


int main()
{
    std::cout << sum(1, 10, 1) << std::endl;
    return 0;
}
```

```
int sum(int start, int end, int step)
{
    int sum = 0;
    if (step > 0)
    {
        for (int i = start; i <= end; i += step)
        {
            sum += i;
        }
    }
    else if (step < 0)
    {
        for (int i = start; i >= end; i += step)
        {
            sum += i;
        }
    }
    return sum;
}
```

Exercise. Write a function prototype for the given function. Run the program to make sure it works.

```
// put prototype of isNiceNumber() here

int main()
{
    std::cout << isNiceNumber(1) << std::endl;
    std::cout << isNiceNumber(42) << std::end;
}


bool isNiceNumber(int x)
{
    return (x == 42);
}
```

Exercise. Write a function prototype for the given function. Run the program to make sure it works.

```
// put prototype for predictGoogleStockPrice here

int main()
{
    std::cout << predictGoogleStockPrice(1, 1, 2008)
              << '\n'
              << "don't count on it ..."
              << std::endl;
    return 0;
}

int predictGoogleStockPrice(int mth, int day, int yr)
{
    if (yr >= 2004)
    {
        return 450 * (yr - 2004)
               + (mth % 2) * (day - 2);
    }
    else
    {
        return 0;
    }
}
```

Exercise. Write a function that accepts an array x of integers and an integer i, computes and returns the sum of values in an array from index position 0 to index position i.

```
// put prototype here

int main()
{
    int x[10] = {0, 2, 4, 6, 8, 1, 3, 5, 7, 9};
    for (int i = 0; i < 10; i++)
    {
        std::cout << sum(x, i);
    }

    return 0;
}

// put function definition (actual function) here
```

Now you might ask: “Why bother? Why not just put the function `f()` before `main()`?”

There are two reasons.

First read this program and figure out what the program is trying to do:

```
#include <iostream>


void f(int x)
{
    std::cout << "in f() ..." << std::endl;
    if (x > 0) g(x - 1);
    return;
}


void g(int x)
{
    std::cout << "in g() ..." << std::endl;
    if (x > 0) f(x - 1);
    return;
}

int main()
{
    f(5);
    return 0;
}
```

Next tell me why it won't work. You have 5 seconds ...

And finally, tell me why moving functions around won't help!!! The point is that `f()` and `g()` calls each other!!!

Now try this ...

```
#include <iostream>


void g(int);


void f(int x)
{
    std::cout << "in f() ... x = " << x << std::endl;
    if (x > 0) g(x - 1);
    return;
}


void g(int x)
{
    std::cout << "in g() ... x = " << x << std::endl;
    if (x > 0) f(x - 1);
    return;
}


int main()
{
    f(5);
    return 0;
}
```

Do you see now that for this scenario, you **must** have a function prototype to make this program work?

For the above situation, we say that `f()` and `g()` are **mutually recursive**. A function say `h()` can also call itself; we say that `h()` is **recursive**. I don't want to spend any more time on recursion since there will be a set of notes for that later. The point here is to show you that there are cases where you have to untangle function dependencies using function prototypes.

Before I go on to the second reason, you should know that the standard “layout” of a C/C++ program looks like this:

```
... documentation (comments) ...

... #includes ...

... global constants ...

... function prototypes ...

int main()
{
    ...
}

... function definitions ...
```

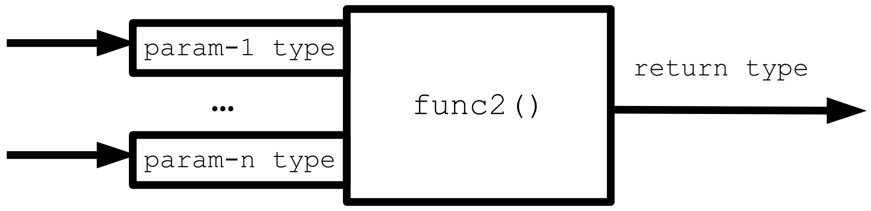
(There are other parts but we haven't talked about them yet.)

So our above program can be professionally (ahem ...) written like this:

```
// This program demonstrates the use of prototypes
// with two mutually recursive functions.

