Functions

PIC 10A, UCLA

Functions

Functions allow for useful sequences of commands and procedures to be done in a clean and tidy way. We use functions to read from a stream into a string with **getline** and to generate random numbers with **rand**. Being able to write functions is also important because they can package useful procedures together that can be called upon multiple times in a program, without having to write identical, lengthy blocks of code, many times in a program.

Using Functions

Given its documentation, we know what a function does, what arguments it requires, whether it outputs a value, and what type that value is. Recall that a function that returns **void** has no return value.

```
/**
The factorial function will compute the factorial of a nonnegative integer.

@param n the integer to compute the factorial for

@return the factorial
*/
unsigned long long factorial ( unsigned long long n );
```

A function should be documented thoroughly, as above, in its declaration.

Using Functions

Seeing the documentation of the **factorial** function, we could use it appropriately

```
unsigned long long int i = 12; // obtain some integer value
int iFactorial = factorial( i ); // and compute its factorial
```

Using Functions

Being able to use the **factorial** function given is significantly more convenient than, for example,

```
unsigned long long int iFactorial = 1;

for ( unsigned long long int j = 1; j <= i; ++j ) { // for j from 1 to i
   iFactorial *= j; // multiply product by j
}
```

Imagine having to compute the factorial for many numbers in the program...

Using Functions

The purpose of a function is to pass relevant data to another part of the program for processing. This other part of the program works with variables that are necessary for the execution of the program and when the instructions are completed, control of the program returns to where it left off.

```
// pass i off to process and set iFactorial
unsigned long long iFactorial = factorial( i );

// and now print the factorial
cout << iFactorial;
```

Declarations and Definitions

A function is **declared** with its **signature**. The signature may or may not name the input arguments. The following are all declarations of a function:

```
double sin(double); // did not name the argument

double pow(double base, double exponent); // named both arguments

double max(double first, double); // did not name 2nd argument

void foo();
```

Recall: the compiler requires the declarations of symbols before they can be used; the linker requires the definitions.

Declarations and Definitions

A function **definition** takes place when the function is given a **body** (a set of braces).

Within the body, we specify what set of commands to execute.

```
// this DECLARES trivialFunction: there is no body
void trivialFunction();

// this DEFINES trivialFunction: there is a body (but it does nothing)
void trivialFunction() { }
```

Remark: actually, **trivialFunction** defined above is also a **declaration** because the signature is there!

Declarations and Definitions

The names of the function **arguments** (the names given in the definition) define **local variables**, variables that are only defined within the scope of the function. The variables that we pass, from left to right, are assigned to the local variables, from left to right.

```
// will print whatever int input it is given
void printInt( int i ) {
    cout << i;
}
```

Above, the **printInt** function has been defined. In **main**, we could write

```
int j = 430;
printInt( j );
```

Declarations and Definitions

Expanding out the call **printInt(j)**, we effectively have:

```
int j = 430;

{ // local scope
    int i = j; // local variable i is defined and initialized by j
    cout << i;
} // now i no longer exists
```

Declarations and Definitions

A function such as

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

When called by

```
foo(3, 4, 5);
```

will create local variables, **x** with initial value **3**, **y** with initial value **4**, and **z** with initial value **5**.

After **foo** has been called, **x**, **y**, and **z** no longer exist.

Declarations and Definitions

Within the function's body, we can refer only to these local variables (and global variables) but we have no knowledge of any other variables.

The function below would not work:

```
void bar( int x ) {
    cout << y; // ERROR: y is not defined here
}
```

even when called within main as

```
int y = 11;
bar(13);
```

because **y** is unknown within the local scope of **bar**.

Declarations and Definitions

For functions that return a value, there must be a **return statement** that can be reached during its execution, indicating what to return.

Once a return statement has been reached, the function terminates, even if there are more lines of code after the return statement!

