

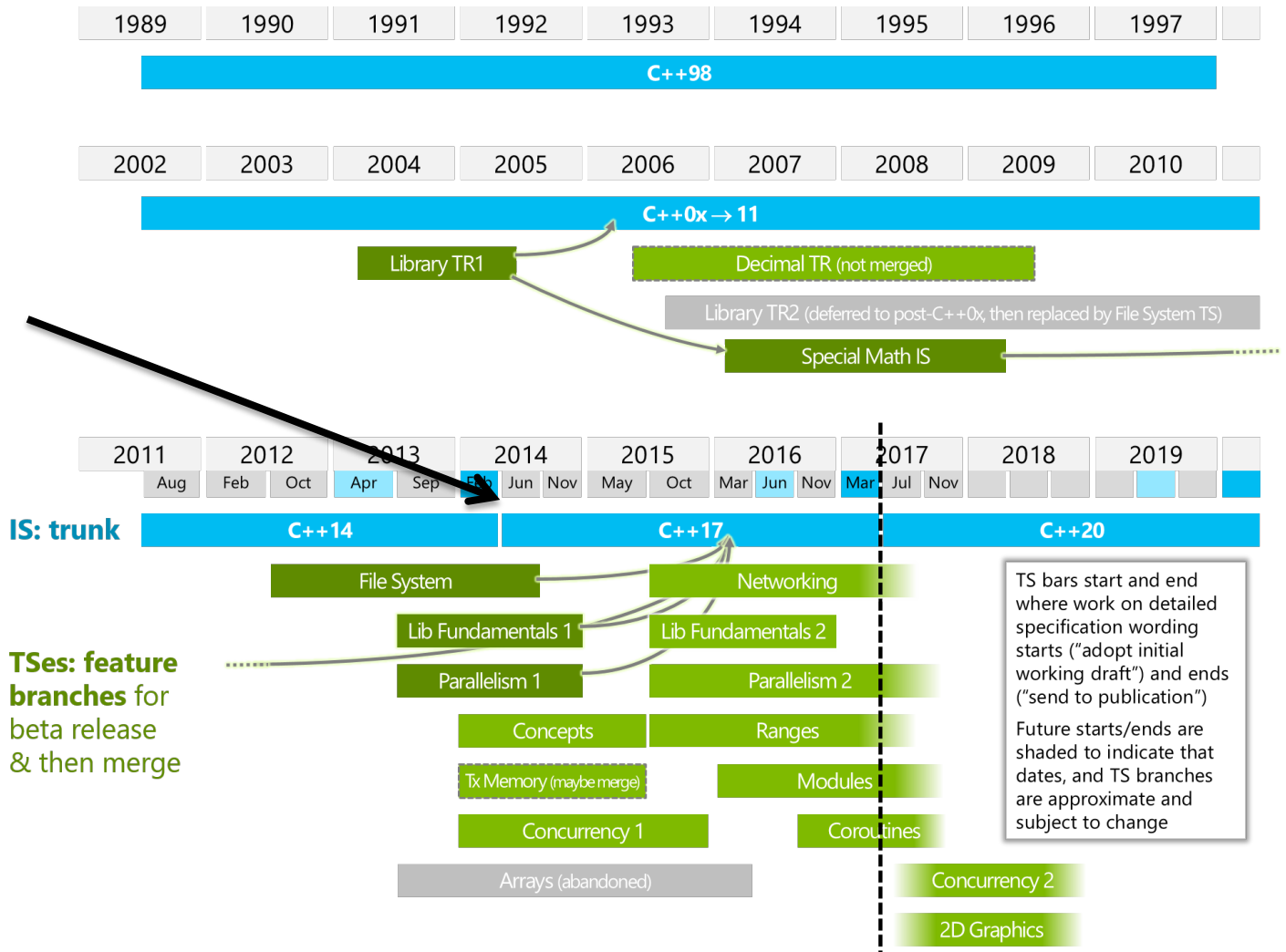
# C++ in a Nutshell

Devin A. Matthews  
UT Austin

MolSSI Software Summer School 2017

# C++ is a big and complex language

Current standard  
(C++14) is more  
than 1300 pages!



...but “you don’t pay for  
what you don’t use”

- C++ features are designed to be **zero-cost**, i.e. if your program doesn’t use a feature, it won’t slow it down.
- This is especially important for more complex features.
- Very few people know *all* of the standard. You can use what you are comfortable with and learn as you go along.

# C++ is an evolving language

- **C++11**
  - First major update in 13 years.
  - Brings many improvements, ushered in what is called “modern C++”.
  - This is what we’ll be focusing on.
- C++14
  - Relatively minor update but brings some nifty new features
- C++17
  - Brings lots of stuff like folds, structured bindings, constexpr if, class template deduction, std::variant/optional/any, and more.
- New standards expected every three years.

# Preamble: from code to instructions

- C++ programs are structured as one or more source files (called **translation units**).
- The translation units are **compiled** (or **built**) into sequences of machine instructions by the **compiler**.
- The compiled code is generally first placed in an **object file** (one for each translation unit).
- The object files are then **linked** into an executable or library.

# Preamble: from code to instructions

Compile:

```
g++ -std=c++11 -c -o test.o test.cxx
```

The diagram shows the command `g++ -std=c++11 -c -o test.o test.cxx` with blue lines connecting it to four callout boxes. A line from `g++` points to a box explaining it's the compiler name. A line from `-std=c++11` points to a box explaining it's the C++11 standard. A line from `-c` points to a box explaining it's for compilation only. A line from `-o test.o` points to a box explaining it's the output file. A line from `test.cxx` points to a box explaining it's the input file(s).

Just compile and don't link.

Put the output file here.