#include <iostream>


void g(int);
void f(int);


int main()
{
    f(5);
    return 0;
}


void f(int x)
{
    std::cout << "in f() ... x = " << x << std::endl;
    if (x > 0) g(x - 1);
    return;
}


void g(int x)
{
    std::cout << "in g() ... x = " << x << std::endl;
    if (x > 0) f(x - 1);
    return;
}
```

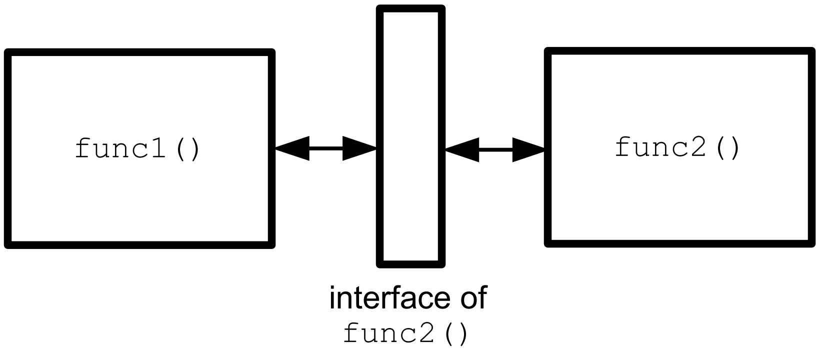
(Actually moving the function bodies below `main()` is not that crucial).

Once again in C/C++, a function, say `func1()`, can call another, say `func2()`, if `func1()` knows the "communication structure" of `func2()`. `func1()` does not need the definition or body of `func2()`. And the communication structure is made up of the types of the parameters of `func2()` and the return type of `func2()`. That's why I drew this picture for you earlier showing the types of data passed in and the type of value returned.



The function prototype of `func2()` is sometimes called the **interface** of `func2()`. Why? Because that is how `func2()`

interfaces with the outside world.



Exercise. Rewrite the following program using function prototypes and moving the function bodies below `main()` and making all constants global to remove duplicate constants declarations.

```
#include <iostream>


void printRules()
{
    const int MIN_HEADS = 2;
    const int MAX_HEADS = 10;
    std::cout << "To join MENSA you need at least "
              << MIN_HEADS << " and at most "
              << MAX_HEADS << std::endl;
    return;
}


int getHeads()
{
    int heads;
    std::cout << "How many heads to you have? ";
    std::cin >> heads;
    return heads;
}


bool passMinTest(int heads)
{
    const int MIN_HEADS = 2;
    if (heads < MIN_HEADS)
    {
        std::cout << "Too few! Try again!"
                  << std::endl;
        return false;
    }
    else
        return true;
}
```

```
bool passMaxTest(int heads)
{
    const int MAX_HEADS = 10;
    if (heads > MAX_HEADS)
    {
        std::cout << "Don't show off! Try again!"
                  << std::endl;
        return false;
    }
    else
        return true;
}


bool passTest(int x)
{
    return passMinTest(x) && passMaxTest(x);
}


int main()
{
    int heads = 0;

    printRules();
    heads = getHeads();
    while (!passTest(heads))
    {
        heads = getHeads();
    }
    std::cout << "OK. You can join MENSA."
              << std::endl;
    return 0;
}
```

Now for the second reason ...

Header Files and Multi-file Compilation

Remember the first week of class when I told you not to worry about

```
#include <iostream>
```

and I told you we will come to it? Now's the time. In this section I'll explain the purpose of this #include business. I'll also explain, at a very high level, the compilation and linking process that your compiler performs on your program in order to produce a machine executable code. Your C++ files are human readable; the machine (your PC/laptop/cellphone/etc) can only understand machine code.

And there's no better way to understand this #include business than to do an example.

First here's a program:

```
// Name: testmax.cpp

#include <iostream>

int max(int, int);


int main()
{
    std::cout << max(3, 5) << std::endl;
    return 0;
}


int max(int x, int y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Make sure it works.