The general syntax for defining a function is as follows:

returnType functionName (comma separated list of named arguments) {

instructions to execute;

return something, or not ... ;

}

Declarations and Definitions

```
// This function will always return the int value 10
int always10() {

    /* we tell the compiler to return the value 10: the function ends here
       and 10 is returned */
    return 10;

    // this line is ignored: a return statement has already been reached
    return 11;
}
```

Declarations and Definitions

We define a **round** function below:

```
int round(double toRound) {

    if ( toRound >=0 ) {
        // if the number is positive, we add 0.5 and cast to round
        return static_cast<int> ( toRound + 0.5 );
    }
    else { // otherwise, we subtract 0.5 and cast to round
        return static_cast<int> ( toRound - 0.5 );
    }
}
```

Declarations and Definitions

If **value** is a **double** that we wish to round, the line of code below:

```
int rounded = round ( value );
```

roughly expands to:

```
int rounded; // not initialized
{
   double toRound = value; // local variable
   int returnValue; // value to return
   if ( toRound >=0 ) { // if positive, add 0.5 and cast
      returnValue = static_cast<int> ( toRound + 0.5 );
   }
   else { // otherwise, we subtract 0.5 and cast to round
      returnValue = static_cast<int> ( toRound - 0.5 );
   }
   rounded = returnValue; // initialize rounded from returnValue
}
```

Declarations and Definitions

Given a function with signature

returnType foo(type1 x, type2 y, type3 z, ...);

If it is called by

foo(a, b, c, ...);

Then within the local scope of the function, this amounts to creating variables by:

returnType ret;
type1 x = a;
type2 y = b;
type3 z = c;
...

Declarations and Definitions

A function returning **void** does not require a **return** statement, but sometimes they are useful. When we want the function to terminate, we simply write

```
return;
```

This allows the function to terminate before its entire body has been executed.

Declarations and Definitions

```
void displayQuotient( double a, double b) {
    if ( b == 0 ) { // if we would divide by 0
        return; // do not do it
    }

    // no division by 0, so do the computation
    cout << a/b;

    // a final return statement here is optional
    return;
}
```

Declarations and Definitions

A function can be **declared** multiple times, but the **definition** can only appear once in the program (for most functions...)

Often functions can be declared (and/or defined) above main and then defined below main (if not defined above).

In other cases, functions can be declared in header files and then defined in **.cpp** files.

Declarations and Definitions

Header file approach: declared in header, defined in a **.cpp** file, and used within **main.cpp**: this approach involves working with multiple files.

Foo.h

```
#ifndef __FOO__
#define __FOO__
void foo();
#endif
```

Foo.cpp

```
foo() { }
```

main.cpp

```
#include "Foo.h"
int main() {
    foo();
    return 0;
}
```

Declarations and Definitions

Declared above **main** and defined after **main**: this approach can make the main cpp file very long if the function(s) defined are long.

main.cpp

```
void foo();

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

void foo() { }
```

Declarations and Definitions

Declared and defined above **main**: a definition is also a declaration. This approach can get messy for long functions appearing above main.

main.cpp

```
void foo() { }

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

Declarations and Definitions

It is an error to reference a function that has not yet been declared.

main.cpp

```
int main() {
    foo(); // ERROR: what is foo?
    return 0;
}

void foo(); // not good enough to declare afterwards!
```

References

Consider the function

```
void increment_version1(int i) {
    ++i;
}
```

Given that we know a local **int** called **i** is constructed when we pass the function an argument, the code and resulting output below are not surprising:

```
int x = 300;
increment_version1(x);
cout << x;
```

```
300
```

x is unchanged because we made a copy of **x** in the variable **i**...

References

On the other hand, if we wanted to change **x**, it might not be surprising that we need to define a function

```
void increment_version2( int & i ) { // make a reference to input
    ++i; // and increase the outside variable
}
```

Then, the code below generates the output also given below:

```
int x = 300;
increment_version2(x);
cout << x;
```

```
301
```

x is changed because **i** is a reference to **x**, which was incremented.

References

Just consider:

```
int x = 300;

{
    int &i = x;
    ++i;
}

cout << x;
```

References

The only way that we can modify a variable that we pass into a function through that function directly is by **“passing the variable by reference,”** i.e. making a local variable a reference to the input variable!

If we do not pass a variable by reference to a function, we are **“passing by value”**, and we are making a copy of that variable within the function.

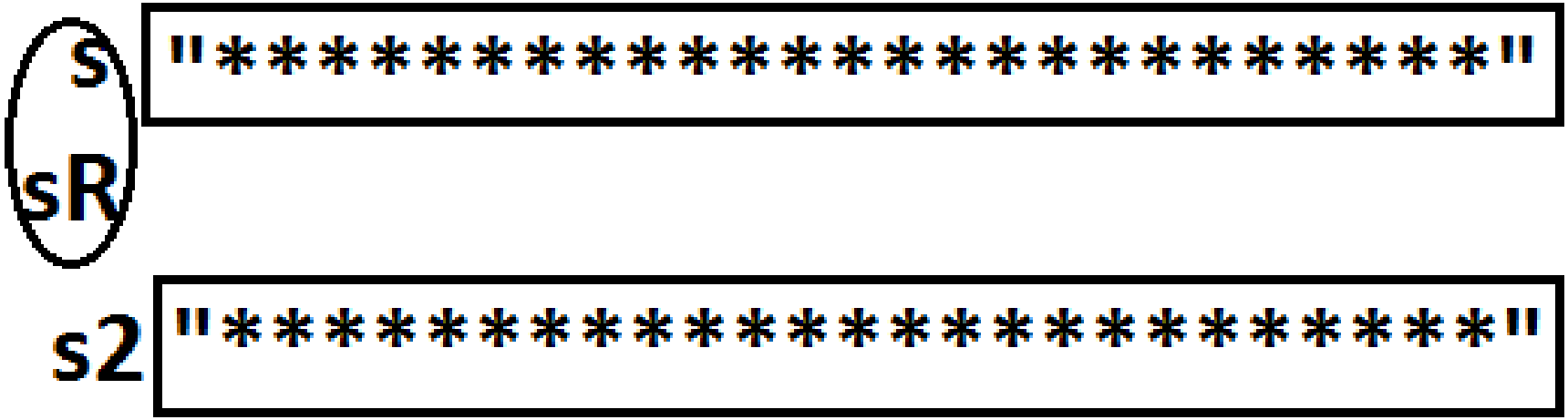
References

There is another aspect to references to consider: it is **more efficient to pass a class object by reference than by value.**

```
string s( 100000, '*'); // s has 100 thousands *'s

// sR is NOT a copy. It references s and only takes up  4 or 8 bytes
string & sR = s;

/* s2 is an independent COPY of s: this copying is expensive
   because s is large */
string s2 = s;
```



References

A class object (**std::string**, **std::vector**, etc.,) should always be passed to a function by reference! It is otherwise inefficient.

Fundamental types like **char**, **bool**, **int**, **long double**, etc. can safely be passed by value. Their size and the size of the reference variables are about the same.

Const Correctness

To protect against bugs in the code, **if a local reference variable within a function does not change, it should be listed as const**.

It can also be useful, for long functions, to make local copies constant if they do not change. But it is absolutely essential that all reference variables that are not to be modified are made const!

Const Correctness

This would be bad...

```
void foo( string& message) {
    string message2;

    if ( true ) { // if true, set message2 to "DESTROY"
        message = "DESTROY"; // oops, change wrong variable!
    }
}

void bar() {
    string s;
    foo(s);

    if ( s == "DESTROY") { // guess we destroy the planet...
        obliteratePlanet():
    }
}
```

Const Correctness

These conventions can be easy to forget, but they are essential to writing safe and efficient code that interfaces with other people's code.