This is the name of the compiler. Other compilers include clang++, icpc, xLC, etc.

We want the C++11 standard and not the default of C++98.

Input file(s) go at the end. C++ files end with (most common first):

- .cpp
- .cxx
- .cc
- .C

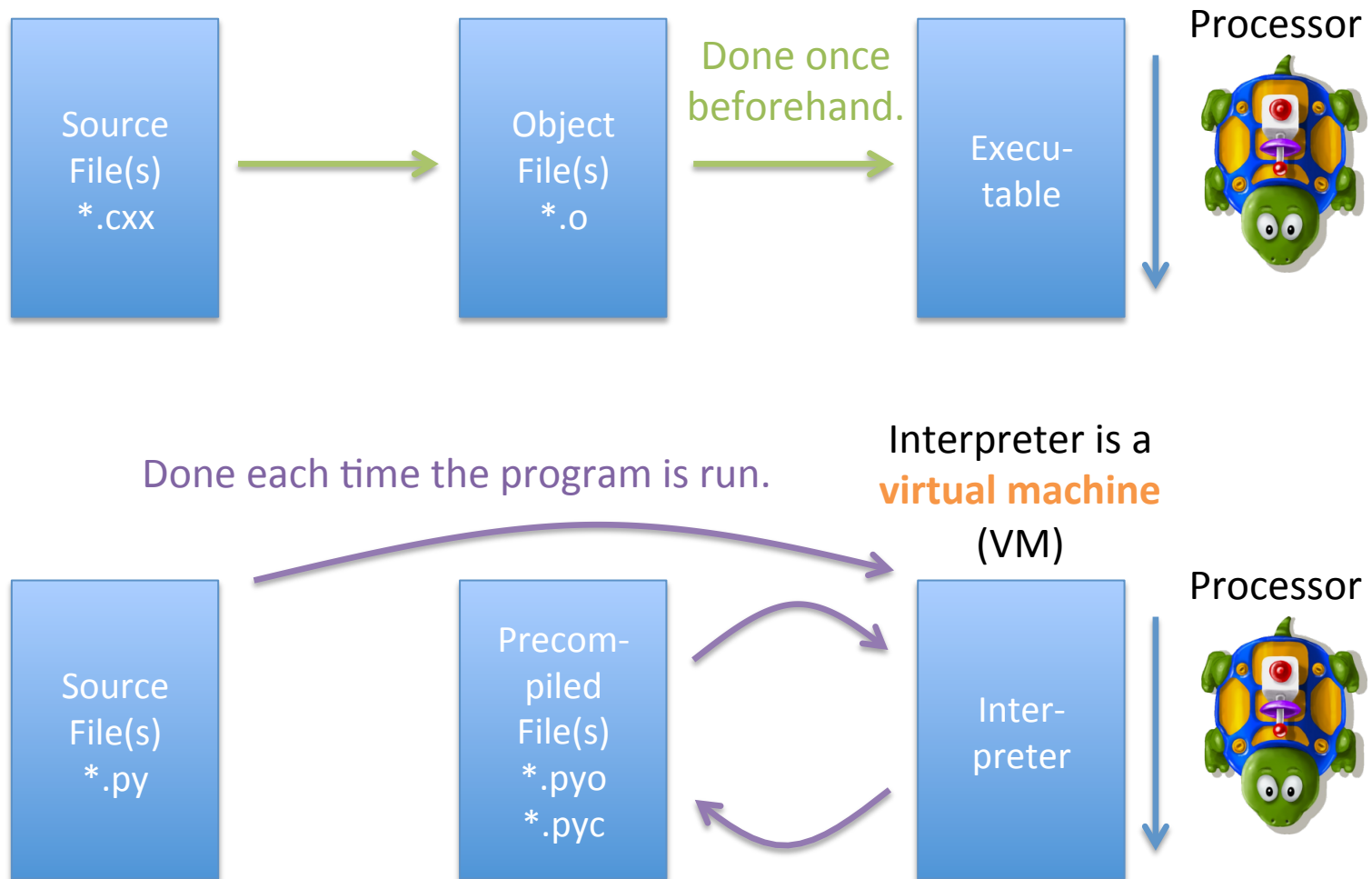
Link:

```
g++ -o test.x test.o
```

# Compiler Flags

Effect	Flag(s)	Compiler	Notes
Don't optimize and generate debugging information	-g -O0	all	This can make for very slow programs. g++ also has "-g -Og" which is a little faster
Optimize a little.	-O1	all	
Optimize some more.	-O2	all	
Optimize a lot.	-O3	all	A good choice for production builds.
Do some "unsafe" math optimizations	-ffast-math -fp-model fast	g++/clang++ icpc	May or may not be acceptable depending on accuracy needed.
Turn on lots of warnings.	-Wall	all	Don't ignore warnings!
Optimize for a particular architecture.	-march=... -x...	g++/clang++ icpc	-march=native and -xHost will optimize for the current machine.
Use the C++XY standard.	-std=c++XY	all	-std=c++11 is a good baseline with widespread support.
Link in an external library.	-l... (ell)	all	
Look for libraries in this path	-L...	all	
Look for header files in this path	-I... (EYE)	all	

# Comparison to Python





# Simplifying compilation

- **Makefiles** offer a way to automate building, compiling only those files that have changed.

**Variables** hold common data like the compiler and flags.

```
CXX=g++  
CXXFLAGS=-std=c++11 -O3 -march=native  
LDFLAGS=-lblas -llapack
```

The default **rule** is the first.

```
all: test.x
```

Rules tell how to build a **target** from the **dependencies**. The target is only built if one of the dependencies is newer.

```
test.x: test1.o test2.o test3.o  
$(CXX) -o test.x $^ $(LDFLAGS)
```

Pattern rules can match multiple targets.

```
%.o: %.CXX  
$(CXX) -c -o $@ $^ $(CXXFLAGS)
```

**Automatic variables** are set for each rule.

- Build systems such as **autoconf** and **CMake** take the automation further, including customization and detection of dependencies.

