25. Function Overloading

Objectives

- Understand function signatures
- Write overloaded functions

Too many function names to remember!

This is an easy:

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

But what if I want to compute the max of two doubles? If I use the above:

```
int max(int, int);

int main()
{
    std::cout << max(3, 42) << ' '
              << max(3.14, 2.718) << std::endl;
    return 0;
}

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

I will lose the fractional part of the true maximum (because when the function is called, `x` will have a value of 3 and y will have a value of 2.)

In other words 3.14 is typecasted to an integer before it's given to `x`.

So I have to write another max function for doubles. To make sure I don't have two functions with the same name I do this:

```
int max(int, int);
double max2(double, double);

int main()
{
    std::cout << max(3, 42) << ' '
              << max2(3.14, 2.718) << std::endl;
    return 0;
}

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

double max2(double x, double y)
{
    if (x < y) return y;
    else return x;
}
```

Now the nightmare begins ...

What if I want the max of an array of integers? An array of doubles? Looks like I need:

```
int max(int, int);
double max2(double, double);
int max3(int[], int len);
double max4(double[], int len);
...
```

YIKES!!! HELP ME MAMA!!!

MY HEAD IS GOING TO BLOW UP JUST REMEMBERING WHICH IS WHICH!!!

Function Overloading

This is from the previous section:

```
int max(int, int);
double max2(double, double);

int main()
{
    std::cout << max(3, 42) << ' '
              << max2(3.14, 2.718) << std::endl;
    return 0;
}

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

double max2(double x, double y)
{
    if (x < y) return y;
    else return x;
}
```

Make this change:

```
int max(int, int);
double max(double, double);

int main()
{
    std::cout << max(3, 42) << ' '
              << max(3.14, 2.718) << std::endl;
    return 0;
}

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

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

Run it.

:O

How come C++ is not confused by two functions with the ***same name***?

The reason is because C++ recognizes functions by their **names *and* the types of the parameters**. Here are the prototypes of the above function:

```
int max(int, int);
double max(double, double);
```

You can think of it this way: the integer `max()` function is identified by `(max, int, int)` and the double `max()` function is identified by `(max, double, double)`.

You see what happens is that the actual function name of `max(int, int)` might be `max_int_int()` and the name of `max(double, double)` might be `max_double_double()`. I say "might" because it depends on the compiler. For instance another compiler might use `max____int____int()` instead of `max_int_int()`. This change of function name is called name mangling.

The types of the parameters is called the **signature** of the function. So the signature of

```
int max(int, int);
```

is `(int, int)`.

The important thing to remember that a function is identified by its **name** *and* **signature**.

Of course now that a function is determined by its name and signature, this tells you that you cannot have two different functions with the same name and signature. Therefore this is BAD:

```
void bad(int x, double y, char z)
{
    std::cout << "first bad ..." << '\n';
    ...
}

int bad(int x, double y, char z)
{
    std::cout << "second bad ..." << '\n';
    ...
}
```

Yes, yes, yes … the return type is different … but C++ only looks at the name and the signature when choosing a function to execute.

So we revise the way C++ searches for identifiers (i.e. names):

- The search for variables is still the same
- The search for functions when they are called is different.
    - If the function name can be found and the types of the arguments passed in matches the function signature (the type of the parameters), then that function is used.
    - Otherwise, C++ will try to typecast the arguments to

values that will match a signature.

Here's one more example. Run it.

```
#include <iostream>

void f(int, int, int);
void f(double, double, double);

int main()
{
    f(1, 2, 3);
    f(1.1, 2.2, 3.3);
    return 0;
}

void f(int x, int y, int z)
{
    std::cout << "f(int, int, int)" << std::endl;
}

void f(double x, double y, double z)
{
    std::cout << "f(double, double, double)"
              << std::endl;
}
```

When you have several functions with the same name (but different signatures!), we say that the function is **overloaded**.

Exercise. Here are two function prototypes:

```
// computes the maximum value of the array passed in
int max(int[], int len);
double max(double[], int len);
```

Implement the function bodies and test them in a `main()`.

Exercise. Implement the following and test your code.

```
// Functions return the minimum of values passed in
// either through parameters or values in an array
int min(int, int);
double min(double, double);
int min(int[], int len);
double min(double[], int len);
```

Exercise. Implement and test your functions for the following prototypes:

```
// area functions
double area(double, double); // area of rectangle
                             // with given sides
double area(double);         // area of circle with
                             // given radius
```

Remember earlier in the semester when I said that there are two + operators. Example:

1 + 2	integer addition operator
1.2 + 3.4	double addition operator

At that point I told you to remember this important distinction. This is very similar to function overloading, except that these are operators and not functions. We say that + is an **overloaded operator**. In CISS245 you will learn to define your own operators so for instance you can define your own operators including +.