Now create a new cpp file called `mymath.cpp` that contains the `max()` function. (Follow the instructions given in class – the process depends very much on the software you use to write programs).

```
// Name: mymath.cpp
int max(int x, int y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Now remove the `max()` function in `testmax.cpp` – do not remove the function prototype. Altogether, now you have two cpp files:

```
// Name: testmax.cpp

#include <iostream>

int max(int, int);


int main()
{
    std::cout << max(3, 5) << std::endl;
    return 0;
}
```

and

```
// Name: mymath.cpp

int max(int, int);

int max(int x, int y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Run your program and you'll find that it works.

When I say “compile and run your program”, the compiler software you use actually performs (at least) two things to build a machine executable code. Each cpp file (and you have two in this case) produces an **object code**. In our example `testmax.cpp` and `mymath.cpp` produce `testmax.obj` and `mymath.obj`. This step is called **compilation**.

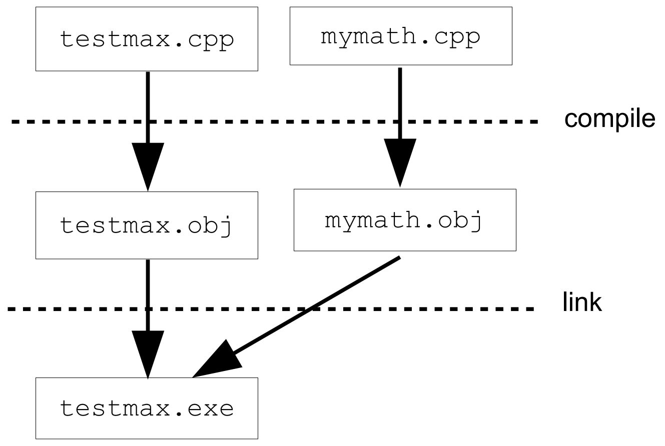
The next step involves combining both object code into a single machine executable code, `testmax.exe`.

You see the function prototype in `testmax.cpp`:

```
// Name: testmax.cpp

#include <iostream>

int max(int, int);

int main()
...
```

tells `main()` how to communicate with `max()` - what types and values should be sent to `max()` and what type and value (if any) is to be received. This is all built into `testmax.obj`. However `testmax.obj` does not contain the logic to actually execute `max()` since the logic is in mymath.obj.

The next thing the compiler software does is called **object code linking**. It takes testmax.obj and mymath.obj and produces testmax.exe. Here's a picture to help you:



When the computer runs the program, the file that is executed is actually testmax.exe.

Now I want to explain another step in the whole compilation process. And this will explain the #include business.

Now create a **header file** mymath.h (follow the instructions given in class – the process depends very much on the software you use to write programs). It includes some boilerplate code and the function prototype from testmax.cpp:

```
// Name: mymath.h
#ifndef MYMATH_H
#define MYMATH_H

int max(int, int);

#endif

```

(remember to add a blank line at the end – it's important!) and your cpp file containing main() should now look like this:

```
// Name: testmax.cpp
#include <iostream>
#include "mymath.h"

int main()
{
    std::cout << max(3, 5) << std::endl;
    return 0;
}
```

Do the same for `mymath.cpp`:

```
// Name: mymath.cpp

#include "mymath.h"

int max(int x, int y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Altogether you have now three files:

```
// Name: mymath.h
#ifndef MYMATH_H
#define MYMATH_H

int max(int, int);

#endif
```

```
// Name: testmax.cpp
#include <iostream>
#include "mymath.h"

int main()
{
    std::cout << max(3, 5) << std::endl;
    return 0;
}
```

```
// Name: mymath.cpp
#include "mymath.h"

int max(int x, int y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Run your program and make sure it works.

What's the point of `#include "mymath.h"`?

When you compile your program, the compiler software copies all the contents of the file `mymath.h` (excluding the boilerplate code) to the places where you have `#include "mymath.h"`. I'm not going to explain the boilerplate code in bold:

```
// Name: mymath.h
#ifndef MYMATH_H
#define MYMATH_H

int max(int, int);

#endif
```

All you need to know is that if you header file is named `xyz.h`, then the boilerplate code should look like this:

```
// Name: xyz.h
#ifndef XYZ_H
#define XYZ_H

...

#endif
```