# Basic C++ Syntax: Hello World

```
#include <iostream>
#include <string>

// This is the main program
int main(int argc, char** argv)
{
    std::string continents =
        {"N. America", "S. America",
         "Europe", "Asia", "Africa",
         "Australia", "Antarctica"};

    for (int i = 0; i < continents.size(); i++)
    {
        std::string& continent = continents[i];
        std::cout << "Hello " << continent << "!\n";
    }

    return 0;
}
```

# Basic C++ Syntax: Hello World

```
#include <iostream>
#include <string>
```

Comments start with `//` or are surrounded by `/*...*/`.

```
// This is the main program
int main(int argc, char** argv)
{
```

C++ doesn't care about extra whitespace (including newlines).

```
    std::string continents =
        {"N. America", "S. America",
         "Europe", "Asia", "Africa",
         "Australia", "Antarctica"};
```

So, the end of a **statement** has to have a semicolon.

```
    for (int i = 0; i < continents.size(); i++)
    {
        std::string& continent = continents[i];
        std::cout << "Hello " << continent << "!\n";
    }
```

```
    return 0;
}
```

You can put more than one statement (or even the whole program!) on one line, but this is usually bad style.

# Basic C++ Syntax: Hello World

```
#include <iostream>
#include <string>

// This is the main program
int main(int argc, char** argv)
{
    std::string continents =
        {"N. America", "S. America",
         "Europe", "Asia", "Africa",
         "Australia", "Antarctica"};

    for (int i = 0; i < continents.size(); i++)
    {
        std::string& continent = continents[i];
        std::cout << "Hello " << continent << "!\n";
    }
    return 0;
}
```

← A reference to "continent" here is an error.

Braces ({} ) denote a **block**. Each block defines a **scope**.

Variables can be defined in any scope, and live until the end of the containing scope.

Variables in an inner scope can have the same name as variables in an outer scope. This is called **shadowing**.

# Basic C++ Syntax: Hello World

```
#include <iostream>
#include <string>

// This is the main program
int main(int argc, char** argv)
{
    std::string continents =
        {"N. America", "S. America",
         "Europe", "Asia", "Africa",
         "Australia", "Antarctica"};

    for (int i = 0; i < continents.size(); i++)
    {
        std::string& continent = continents[i];
        std::cout << "Hello " << continent << "!\n";
    }

    return 0;
}
```

C++ is a **strongly-typed language**. Every variable has a **type**, and the type cannot change.

C++ supports the usual logic and math operators: +, -, \*, /, %, ++, --, <, >, <=, >=, ==, !=, ~, |, &, ^, ||, &&, !, =, |=, &=, ^=, +=, -=, \*=, /=, <<, >>, <<=, >>=, as well as some others: (), [], ->, &, \*, ?:, .\*, and ",," (comma)

Many operators can be overloaded (i.e. they do different things to different types). Ex: what does this "<<" do? (Hint, it's not shift left)

# Types in C++

- C++ has the basic types:
  - Integral:
    - **char** (1 byte), **short** (2 bytes), **int** (4 bytes), **long** (4 or 8 bytes), and **long long** (8 bytes). Each of these can be either *signed* (default except char) or *unsigned*.
  - Floating point:
    - **float** (4 bytes), **double** (8 bytes), and **long double** (8-16 bytes—not very useful in practice).
  - Boolean:
    - **bool** (1 byte usually, possible values are **true** and **false**).
  - Character types:
    - **char** (again), **wchar\_t**, **char16\_t**, **char32\_t**.
  - Other:
    - **void** (i.e. “no type”), **nullptr\_t**.

# Types in C++

- New types can be built using **classes** (discussed later), but we can also make new names (called **typedefs**) for existing types:

```
typedef int my_int;  
my_int x = 3;
```

```
using my_double = double;  
my_double z = 1.0;
```

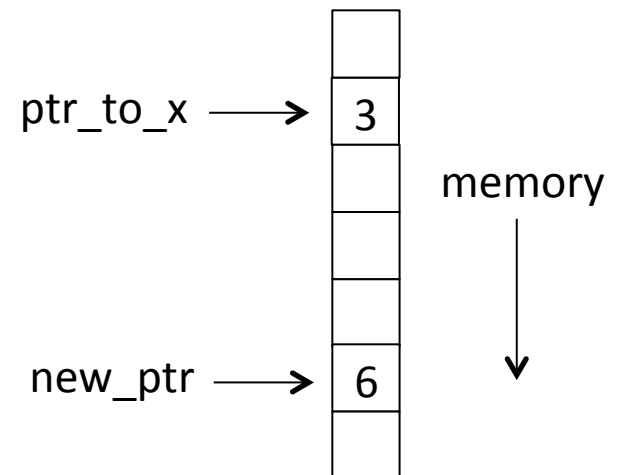
New preferred syntax.

- The standard library defines many typedefs such as **size\_t**, **ptrdiff\_t**, **int[8/16/32/64]\_t**, etc.

