More Functions

"Worrying is the most natural and spontaneous of all human functions."

-- Lewis Thomas

reference parameters

By default, you ***copy*** the values from argument to parameter. But you can change that:

- if you declare the type of the parameter to be a reference, then the arg and the param refer to the same value
  - a change to the function parameter changes the invoker's argument

Ex 7.1, swap with references

```
void swap (long & first, long & second){
// a reference is an alias
  long temp;
  temp = first;
  first = second;
  second = temp;
}
```

`void` means no return

parameters are references

change the reference parameters and you change the corresponding invoker arguments

Ex 7.2, pointer parameters

You can do the same thing by passing pointers to original argument:

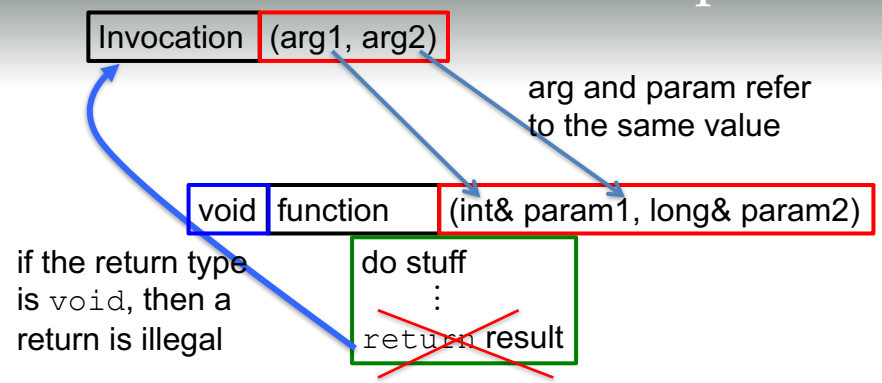
- through the pointer, you can change the argument
- you can set them as const as well (pass a kind of copy, no changes)

compare defs

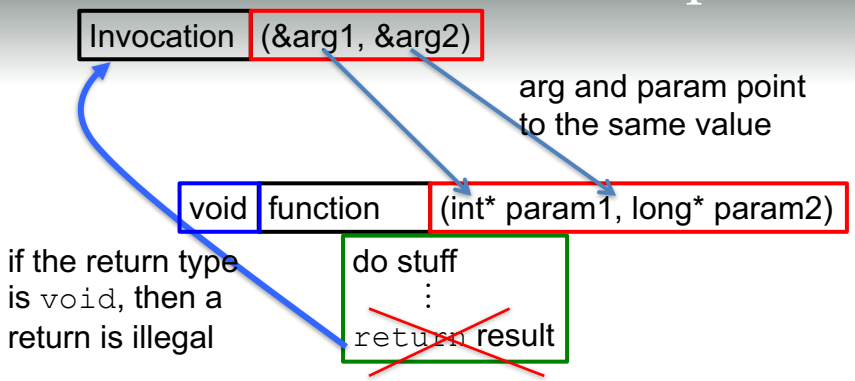
```
// Ex 7.1
void swap (long & first, long & second){
  long temp;
  temp = first;
  first = second;
  second = temp;
}
```

```
// Ex 7.2
void swap (long *first, long *second){
   long temp;
   temp = *first;
   *first = *second;
   *second = temp;
}
```


process



process



invocations

```
int main (){
   // call with refs
  long one=100, two=200;
  swap(one, two)
}
```

```
int main (){
  // call with ptrs
  long one=100, two=200;
  swap(&one, &two);
}
```