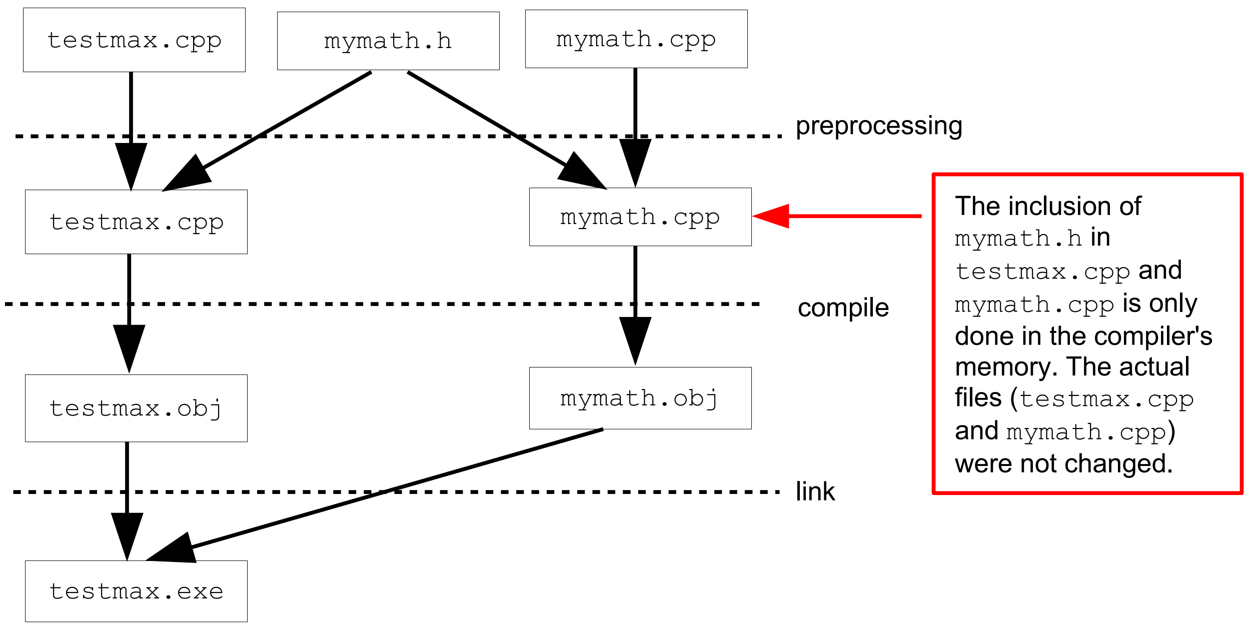
This step is called **preprocessing**. For our example, after preprocessing the `testmax.cpp` file

```
// Name: testmax.cpp
#include <iostream>
#include "mymath.h"

int main()
{
    std::cout << max(3, 5) << std::endl;
    return 0;
}
```

it becomes the following in the memory of the compiler software (the actual testmax.cpp file is ***not*** changed).

```
// testmax.cpp
#include <iostream>

int max(int, int);

int main()
{
    std::cout << max(3, 5) << std::endl;
    return 0;
}
```

Anyway ... the compilation process is now actually made up of three steps:


testmax.cpp
mymath.h
mymath.cpp
preprocessing
testmax.cpp
mymath.cpp
compile
testmax.obj
mymath.obj
link
testmax.exe
The inclusion of mymath.h in testmax.cpp and mymath.cpp is only done in the compiler's memory. The actual files (testmax.cpp and mymath.cpp) were not changed.

To learn more about issues like machine code (in the file `testmax.exe`) you need to take CISS360 (Assembly Language and Computer Systems.)

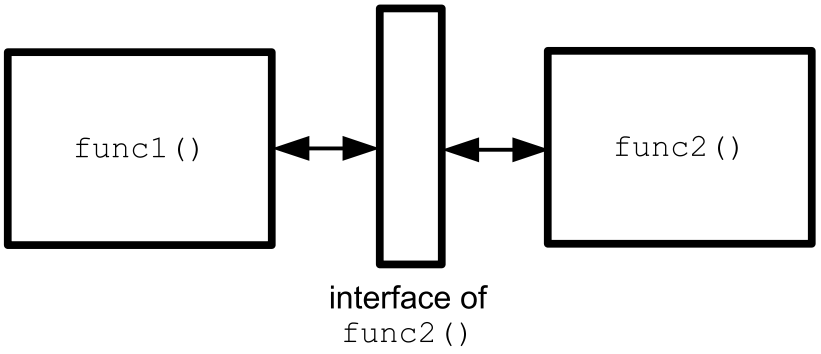
Exercise. Add a `min()` function to `mymath.cpp`, add a function prototype for `min()` in `mymath.h`. Finally modify your `main()` as follows:

```
// testmax.cpp
#include <iostream>
#include "mymath.h"

int main()
{
    std::cout << max(3, 5) << std::endl;
    std::cout << min(3, 5) << std::endl;
    return 0;
}
```