# Pointers

- For every type, there is also a special type called a **pointer**. E.g. **int\*** is a pointer to an **int**.
  - You get pointers by taking the address of objects with **&**.
  - You access the object being pointed to with the dereference operator **\***.
- A pointer stores the location in memory of an object. It is not an integer, but you can do some math with it:

```
int x = 3;  
int* ptr_to_x = &x;  
int* new_ptr = ptr_to_x+4;  
// this is a bad idea because  
// nothing exists there...  
*new_ptr = 6;
```





# Arrays

- An **array** is declared using [] after the name:

```
int array_of_5_ints[5]; // uninitialized
double array_of_doubles[3] = {1.0, 2.0, 3.0};
double another_array[] = {1.0, 2.0}; // size is 2
double big_array[100] = {}; // zero-initialize
int twod_array[3][3] = {{1,2,3},{4,5,6},{7,8,9}}; // row-major
```

- Elements are accessed with [] too:

```
array_of_doubles[1] = 5; // now it holds {1.0, 5.0, 3.0}
twod_array[0][0] *= 5;
```

- Arrays are basically just pointers and vice versa, but some times casts are needed:

```
double* ptr = array_of_doubles; // error, it needs a cast
double* ptr = (double*)array_of_doubles;
ptr[1] = 5; // exactly the same as above
1[ptr] = 5; // bonus Q: does this compile and what does it do?
```

# Operator new and heap vs. stack

- Variables declared directly in a scope are put on the **stack**, a special area of memory. Putting something on the stack costs almost nothing.
- Other variables can be put on the **heap** (memory outside the stack) with the **new** operator:

```
// you can only get to the heap with a pointer
double* x = new double();
*x = 4.0;
// alternatively: double* x = new double(4.0);
```

- But you have to remember to release the memory with the **delete** operator (C++ doesn't have garbage collection):

```
delete x;
```

- Variables on the stack are automatically cleaned up.

Well-written C++  
programs have *few or*  
*no* pointers in them and  
*never* use new/delete  
directly!

# References

- A **reference** is an alias to another variable. Unlike pointers, they can't be changed once initialized (i.e. references are *usually* “safe”).
- References are initialized (**bound**) *without* the **&** operator.

```
int x = 3;  
int& y = x;  
y = 4;  
// x is now also 4
```

```
// bonus Q: what is  
// the value of z?  
double z;  
// alternative syntax  
double& w(z);
```

# Lvalues and rvalues

- **Rvalues** are things that can only go on the right side of an assignment (= operator):
  - `int x; x = 3; // OK, so x is not an rvalue`
  - `1 = 0; // nope, "1" is an rvalue (actually a literal)`
  - `function_that_returns_an_int() = 0;`  
`// doesn't work, the return value is a temporary`
- **Lvalues** are things that can go on the left of =:
  - `int x; x = 3;`
  - `function_that_returns_an_int_ref() = 6;`  
`// OK, normal references are "lvalue references"`
- If we want to reference an rvalue we can use a special **rvalue reference**:
  - `int& x = function_that_returns_an_int(); // doesn't work`
  - `int&& x = function_that_returns_an_int();`  
`// OK, we capture the temporary`
  - `x = 3; // (Why does this work? Once it has a name`  
`// it is now an lvalue)`

# The const keyword

- Sometimes we want to indicate to the compiler that the value of a variable shouldn't change. This is indicated with the **const** keyword:

```
const int x = 3; // x will always be 3
x = 4; // compiler error
int& y = x; // also an error, otherwise we could
            // change x through y
```

- Const doesn't suddenly change how the program runs when you take it out (usually!), but it helps the programmer to keep themselves from making dumb mistakes.
- Const is also useful in public interfaces where you don't want users messing with your data.
- A program where every variable that should not change is marked const is called **const-correct**.

# Casts

- **Casts** transform one type into another (a **conversion**). They come in several flavors:
  - **static\_cast<NewType>(value)**
    - Converts 'a' into a meaningful value of type 'TypeB' if possible.
    - Ex: `double x = static_cast<double>(4);`
  - **const\_cast<MutableType>(const\_value)**
    - Adds or gets rid of the const qualifier. Avoid unless absolutely necessary.
  - **dynamic\_cast<NewType>(value)**
    - Cast which can look up the "real" type of 'value' at run-time. Outside our scope.
  - **reinterpret\_cast<NewType>(value)**
    - Pretend that the raw storage of 'value' is of type 'NewType'.
    - Ex: `int64_t& bits = reinterpret_cast<int64_t&>(double_value);`
  - C-style cast: **(NewType) value**
    - Can do any of the above when you least expect it. Generally avoid.
- Some conversions are **implicit**, i.e. you don't have to explicitly tell the compiler to do the cast.
  - Ex (numeric conversions): `double x = 4;`
  - Ex (adding const): `const int& y = non_const_int_ref;`
  - When implicit casting can happen can be controlled (advanced topic).

# Auto and decltype

- Some times you don't really care what the type of a variable is (but it still has one!) or the type name is too long to type out all the time. The compiler can **deduce** the type with **auto**:

```
auto x = 3; // x is an 'int' b/c integer literals are ints
auto y = crazy_long_return_type();
const auto& z = y; // can be references too
```

- You can also get the type of a variable or expression and use it (to declare another variable for example) with the **decltype** keyword:

```
auto foo = ...;
decltype(foo) bar;
decltype(array_of_somethings[4]) baz;
```



# Flow control: if

- Basic one-liner:

```
if (condition) statement;
```

- If with block scope:

```
if (condition)
{
    statement(s);
}
```

- If else

```
if (condition)
{
    statement(s);
}
else
{
    statement(s);
}
```

- Chaining

```
if (condition1)
{
    statement(s);
}
else if (condition2)
{
    statement(s);
}
else
{
    statement(s);
}
```

- Prefer {} for safety:

```
if (condition)
    statement;
    statement; // oops!
```

# Flow control: for

- Traditional for loop:

```
for (setup; check; increment)
{
    statement(s);
}
```

- **setup** is a variable declaration.
- **check** is a boolean expression which will exit the loop when false.
- **increment** is executed after each loop iteration.

```
std::vector<double> x(10);

for (int i = 0; i < x.size(); i++)
{
    std::cout << x[i] << '\n';
}
```