best of both worlds

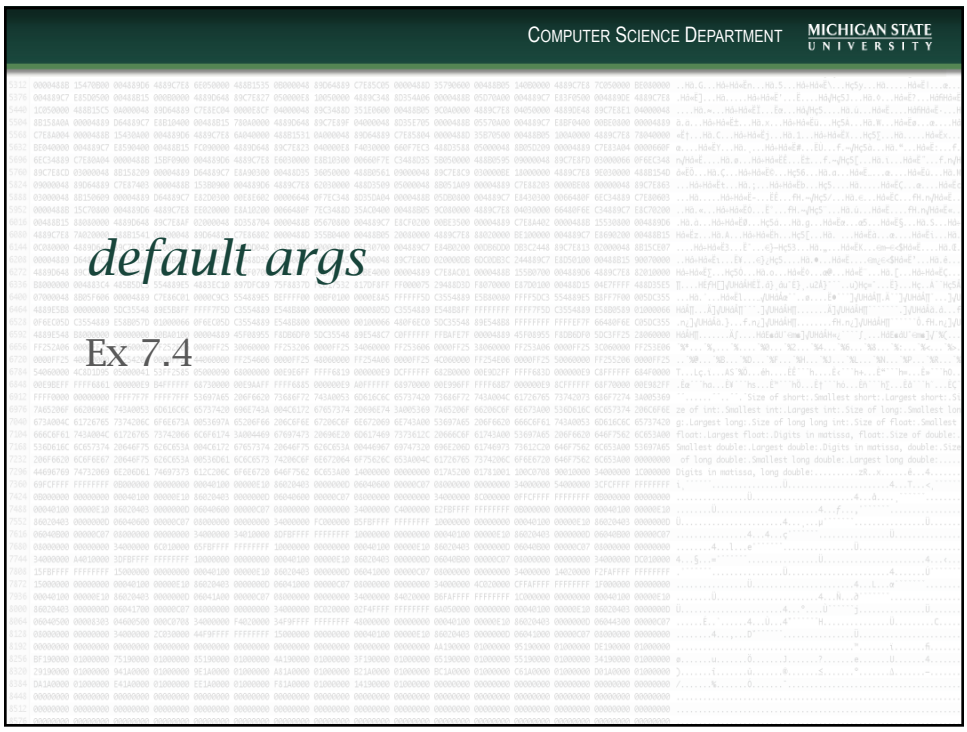
If you want to pass args-to-params by reference (to avoid copying) but do not want to allow the function to change such parameters, make them `const`

- you can add `const` to a ref parameter, and in so doing make that "gate" a constant, cannot change the underlying value ***through*** it
- still a copy

Ex 7.3

```
double circ_area(const double &radius,
                 const double &pi){
  return pow(radius,2) * pi;
}
int main(){
  double r;
  double pi = atan(1.0)*4.0;  // calc pi
  cout << "Give me a radius:";
  cin>>r;
  cout <<"Circle of radius "<<r
       <<" has area:"
       <<circ_area(r,pi)<<endl;
}
```

pass by reference but no changes!



default args

Ex 7.4

setting defaults

You may set the default values for a parameter.

- if the parameter is not provided, the default is used
- if the parameter is provided, the provided value is used