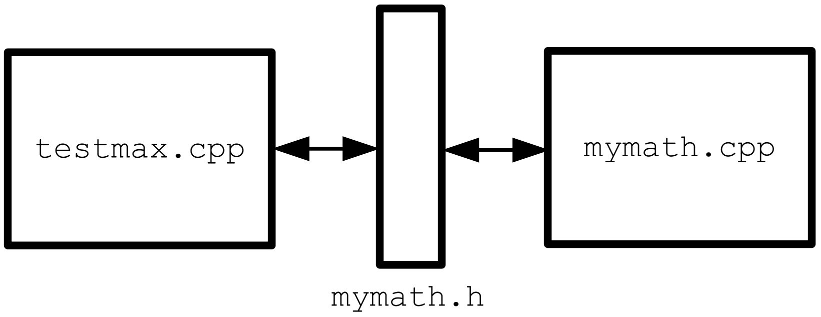
Run the program.

So what have we done?

In our previous example I told you that a function prototype for function `func2()` is like its interface with the outside world. If `func1()` calls `func2()` we have this picture:



For our current example, we placed our functions into a separate cpp file called `mymath.cpp` and put all the function prototypes of functions in `mymath.cpp` into `mymath.h`. We can then call the functions in `mymath.cpp` from our main source file (say it's called testmax.cpp)

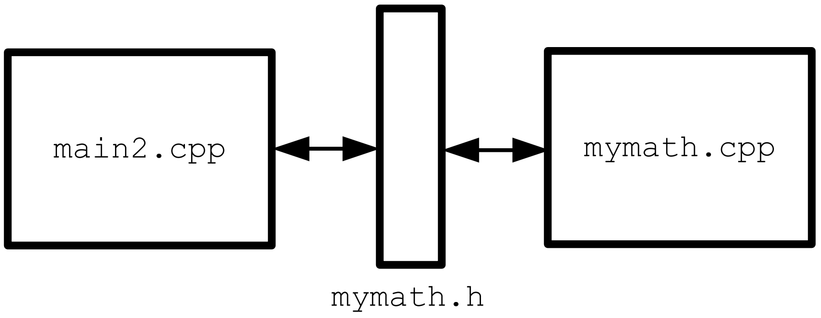


Of course any function in `testmax.cpp` (not just `main()`) can call the functions in `mymath.cpp`.

We have basically broken up our original program into two cpp file. The header file `mymath.h` provides an interface for `mymath.cpp`.

Why is that good?

Because if one day you need the `max()` and `min()` functions in another project, say the main source file is called `main2.cpp` then you can copy `mymath.cpp` and `mymath.h` to the project space of `main2.cpp` and you have the functions in `mymath.cpp` available to `main2.cpp`.



Exercise. Create a brand new project, copy `mymath.cpp` and `mymath.h` to the new project folder. Add these files to your project. Create a `main()` that calls `min()` and `max()`.