- Range-based for loop:

```
for (value : container)
{
    statement(s);
}
```

- **value** is a variable declaration.
- **container** is something that can be iterated over.

```
std::vector<double> x(10);

for (double& y : x)
{
    std::cout << y << '\n';
}
```

# Flow control: while and do while

- While loop:

```
while (condition)
{
    statement(s);
}
```

- For any loop, **continue** goes to next iteration while **break** stops the loop.

```
while (condition)
{
    if (skip) continue;
    if (done) break;
    std::cout << "Still going!\n";
}
```

- Do while loop:

```
do
{
    statement(s);
}
while (condition);
```

- Example:

```
bool done;
do
{
    done = am_i_done_yet();
}
while (!done);
```

# Flow control: switch

- Switch selects something to do based on the value of an *integral* variable:

```
switch (value)
{
    case 0:
        statement(s);
        break;
    case 4:
        statement(s);
        break;
    case 3:
        statement(s);
        break;
    default:
        statement(s);
};
```

**default** gets executed if no other case matches.

```
switch (value)
{
    case 3:
    {
        int x = 4;
        statement(s);
    }
    // x not visible here
    // fall through
    statement(s);
    case 45:
        statement(s);
};
```

Switch cases don't have their own scope unless you put in your own {}.

If there is no **break**, execution **falls through** to the next case.

# Functions

- Functions take zero or more **arguments** and return either exactly zero (void) or one **return values**.

```
void foo(int arg1, double arg2)
{
    if (return_early)
        return;

    //do some stuff
}
```

```
void(*ptr_to_foo)(int,double) = &foo;
//easier: auto ptr_to_foo = &foo;
```

```
ptr_to_foo(4, 1.0);
```

You can have pointers to functions and call them. The types get pretty complex, so auto is helpful.

Functions can return at any time and from multiple points (of course just one is reached each time).

Trailing arguments can have default values.

```
double bar(long num = 0)
{
    if (num == 0) return 1;
    return 2*bar(num-1);
}
```

```
// or:
auto bar(long num = 0)
    -> double
{
    ...;
}
```

Trailing return type syntax is sometimes necessary.

# Function overloading

- Multiple functions can have the same name as long as they have different **function signatures** (name + types of arguments, but not return type!):

```
int foo(int x) { ...; }  
long foo(int x) { ...; } // not OK  
long foo(double x, int y) { ...; } // OK
```

```
foo(4); // calls the first one  
foo(1.0, 2); // calls the third one
```

- The compiler picks the function to call through **overload resolution**:
  - Find all **visible** functions with the same name (the overload candidates). This involves **name lookup** which can be quite complex.
  - Select viable functions (mainly means argument types match)
  - Rank the candidates by some complicated criteria
  - If exactly one candidate is the “best”, call it. Otherwise it is an error.

# Namespaces

- **Namespaces** form a scope for global names. They can be used to compartmentalize code and resolve ambiguous naming.

```
namespace foo
{
void bar(int) { ...; }
}

void bar(int) { ...; }

int main()
{
    bar(1); // calls the second one
    foo::bar(3); // calls the first one through a qualified access
    using namespace bar; // like import <blah> in Python
    bar(6); // calls the first one (why?)
}
```

- Namespaces can be nested arbitrarily: `foo::bar::baz::quux()`.

# Classes and structs

- **Classes and structs** combine data (**member variables**) and functions (**member methods**) in one package.
- An object of class type is an **instance**. The class itself doesn't have any data (except for static members).

```
// this defines the class type
struct some_data
{
    int num;
    double* data;
    static int max_size;

    void set_size(int n) ...
    int get_size() { return this->num; }
    static int get_max_size() ...

    double* get_data() { return data; }
};

int some_data::max_size = 1000000;

some_data x;
// call a member method
x.set_size(some_data::get_max_size());
```

Methods and member variables have to be accessed through an instance using the “.” (dot) operator.

Members can also be accessed with the **this pointer**. Accesses to members of pointers to classes use the “->” operator.

Class and struct definitions have to end with a **semicolon**.

**static variables** and methods exist without an instance. They can be used with a qualified access or through an instance.



# Constructors and destructors

- When a class instance is created, a special function called a **constructor** is called. Constructors are defined as overloaded methods with no return type and the name of the class.

```
class foo
{
    // the "default constructor"
    foo() ...

    // the "copy constructor"
    foo(const foo& other) ...

    // a custom constructor
    foo(int x, int y) ...
}
```

```
{
    foo x;

    foo y(x);

    foo z(1, 5);
}
```

- Similarly when an instance goes out of scope or is **deleted** the **destructor** is called.

```
~foo() ...

};
```

```
// z.~foo()
// y.~foo()
// x.~foo()
}
```

# Constructor initializer lists

- The member variables can (and should) be **initialized** (why is this different that **assigned**?) in a constructor **initializer list**:

```
struct point3d  
{
```

```
    double x, y;  
    double z = 0;
```

Initializers should be in the same order as the variables are declared.

```
    point3d() : x(0), y(0), z(0) {}
```

You don't have to use this-> when member variables are shadowed.

```
    point3d(double x, double y, double z)  
    : x(x), y(y), z(z) {}
```

```
    // note we could also do this for both  
    // point3d(double x=0, double y=0, double z=0)  
    // : x(x), y(y), z(z) {}
```

```
    ...
```

```
};
```

All of the member variables should initialize, or given **default values**. (Q: what happens when a member is not initialized?)

# Method definitions

- Methods can either be defined inside the class definition (an **inline** definition), or outside the class:

```
class foo
{
    void method1()
    {
        ...
    }

    void method2(int) const;
};

void foo::method2(int x) const
{
    ...
}
```

Inline definitions are usually reserved for short methods.