```
/* This function does not modify its input and should follow const
correctness. It will cause problems for foo, which does follow const
correctness */
void bar( string& input ) { // should be const string&
    cout << input;
}

// This function IS const correct
void foo( const string& input ) {
    bar(input); // DOES NOT COMPILE BECAUSE bar is not const correct!
}
```

Const Correctness

A function that accepts a reference cannot be called upon a const object!

The compiler assumes any time we do not define a variable as const that we intend to modify it at some point.

Const Correctness

A class object should **always be passed by reference** (exceptions to this are beyond the scope of this discussion).

If the class object should not change within the function, it should be **passed as a reference to const**.

```
void changeTheString ( string& in );
void doNotChangeTheString( const string& in );
```

Const Correctness

Unless a fundamental type should be changed, a fundamental type can be passed by value.

If the **fundamental type must change, it must be passed by reference**.

If the fundamental type should not change within the function and it was passed by value, it can also be listed as constant.

```
void localCopyConstant ( const double );
void localCopyCanChange ( double );
void modifiesInputArgument ( double & );
```

Const Correctness

Remark: A function that accepts a **const string&** input can be called upon a string literal, too!

```
void foo( const string& in ) { }
```

It is okay to write

```
foo("Hello");
```

because internally, a temporary **std::string** object (an r-value) is constructed, which the **const string&** can bind to. It would not be possible to have a scenario such as

void bar (string & in) { }

with a call

```
bar("Hello"); // ERROR: cannot bind L-value reference to R-value
```

Overloading

A function name can be **overloaded**; this means that there can be multiple functions with the same name, provided they can be distinguished by their input arguments.

Two functions with the same name, but different arguments, are seen as different functions. For example, we could have

```
void foo();
void foo(const int);
double foo(const double);
string foo(const char, const size_t);
```

Above, these are all different functions.

Overloading

Two functions cannot be differentiated based on return type alone.

```
void bar(); // PROBLEM: this and
int bar(); // this share the same void input

// PROBLEM: both functions below share the same inputs
double baz(int, int, double);
vector<int> baz(int, int, double);
```

Overloading I

Consider the definitions below:

```
void foo() {
    cout << "foo"<< endl;
}

void foo(const int i) {
    cout << i << endl;
}

void foo( const double d) {
    cout << d << endl;
}

string foo(const char c, const size_t s) {
    return string(s, c);
}
```

Overloading II

Then in calling

```
foo(); // invokes void foo()
foo(14); // invokes void foo(const int)
foo(14.); // invokes void foo(const double)
cout << foo(':', 6); // invokes string foo(const char, const size_t)
```

we arrive at the output:

```
foo
14
14
::::::
```

Overloading

When a function is called with a set of arguments, a list of **candidate functions** is generated: this is a list of all functions with the same number of input arguments as provided and such that there exist **conversions** (also called **coercions**) between the input arguments supplied and those of the function; for examples:

- **int** can be promoted to **double**
- **float** can be converted to **const float**
- **long double** can be cast to **int**
- **const char*** (string literal) can be converted to **std::string**
- **bool []** (array of **bool**) can be converted to **bool ***

Ideally, there would be no conversions, but that is seldom the case.

Promotions and Coercions

The compiler tries to find the best match for a given input and it may at times make a conversion. Consider

```
void moo(int);
void moo(int, double);
void moo(const string&);
```

with the following calls:

```
moo(43.8); // converts 43.8 (double) to 43 (int), calls moo(int)
moo(11, 11); // promotes 11 (int) to 11. (double), calls moo(int, double)
moo("hey"); // converts "hey" to const string&, calls moo(const string&)
```

When there is not an exact match, provided the arguments can be converted between appropriate types, these conversions are done.

Promotions and Coercions

The compiler ultimately chooses the **best** match: this is the candidate function that requires the fewest number of conversions, among other things...

If there is no best match or the conversions cannot be done, a function **call is in error**. Consider