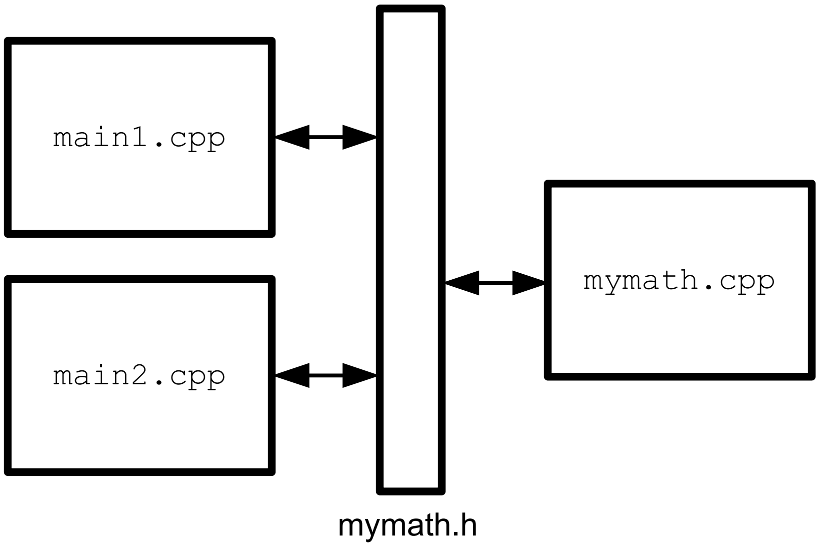
OK. Let's step back and take a look at what we've done. Previously we "re-use" the code for `max()` and `min()` by creating a function from chunks of code. This reduces code duplication.

But this is code reduction for **one single cpp file**.

To make the functions **more re-usable**, we took those out and put them into a new cpp file. We need a header file too.

This allows us to re-use code easily for two totally separate programs.

(Actually you don't even need to copy `mymath.cpp` and `mymath.h` to the new project. You can have one single copy of `mymath.h` and `mymath.cpp` but I don't want to go into that for CISS240.)



OK but what about

```
#include <iostream>
```

For header files for C++ standard cpp files you should write

```
#include <iostream>
```

which is actually the same as

```
#include "iostream.h"
```

So in other words

```
    #include <...> // for C++ standard header files
    #include "..." // for your header files
```

And remember that when you use `<...>` you do not have the `.h` part of the header filename.

You might say, “Well why doesn't the compiler just search for the function I'm calling? Just scan the hard disk for all cpp and header files and look for the function.”

True.

But compilers don't work that way because it could potentially take half an hour before the compiler reports an error that the function can't be found on your hard drive!!! Your hard drive is pretty huge, you know. For a really huge hard drive it might take even more time.

Exercise. Here's a previous program. Move all the function bodies to `heads.cpp` and create a header file (pun?) `heads.h`. Run your program and make sure it works. The global constants should be in the header file.

```
#include <iostream>

void printRules()
{
    const int MIN_HEADS = 2;
    const int MAX_HEADS = 10;
    std::cout << "To join MENSA you must have at "
              << "least " << MIN_HEADS
              << " and at most "
              << MAX_HEADS << std::endl;
    return;
}

int getHeads()
{
    int heads;
    std::cout << "How many heads to you have? ";
    std::cin >> heads;
    return heads;
}

bool passMinTest(int heads)
{
    const int MIN_HEADS = 2;
    if (heads < MIN_HEADS)
    {
        std::cout << "Too few! Try again!"
                  << std::endl;
        return false;
    }
    else
        return true;
}

bool passMaxTest(int heads)
{
    const int MAX_HEADS = 10;
    if (heads > MAX_HEADS)
    {
        std::cout << "Don't show off! Try again!"
                  << std::endl;
        return false;
    }
    else
        return true;

}

bool passTest(int x)
{
    return passMinTest(x) && passMaxTest(x);
}

int main()
{
    int heads = 0;

    printRules();
    heads = getHeads();
    while (!passTest(heads))
    {
        heads = getHeads();
    }
    std::cout << "OK. You can join MENSA."
              << std::endl;
    return 0;
}
```

"Why do I sometimes see Parameter Names for Function Prototypes?"

I already told you that function prototypes only need the name of the functions and the various types: the parameter types and return types.

However sometimes you see parameter names. For instance instead of this prototype:

```
int getSalary(int, int);
```

you might have this:

```
int getSalary(int employeeId, int overtime);
```

The reason for giving parametric names in function prototypes even though they are ignored by C++ is that they make the function prototype **easier to read and use**. This is especially the case where the prototype has many parameters of the same type. For instance this

```
int getSalary(int employeeId, int overtime);
```

is more useful than this:

```
int getSalary(int, int);
```

Giving parameter names also allows the programmer to document the function:

```
//--------------------------------------------------
// The getSalary() function accepts employeeId and
// overtime (in number of minutes) and returns the
// salary (in cents). Note that a valid employeeId
// ranges from 10000 to 99999.
// If the employeeId is not valid or if the overtime
// is negative the return value is -1.
//--------------------------------------------------
int getSalary(int employeeId, int overtime);
```

Exercise: integer array library

Write an integer array library that has files `IntArray.h` and `IntArray.cpp` that has the following features.