The part in the class is a **prototype** (more on these later). Argument names are optional.

Methods can also be **const qualified**. Non-const methods can't be called on a const instance.

The external definition has to be fully qualified with the class name. Qualifiers are repeated (except for static).

Method definitions can be in an entirely different translation unit (file).

# Operator overloading

- Classes can contain special methods called **operator overloads**. The most important operator overload is **operator=** (the **assignment operator**):

```
class operations
{
    // the most common signature for operator=
    // aka the "copy assignment" operator
    operations& operator=(const operations&);

    // the "move assignment" operator
    operations& operator=(operations&&);

    // the "function call" operator (any number of arguments)
    double operator()(int x, double x);

    // the indexing operator (one argument)
    int& operator[](long index);

    // math operators
    operations& operator+=(const operations&);
    operations operator+(const operations&);
    // etc.
};
```

The compiler will generate **operator=** (copy and move) for you in most cases.

The **move assignment** operator takes an rvalue reference. Since rvalues represent things like temporaries, we can steal the other object's resources. But, we have to leave it in a **well-defined state**. You can also have a **move constructor**.

# The Rule of 5

- The “Big 5” include:
  - The **copy constructor**
  - The **move constructor**
  - The **destructor**
  - The **copy assignment operator**
  - The **move assignment operator**
- Normally, the compiler writes default versions of all five of these for you.
- If you have to write a custom version of any one of these, then ***write a corresponding version of all five.***
- Generally this is only necessary if the class manages some resources, e.g. through new/delete or by opening files.

# Public, protected, private

- Class and struct members have three access levels: **public** (anyone can access), **protected** (only derived classes and friends can access), and **private** (only this class can access).
- The **only** difference between a **class** and a **struct** is that class defaults to private and struct defaults to public.

```
class foo
{
    public:
        int get_value()
        {
            // OK here
            do_some_stuff_first();
            // and here
            return my_value;
        }

        protected:
            void do_some_stuff_first() ...

            int my_value;
};

foo instance;

// this is an error
instance.my_value = 4;

// this too
instance.do_some_stuff_first();

// but this is OK
instance.get_value();
```

# Inheritance

- A class can have zero or more **base classes**. The class **inherits** all of the base classes' member variables and methods. This class is now a **derived class** of its bases.
- **static\_cast<...>** can convert from base to derived and vice versa.

```
struct A
{
    int x;
};
```

```
struct B
{
    void foo();
};
```

```
class C : public A, private B
{
public:
    int meaning_of_life()
    {
        return 42;
    }
};
```

Inheritance can be public, protected, or private. This limits the access level of the base classes members, but does not increase it. The default for **class** is private and for **struct** is public.

Derived class members can **shadow** base class members. You can get the base class member with a funky syntax:

```
Derived x;    x.Base::member();
```

```
C instance;
```

```
// OK, this is public
instance.x = 4;
```

```
// error, foo is visible but private
instance.foo();
```

```
// we can access C's own members too
C.meaning_of_life();
```

# Header files

- **Q:** For big projects with lots of files, how does a function/class in file A use a function/class in file B?
- **A:** file A **#includes** a **header file** (usually ends in .hpp, .hxx, .hh, or .h) describing the contents of file B.
- Header files are just plain C++ code. They are **textually substituted** by **#include**. But, they generally contain certain types of constructs:
  - **Prototypes:** this is the declaration of a function without its definition:

```
void proto_func(int x);
```
  - **Class declarations:** a class, but with prototypes for most of its methods.
  - **Forward declarations:** this is like a prototype for a class.
  - **Typedefs** and **using** statements
  - **Extern** variables

```
#include "some_other_file.hpp"
// now I can call functions and use types from
// some_other_file.cxx
```

```
#include "my_project.hpp"
// another convention is to have a project- or module-wide header
```



# More about inline and static

- **Inline** methods (default if defined in class) and functions (requires **inline** keyword) can be substituted by the compiler into the calling code:

```
class foo { int x; public: int getx() { return x; } };  
foo instance;  
instance.getx(); // compiler can turn this into instance.x
```

- Functions defined in header files should be **inline**, or else you get multiple definitions.
- **Static** variables and functions (but not methods!) do not exist outside the current translation unit:

```
// I can have a (possibly different) one  
// of these in as many files as I like  
static int helper_function();
```

# Lambda functions

- **Lambda functions** are anonymous functions defined for a special purpose.
- Lambdas can also **capture** variables from the surrounding scope. The lambda stores a reference or copy of the captured variables in a **closure** (essentially a struct defined on-the-fly).

```
// let's say we have a function
// for_each(container, func)
int count = 0;
for_each(array_of_ints,
[&count](int x)
{
    x = count++;
})
// there are 'count' items
```

vs.

```
struct counter
{
    int count = 0;

    void operator()(int x)
    {
        x = count++;
    }
};

counter cntr;
for_each(array_of_ints, cntr);
int count = cntr.count;
```

The lambda argument list (...) can be omitted if it doesn't take any arguments.

# Lambda captures

- Lambdas can capture any variable in the enclosing scope, but none are captured by default.
  - `[]`: default, no capture
  - `[=]`: capture all variables used in the lambda as copies
  - `[&]`: capture all variables used in the lambda by reference
  - `[name]`: capture 'name' as a copy
  - `[&name]`: capture 'name' by reference
  - `[&,but_copy_this]`: mix-n-match
- Variables captured as copies can't be modified unless the lambda is **mutable**:

```
int x = 3;
[=] mutable
{
    x = 2; // would be an error without 'mutable'
}();
```

# Saving lambdas for later

- Lambdas can be saved in a variable for delayed or repeated use. **auto** is very useful for this:

```
auto func = [&](int x, int y) { ... };  
  
func(1, 2);  
do_something_with_this(func);
```

- **Pure lambdas** (with no captures) can be converted to a function pointer:

```
double (*sqr)(double) =  
    [](double x) -> double  
    { return 2*x; };
```


The return type for a lambda can usually be **deduced**. Otherwise, you have to specify with the **trailing return type** syntax.

