overloaded functions

name + param types = function

overloaded function

We've seen this before. An overloaded function is a function that

- has one name
- represents different operations depending on its parameter types

C++ supports function overloading

name mangling

Real process, how the compiler creates a unique name based on the function name and its associated types.

- mangled name allows for look up of the correct function
  - nm shows mangled names
  - http://demangler.com/

function signature

Function signature consists of:

- function name
- function return type
- the types, and their order, of the parameters

Names of the parameters do not matter!

Uniquely identifies (or should) a function

Two different functions with the same name

two different functions (Ex 6.5)

```
void swap (double &d1, double &d2){
  cout << "This must be the double swap"<<endl;
  double temp;
  temp = d1;
  d1 = d2;
  d2 = temp;
}
```

```
void swap (int &i1, int &i2){
  cout << "This must be the int swap"<<endl;
  int temp;
  temp = i1;
  i1 = i2;
  i2 = temp;
}
```

resolving can be complicated

You can find various refs (https://en.cppreference.com/w/cpp/language/overload_resolution) as to the rules.

- the problem is basically conversion. What happens if a conversion is available that might convert one type to another?

Ex 6.6

```
int f(){
  cout << "f, no arg"<<endl;
}
int f(int i){
  cout << "f, 1 int arg"<<endl;
}
int f(int i,int j){
  cout << "f, 2 int arg"<<endl;
}
int f(double x, double y=3.14159){
  cout << "F, 2 arg with default}"<<endl;
}

int main (){
  f(5.65);  // which one???
  f(42, 2.65);  // which one???
}
```

Easier to have happen then you think

This seems like a bad place to end up, but because code can be written in pieces by different people, conversion functions might creep in that allow for this kind of problem.

Beware!

A word on const

Trying to differentiate parameter types based on top-level const does not work. These are ***the same functions***!

```
long my_fun (const long p1){
   cout << "const fn"<<endl;
}
```

```
long my_fun(long p1){
   cout << "reg fn" <<endl;
}
```

```
int main(){
   const long c_long = 1;
   long my_long = 2;
   my_fun(c_long);
   my_fun (my_long);
}
```

templates

making a pattern of a function for multiple types

Example 6.7

overloading, double edge sword

Nice to be able to overload a function based on types

What a pain, some function (very general) requires that I re-write it for every type, especially for any new one I create!

template

The way to get around it is called a **template.** A template is a ***pattern***, a pattern that can be used to ***create a function*** with whatever types we want.

Need to get that a ***template is not a function***, it is how to create a function with some type information set

basis of everything in the STL

While pointers are a basis for a lot of how C (the underlying language) works, templates are the basis for C++/STL and how it really solves many problems of generality with types.

Ex 6.7

```
template <typename my_type>
void swap (my_type &first, my_type &second){
   my_type temp;
   temp = first;
   first = second;
   second = first;
}
```

type in-between
< >

keyword

template type
variable

```
template <typename my_type>
void swap (my_type &first, my_type &second){
   my_type temp;
   temp = first;
   first = second;
   second = temp;
}
```

use the template type
var everywhere
you need template
behavior

```
template <typename my_type>
void swap (my_type& first,
my_type& second){
   my_type temp;
   temp = first;
   first = second;
   second = temp;
}
```

1) look for swap with two ints

```
int i=1, j=2;
swap(i,j);
```

3. Call new fn

2) substitute `int` for `my_type` ***create*** the function

```
void swap (int& first, int&
second){
   int temp;
   temp = first;
   first = second;
   second = temp;
}
```

generic function

By writing the function as a template, we can write a *generic function*:

- a function which, even in C++ (which is type crazy), is generic **for all types**.

Remember: a template is a pattern to make a function. ***A template is not a function***

force the type

Typically the compiler deduces the type for substitution in the template from the provided arguments

You can force (though you must be careful) the type used, but it has to work with the args and the created function

Ex 6.8, force the template type

Invocation

```
double result;
long i=1, j=2;
result = swap<double>(i,j);
```

template type
directly indicated

Will see this again and again. We specify in the invocation the type we want used in the template