```
baah(int, int, double);
baah(int, double, double);
```

with the following calls:

```
/* will invoke baah(int, int, double): only requires one conversion 9 (int) to
   9. (double) */
baah(7, 8, 9);

/* will invoke baah(int, int, double): only requires two conversions
   6.3 (double) to 6 (int) and 1 (int) to 1.0 (double) */
baah(6.3, 0, 1);
```

Promotions and Coercions

Now consider

```
void baah(const int, float);
void baah(const int, long long int );
void baah(int);
```

with

```
/* ERROR: There is no best match. Must either convert 1.1 (double)
   to float or to long long int */
baah(1, 1.1);

/* ERROR: There is no conversion between string literal and int */
baah("hello");
```

Functions with Default Arguments

Function arguments are set from left to right.

It is possible to specify default arguments of a function from right to left such that, should fewer arguments be given than required inputs, the compiler will fill in those arguments with the specified default values.

To provide a default argument value, instead of just listing the type and variable name, we add **= someValue**.

```
// local variables j and k have default values of -1 and 22
void foo( int i, int j = -1, int k = 22 );
```

Functions with Default Arguments

```
void foo( int i, int j = -1, int k = 22 );
```

In calling functions with default arguments, these default values can be used if the function is called with fewer arguments.

```
foo(3, 4, 5); // i==3, j==4, k==5
foo(3, 4); // i == 3, j==4, k==22
foo(3); // i==3, j==-1, k==22
```

Functions with Default Arguments

The default values for arguments must be given from right to left.

```
// okay
void xyzzy( long double d, unsigned u, char c = '*');

// ERROR: only gave middle a default but not right
void quuz( long double d, unsigned u = 343, char c);
```

Functions with Default Arguments

The default value for a local variable can be specified only once in all its declarations and its definition. It is *often best to specify the values in the initial declaration.*

```
// okay, definition only provides default value
void bar( double d );
void bar ( double d = 3.14 ) { }

// okay, declaration only provides default value
void baz ( int i, char c = '%');
void baz ( int i, char c ) { }

// ERROR: default value specified twice
void waldo ( int i, double x = 4.44 );
void waldo ( int i, double x = 4.44 ) { }
```

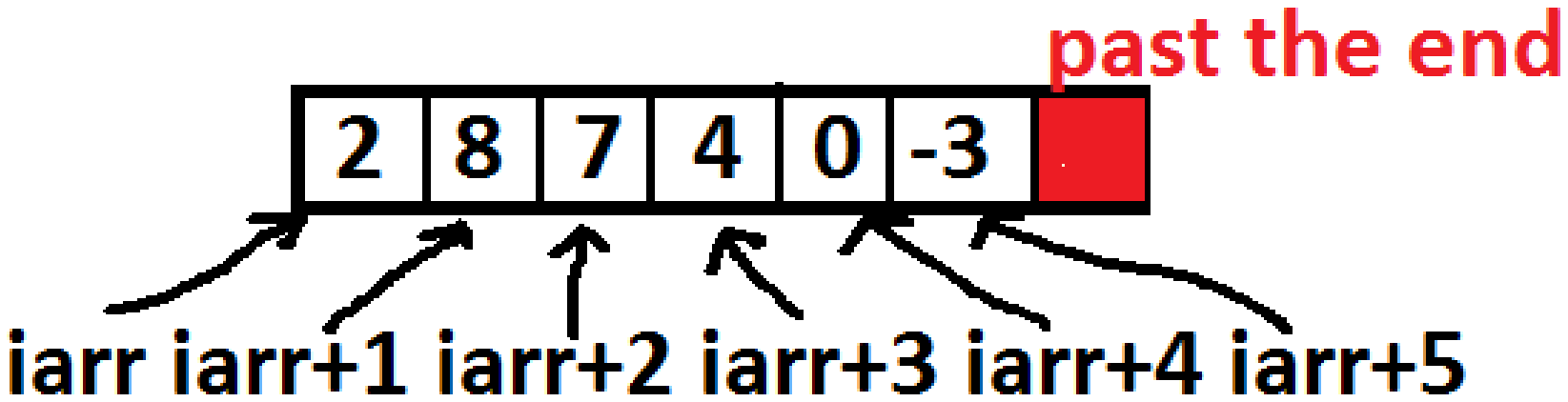
Pointer Arithmetic

Recall that when an array is created, the array name is itself a pointer to the first element.