- You can also save them in the **std::function** type (even with captures):

```
double base = M_E;  
std::function<double(double)> exp =  
    [base](double exponent) { return std::pow(base, exponent); }  
  
exp(-1.0);
```

# Templates

- **Templates** are one of C++'s most important and most complicated features.
- The most common use of templates is to create **generic** code:

<pre>class float_thing... class double_thing... class complex_float_thing... class complex_double_thing... class int_thing... class long_thing... class some_other_thing...</pre>		<pre>template &lt;typename T&gt; class thing;  thing&lt;float&gt; ...; thing&lt;double&gt; ...; // etc.</pre>
---	--	---

# Some simple examples

- Dynamic arrays (using the `std::vector` template):

```
std::vector<double> array;  
array.resize(100);  
for (int i = 0; i < array.size(); i++)  
    array[i] = i*i;
```

- Functions can also be templates:

```
template <typename T>  
std::complex<T> complex_from_mag_angle(T mag, T angle)  
{  
    return std::complex<T>(std::cos(angle)*mag,  
                           std::sin(angle)*mag);  
}
```

# Template specialization

- Sometimes you want a template function or class to do something special for a particular type. This can be accomplished with **template specialization**:

```
// generic implementation
// note that template parameters
// can have default values too
// e.g. foo<> = foo<double>
template <typename T=double>
class foo { ... };

// specialization
template <>
class foo<int> { ... };

// more specializations
// if needed
template <>
class foo<long> { ... };
```

Function templates don't necessarily need the **template parameter list** (<...>) when they are called. When the parameters can be **deduced** from the arguments it can be left off.

```
// it doesn't have to be T...
template <typename MyType>
void process(MyType& item);

// calls process<int>
int x; process(x);

// template functions can
// be specialized too
template <>
void process<SpecialType>
    (SpecialType& item);
```

# Partial specialization

- Class templates (but not functions!) can be **partially specialized**:

```
template <typename Key, typename Value>  
class map { ... };
```

```
// special implementation for when the  
// key is a std::string  
template <typename Value>  
class map<std::string, Value> { ... };
```



# Variadic templates

- Sometimes you need a template, but you don't know how many template parameters there will be. **Variadic templates** allow you to take any number of parameters in a **parameter pack**.
- An expression involving the parameter pack is **expanded** with the “...” operator.

```
template <typename Func, typename... Types>
auto call(Func&& f, Types&&... args)
    // this part is optional in C++14 and later
    -> decltype(f(std::forward<Types>(args) ...))
{
    return f(std::forward<Types>(args) ...);
}
```

- This is an example of **perfect forwarding**, which uses something called a **universal reference** (template + rvalue reference). The details of this are outside our scope.

# A heavy-duty example

- The C++ standard library includes a variadic template type called **std::tuple**, which holds any number of different types of objects:

```
std::tuple<int, double, const foo&> x(...);  
// calls the bar method of  
// the const foo& member of x  
std::get<2>(x).bar();
```

- Suppose we wanted to print out the contents of **any** std::tuple (assuming we know how to print each element)?

# A heavy-duty example

- A loop doesn't work. Template arguments (in this case for `std::get`) have to be **constant expressions**, i.e. the value has to be known at compile time.

```
// this doesn't work
template <typename... Types>
void print(const std::tuple<Types...& t)
{
    for (int i = 0; i < sizeof...(Types); i++)
        print(std::get<i>(t));
}
```

Expand the parameter pack  
*inside* `std::tuple`'s template  
parameter list.

`sizeof...` gives the number of  
parameters in the pack.

# A heavy-duty example

- Instead, let's use another template:

```
// thinking ahead: why a struct and not a function?
```

```
template <int I, typename... Types>
```

```
struct tuple_printer
```

```
{
```

```
    tuple_printer(const std::tuple<Types...>& t)
```

```
    {
```

```
        print(std::get<I>(t));
```

```
    }
```

```
};
```

```
template <typename... Types>
```

```
void print(const std::tuple<Types...>& t)
```

```
{
```

```
    // But what to do here? We still can't do a loop
```

```
}
```

# A heavy-duty example

- We can also make our template **recursive**:

**// But what happens when we get to the end?**

```
template <int I, typename... Types>
struct tuple_printer
{
    tuple_printer(const std::tuple<Types...>& t)
    {
        print(std::get<I>(t));
        tuple_printer<I+1, Types...>{t};
    }
};
```

Bonus question:  
Why {t} not (t)?

```
template <typename... Types>
void print(const std::tuple<Types...>& t)
{
    // kick off the recursion
    tuple_printer<0, Types...>{t};
}
```

# A heavy-duty example

- **Template specialization** comes to the rescue.

```
template <int N, int I, typename... Types>
struct tuple_printer
{
    tuple_printer(const std::tuple<Types...>& t)
    {
        print(std::get<I>(t));
        tuple_printer<N, I+1, Types...>{t};
    }
};
```

```
template <int N, typename... Types>
struct tuple_printer<N, N, Types...>
{
    tuple_printer(const std::tuple<Types...>&) { //do nothing }
};
```

```
template <typename... Types>
void print(const std::tuple<Types...>& t)
{
    tuple_printer<sizeof...(Types), 0, Types...>{t};
}
```

Bonus question:  
Why no name for the  
function argument?