WARNING: `const int` and `int`

Here's a warning: “const int” is the same type as “int”. The “const” is about the nature of the value that a variable refers to, i.e. for

```
const int x = 42;
```

the const simply means that x is a constant variable. The type of value that x has is still an `int`. Now why is this important in the context of function overloading? Well … the above implies that

```
void f(int x)
{
}

void f(const int x)
{
}

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

will not compile because the two functions `f` above have the same signature. Make sure you at least compile the above and read the error message from the compiler so that when this error bites you, you'll know why.

Exercise. What about references? Are `int&` and `int` considered the same too? In other words does the following compile:

```
void f(int x)
{
}

void f(int & x)
{
}

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

Why Overloading?

The reason why we have function (and operator) overloading is clear: Many functions perform similar operations and we want to remember fewer names.

This is also the case in Math. Not only is + used for addition of real numbers, it's also used for things like addition of functions, addition of vectors, addition of matrices, etc. So having function/operator overloading in a programming language allows us to design software to match the mathematical language/symbols that we are used to.

Even in daily use of the English language, we see the same word used similar but different situations: open a door, open a jar, open my wallet, open your mind, etc.

Overloading is a software technique for controlling software complexities.

(Some older programming languages do not have function overloading. Some don't even allow you to define your own operators. Also, there are programming languages where overloading is not allowed. In fact there are programming languages where the symbol for adding integers is different from the symbol for adding doubles.)

Return type warning

I've already mentioned this … but's it's such a common error that it's a good idea to repeat it …

… the **return type is *not* part of the identification of a function**. In other words the following is **incorrect**:

```
int f(int);    // identified as f, int
double f(int); // identified as f, int
```

Why? Because of this ...

Recall that it's OK ***not*** to use the return value of a function. For instance this is a valid piece of code:

```
int f()
{
    return 42;
}

int main()
{
    f(); // return value is not used
    return 0;
}
```

In fact even this is OK:

```
int main()
{
    42;
    return 0;
}
```

Now suppose you have a C++ compiler (called Crazy C++) that identifies function by their names, signatures **and return types**. Then you have this **problem**:

```
int f(int)
{}

double f(int) // should be different from above f()
{}

int main()
{
    f(42); // Crazy C++: ... which one to use???
           // the return type is not indicated!!!
    return 0;
}
```

Make sure you run this program.

Type Casting and Ambiguous Invocation

Recall that this is how C++ searches for functions (see above):

- If the function name can be found and the types of the arguments passed in matches the function signature (the type of the parameters), then that function is used.
- Otherwise, C++ will try to typecast to something suitable.

Try this:

```
#include <iostream>


void f(int, int, int);


int main()
{
    f(1, 2, 3);
    f(4.4, 5.5, 6.6);
    return 0;
}


void f(int x, int y, int z)
{
    std::cout << "f(int, int, int)" << std::endl;
}
```

Now this:

```
#include <iostream>


void f(int, int, int);
void f(double, double, double);


int main()
{
    f(1, 2, 3);
    f(1.1, 2.2, 3.3);
    return 0;
}


void f(int x, int y, int z)
{
    std::cout << "f(int, int, int)" << std::endl;
}


void f(double x, double y, double z)
{
    std::cout << "f(double, double, double)"
```

```
    << std::endl;
}
```

So far so good …

But now ...

Exercise. Add `f(1, 2, 3.4);` to the main of the program above and run it.

```
...

int main()
{
  f(1, 2, 3);
  f(1.1, 2.2, 3.3);
  f(1, 2, 3.4);
  return 0;
}

...
```

Does it work?

There are two possibilities: C++ can call

`void f(int, int, int)`	type cast 3.4 to 3
`void f(double, double, double)`	type cast 1 to 1.0, 2 to 2.0

If there are two or more possible function calls after attempts to type cast, C++ will choke and yell at you. This type of error is called

ambiguous invocation.

As mentioned before: Always be explicit in your code -- as much as possible avoid implicit typecasts that are done behind your back. Something like:

```
int x = 0;
double y = x;
```

should be written

```
int x = 0;
double y = double(x);
```

Exercise. Are there any ambiguous function calls? Find and circle if any.

```
void f(int x, int y, int z)
{}

void f(int x)
```

```
{}

void f()
{}

void f(int x, int y, double z)
{}

int main()
{
    f(1);
    f(1, 2, 3);
    f(1.1, 3.3, 4.4);
    return 0;
}
```

Verify with your C++.