23. Inline Functions

Objectives

- Write inline functions
- Understand the difference between an inline function and a non-inline function
- Write inline functions for multi-file compilation

This is a very short chapter because the concept of inline function is pretty simple from a beginner programmer's point of view. But later on you'll actually see inline functions popping up in many places (but in an implicit and hidden way) especially in C++ templates, generic programming, lambdas, metaprogramming. So understanding this concept is very important for *other* C++ features.

Inline functions

You have seen this before:

```
#include <iostream>

int max(int x, int y)
{
  if (x < y) return y;
  else return x;
}

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

(For very simple if-else statements, the above style is OK.)

Now add one single keyword:

```
#include <iostream>

inline int max(int x, int y)
{
  if (x < y) return y;
  else return x;
}

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

Note that it's common to see this coding style:

```
inline
int max(int x, int y)
{
  if (x < y) return y;
  else return x;
}
```

I suggest you use this style.

In terms of what a function can **do**, it doesn't make any difference if a function is inlined or not. The only difference is really in the speed of execution of the code.

An **inline function** is not exactly a function. When you compile a program with an inline function, what happens is that the compiler (more or less) copies the code to the place where you called the inline function. In other words, when you get C++ to compile the following program:

```
#include <iostream>

inline
int max(int x, int y)
{
  if (x < y) return y;
  else return x;
}

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

your compiler reads the program into its memory and changes it to this:

```
#include <iostream>

int main()
{
  int m;
  if (3 < 5) m = 5;
  else m = 3;
  std::cout << m << std::endl;
  return 0;
}
```

before generating the executable. There is really no `max()` function. In other words, an inline function is like a rubber stamp for a chunk of code.

If your code calls the inline `max()` function 100 times, the body of `max()` is copied 100 times.

Note that the copying of the body of the function to the places where the function is called is done when the compiler builds an executable file. Your actual cpp file is not changed; the above modification is done in the compiler's memory. You do not see the code duplicated in your source file.

Exercise. Write an inline `min()` function. Test your function.

Why inline functions?

Why do we have inline functions?

Like I said it's faster (see caveat later). Why?

Because there is some CPU cost in just making a function call. Look at the notes on Functions. Recall that to make a function call, the CPU must set up a stack which contains at least information on how to go back to the caller function. That takes time. It's the same when a function call returns. In other words, just **the act of calling the function and returning from a function call** (i.e. not including the computation done in the body of function called) **takes time**. In fact it's even possible that the cost of making a function call and returning from the call takes more time than the actual computation in the body of the function!!!

Inline functions calls do ***not*** involve function calls at all because the inline function's body has already been copied to the place where you make the function call.

"When should I inline a function?"

Now you might say: "In that case, let's inline every function! That will make my program super fast!"

Well ...

Yes, you can inline any function.

The problem is that the your program might become larger than what it should be.

Good choices for functions to be changed to inline functions are those which are very small, especially those which are used frequently. Function bodies with only one or two lines of code are good candidates.

Now for some caveats …

- If the compiler feels that the function body is too huge, it might actually ignore your request to inline a function. So an inline function is only a ***request*** to the compiler to inline the function.
- Some C++ compilers actually honor inline requests unless told it to do otherwise.
- If too many functions are inlined, the executable binary code can be huge, which of course means that you need more memory to run the code which means that you might be slowing it down!!! If the executable binary code is too huge it might cause memory thrashing where memory pages constantly gets swapped in and out of your computer memory.

Multi-file compilation

Although an inline function is similar to a regular function, there is **one difference** if you want to use inline functions in a multi-file project.

The whole inline function (with the function body) goes into the header file. You do NOT write prototypes for inline functions.

Here's our max example using a header file and a separate cpp file. It ***won't*** work. Try it now and look at the error message:

```
#include <iostream>
#include "test.h"

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

```
// test.h

#ifndef TEST_H
#define TEST_H

inline
int max(int x, int y);     // prototype of max

#endif
```

```
// test.cpp
#include "test.h"

inline
int max(int x, int y)      // definition of max
{
  if (x < y) return y;
  else return x;
}
```

Do this instead:

```
#include <iostream>
#include "test.h"

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

```
// test.h

#ifndef TEST_H
#define TEST_H

inline
int max(int x, int y)
{
  if (x < y) return y;
  else return x;
}

#endif
```

(You can get rid of the test.cpp file) Now it works.

Remember that if you want to have multi-file projects and your inline functions are not in your main cpp file, then the **inline function bodies must be in header files**. Don't you forget that!

Note that you can have prototypes for inlined function:

```
// test.h

#ifndef TEST_H
#define TEST_H

int max(int x, int y);

inline
int max(int x, int y)
{
  if (x < y) return y;
  else return x;
}

#endif
```

The inline keyword follows the function definition. You do not need the inline keyword for the prototype. Make sure you run this so that you remember how to write inlined prototypes. It will be very important when you get into very complex modern C++ programming.

You can also check that if you do use the inline keyword for the prototype, your compiler most likely will not give you an error.

Exercise. In the above, add an inline `min` function in the header file to compute the minimum of two integer functions and then an inline `swap` function to swap the values of two integer variables.