# Templates on steroids: metaprogramming

- We did a little of this in our last example. But here are some other common fancy template tricks:
  - Picking overloads based on template parameters:

```
// version for float, double
// yikes this is ugly!
template <typename T>
typename std::enable_if<
    is_floating_point<T>::value,
    T>::type
random_number();
```

```
// version for integers
// yikes this is ugly!
template <typename T>
typename std::enable_if<
    is_integral<T>::value,
    T>::type
random_number();
```

- “Smart” partial specializations:

```
// default to putting
// data on the stack
template <typename T, size_t N,
        typename=void>
struct static_storage
{
    T data[N];
    ...
};
```

```
// but for large N use
// dynamic storage
template <typename T, size_t N>
struct static_storage<T, N,
        typename std::enable_if<
            (N>1000000)>::type>
{
    std::vector<T> data;
    ...
};
```

# Friendlier templates: the STL

- The **Standard Template Library** (STL) contains a lot of useful classes and functions accessible through various header files:
  - Algorithms
  - Container types
  - Time functions
  - I/O functions
  - Regular expressions
  - Threading
  - Atomic operations
  - Utilities and type traits (like `std::enable_if`)
  - And more...



# STL containers

- **std::vector**: Linear, dynamically allocated array. Accessing elements is fast but putting in new elements (especially in the middle) can be fairly slow.
- **std::list**: Linked list of elements. Getting the  $n^{\text{th}}$  element is slow, but inserting is constant time.
- **std::map**: Tree-based map from one set of things to another. **std::unordered\_map** uses a hash table.
- **std::set**: This is a set, duh.
- More: **std::queue**, **std::deque**, **std::stack**, etc.

# STL algorithms

- STL algorithms act on **iterators**. You get iterators with **container.begin()** and **container.end()**:

- Sorting:

```
std::sort(data.begin(), data.end());
```

- Fill/copy:

```
// make sure v2 has enough room!
```

```
std::copy(v1.begin(), v1.end(), v2.begin());
```

```
std::fill_n(v1.begin(), v1.size(), 0);
```

- And way too many more to go over here.

# STL I/O: iostreams

- C++ can use “old-style” **printf**, but it also has a new **stream**-based interface:

```
// write to standard output
std::cout << "Hi there!\n";

// a custom stream operator
std::ostream& operator<<(std::ostream& os, const some_type& x)
{
    // print x to the stream os
    return os;
}

some_type x(...); std::cout << x << '\n';

// basic file I/O
std::ofstream ofs("output_file");
ofs << some << data << std::endl; // std::endl = '\n' + flush
// ofs will be closed automatically in the destructor
```

# STL smart pointers

- Some times local objects and references are not enough, and you really need something like **new** and **delete**. The STL provides **smart pointers** for this that add some safety:
  - **std::unique\_ptr<T>**: like **T\***, but the object will be automatically deleted when the **unique\_ptr** goes out of scope. You can “give” the object to someone else using the move constructor or move assignment operator (using **std::move**).
  - **std::shared\_ptr<T>**: like **unique\_ptr**, except that it can be copied and assigned like normal. It is **reference-counted**, so it only gets destroyed when all references to it go out of scope.
  - You can conveniently create these with **std::make\_shared** and **std::make\_unique** (C++14).

# Alternate standard libraries

- The STL has a lot of stuff but not nearly everything you might want. Several popular additional libraries exist:
  - **Boost**: Boost is perhaps the single most well-known set of C++ libraries. Many Boost innovations make their way in to the official standard.
  - **Bloomberg Development Environment** (BDE).
  - **Folly**: Facebook's C++ libraries.
  - **Qt** and **KDE** C++ libraries include more than just GUI components.
  - And many more...

# Some additional resources

- <https://isocpp.org/get-started>
- <http://www.learncpp.com>
- <http://www.cplusplus.com/doc/tutorial>
- <http://en.cppreference.com>  
***THE*** definitive reference on the C++  
standard library; I use it almost every day.

# Things to Google (in no particular order)

- One Definition Rule
- RVO/NRVO
- Copy elision
- Why not to use `_name` or `__name?`
- `nullptr` vs. `NULL`
- `explicit` keyword
- `mutable` keyword
- `volatile` keyword
- `extern` keyword
- Explicit template instantiation
- Extern template declaration
- Why are C-style casts bad?
- Why not to use “using namespace” in header files?
- Include guards
- Narrowing conversions
- Conversion operators
- Pass-by-value vs. pass-by-reference
- `++i` vs. `i++`
- ADL
- CRTP
- RAII
- Pimpl
- Mixins
- Virtual inheritance
- Diamond inheritance
- Virtual functions
- Dynamic casts
- Member pointers
- Template template parameters
- Generic lambdas
- Fold expressions
- Structured bindings
- `constexpr` keyword
- `constexpr` if
- Generalized `constexpr`
- Variable templates
- Template aliases
- Static initialization order fiasco
- Copy and swap
- Construct on first use
- Empty base optimization
- Unions
- Variants
- Short string optimization
- `final` keyword
- `override` keyword
- Inner classes
- Enumerations
- Strongly-typed enumerations
- Macros (use with care)
- Variadic macros
- Exceptions
- `noexcept` keyword
- Safe bool
- SFINAE
- Type erasure
- Virtual destructor
- Virtual constructor
- Friend functions
- Friend classes
- Template friends
- RTTI
- Variadic functions (do not use)
- User-defined literals
- Visitor pattern
- `__restrict__`
- Delegating constructors
- Inheriting constructors
- `std::any`
- Inline variables
- Type traits
- Member detection
- Defaulted constructors
- Defaulted assignment operators
- Deleted constructors
- Deleted assignment operators
- Placement new
- Iterator classes
- Allocators

Questions?