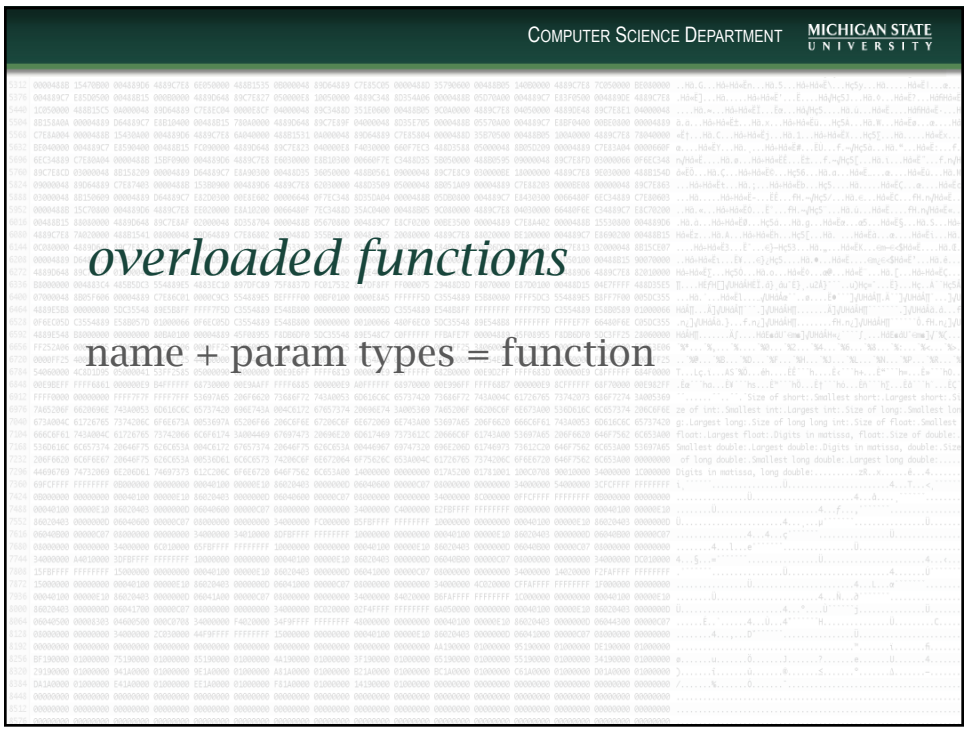
```
int increment (int val, int inc=1){
  val += inc;
  return val;
}

int main (){
  int my_int = 27;
  cout << increment(my_int,5);  // 32
  cout << increment(my_int);    // 28
  // cout << increment();  need val!
}
```

variable inc has a default of 1.

order dependency

There is an order dependency here. You must have all the required parameters (those without defaults) before any default argument parameters!

You cannot mix and match, nor can you call out by name (in the invoker) which parameter you set. Everything must be done in order



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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

Example 7.5

two different functions!!

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

Example 7.5

```
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

Section 6.6 of the book goes through the "rules" for deciding which, if any, function is appropriate for a set of arguments.

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

Ex 7.6

```
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 7.7

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.

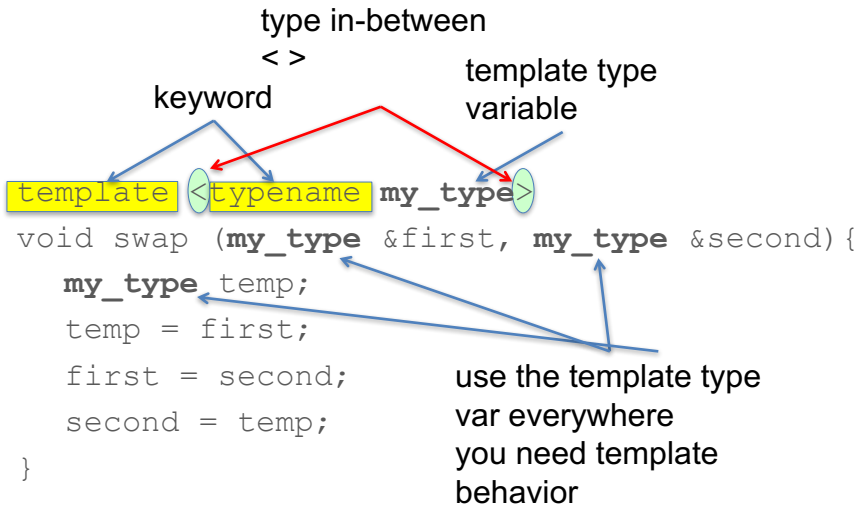
Need to get that a ***template is not a function***, it is how to create a function with some type information set

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 7.7

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

```
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. It 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 7.8, 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

trailing return type and auto

If you want to use an `auto` for a return type, especially in a template, you use a trailing return type

```
auto my_fun(int x, int y) -> decltype(x + y)
```


pointers to functions

Useful topic, look at section 6.7 of the book