`array_init(x, x_len, val, n)`
copy integer `val` into `x` from index `0` to index `n - 1` and set `x_len` to `n`.

`array_assign(x, x_len, y, y_start, y_end)`
copy `y` from index `y_start` to `y_end - 1` to `x` starting at index `0`. `x_len` is set accordingly.

`array_concat(x, x_len, y, y_start, y_end)`
concatenate the subarray of `y` from index `y_start` to `y_end - 1` to `x` beginning at index `x_len`. `x_len` is changed accordingly.

`array_isequal(x, x_start, x_end, y, y_start, y_end)`
return true if and only if `x` from index `x_start` to `x_end - 1` is the same as `y` from index `y_start` to `y_end - 1`.

`array_replace(x, x_start, x_end, source, target)`
replace all occurrences of integer `source` in `x` from index `start` to `end - 1` with integer `target`.

`array_count(x, x_start, x_end, target)`
returns the number of times integer `target` occurs in `x` from index `x_start` to `x_end - 1`.

`array_max(x, x_start, x_end)`
return maximum value of `x` from index `x_start` to `x_end - 1`.

`array_min(x, x_start, x_end)`
return minimum value of `x` from index `x_start` to `x_end - 1`.

`array_max_index(x, x_start, x_end)`
return index of maximum value of `x` from index `x_start` to `x_end - 1`.

`array_min_index(x, x_start, x_end)`
return index of minimum value of `x` from index `x_start` to `x_end - 1`.

`array_isascending(x, x_start, x_end)`
return true if and only if `x` from index `x_start` to `x_end - 1` is sorted in ascending order.

`array_bubblesort(x, x_start, x_end)`
perform bubblesort on `x` from index `x_start` to `x_end - 1`.

`array_linearsearch(x, x_start, x_end, target)`
perform linear search on `x` from index `x_start` to `x_end - 1` searching for integer `target` and return its index in `x`.

`array_binarysearch(x, x_start, x_end, target)`

perform binarysearch on `x` from index `x_start` to `x_end - 1`
searching for integer `target` and return its index in `x`.

Exercise: C-string library

Look at the chapter 18 Characters and C-strings. There are a bunch of useful C-string functions in the C-string library. Implement your own C-string library in files `mystring.h` and `mystring.cpp`.

`strlen(s)`
returns the string length of C-string `s`

`strcpy(s, t)`
copy string `t` to `s`

`strclear(s)`
after calling this function `s` becomes `""`. This is the same as `strcpy(s, "")`.

`strcat(s, t)`
concatenate `t` to `s`

`strcmp(s, t)`
returns `0` if C-strings `s` and `t` are the same strings

`strreplace(s, source, target)`
replace all occurrences of character source by character target. For instance if `s` is `"hello world"`, after calling `strreplace(s, 'l', 'm')`, `s` becomes `"hemmo wormd"`.

`strleftstrip(s)`
remove all whitespace characters (`' '`, `'\n'`, `'\t'`) on the left of `s`. For instance if `s` is `"   abc   "`, after calling `strleftstrip(s)`, `s` becomes `"abc   "`.

`strrightstrip(s)`
remove all whitespace characters (`' '`, `'\n'`, `'\t'`) on the right of `s`. For instance if `s` is `"   abc   "`, after calling `strrightstrip(s)`, `s` becomes `"   abc"`.

There are actually a lot more C-string functions that comes with your C/C++ compiler.

Summary

A function prototype is a statement that looks like the header of a function. For instance the prototype of the function

```
double f(int x, int y, char z)
{
     if (z == 'a')
    {
        return x + y;
    }
    else
    {
        return x * y;
    }
}
```

is just

```
double f(int, int, char);
```

Once a prototype of a function f is given, the program can call the function; the body of f need not be defined before f is called.

Prototypes of functions can be kept in a file and then “#include” in a cpp file. The definition of the functions can be kept in a different cpp file.