```
int iarr[] = { 2, 8, 7, 4 , 0, -3 };
```

It turns out, we can use **iarr** as a pointer and perform **pointer arithmetic**.

```
iarr; // points to iarr[0], 2
iarr+1; // points to iarr[1], 8
iarr+2; // points to iarr[2], 7
// etc.
```



Pointer Arithmetic I

In an intuitive manner, we can increment/decrement pointers: incrementing a pointer advances it to the next element and decrementing a pointer makes it point to the previous element.

```
// iarr == { 2, 8, 7, 4, 0, -3 }

int *x = iarr; // x points to iarr[0], 2

cout << *x << endl;
++x; // x now points to iarr[1]
cout << *x << endl;
```

```
2
8
```

Pointer Arithmetic II

```
// iarr == { 2, 8, 7, 4, 0, -3 }

int *y = x + 4; // y points to iarr[4], 0

cout << *y << endl;
- -y; // y now points to iarr[3], 4
cout << *y << endl;
```

```
0
4
```

Pointer Arithmetic

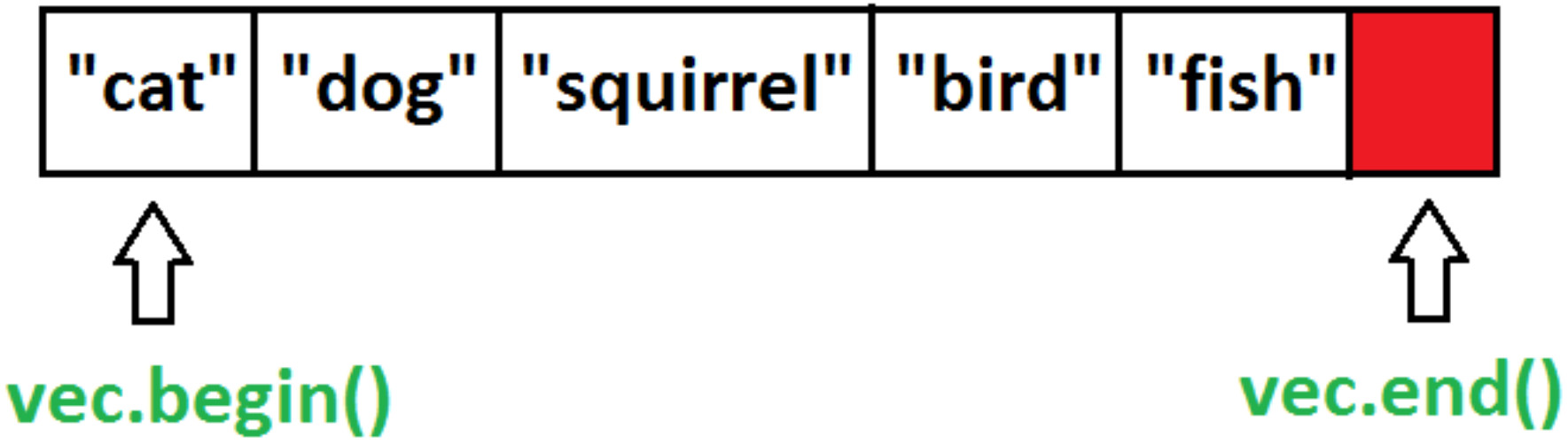
Although arrays lack member functions, we can use **std::begin** and **std::end**, defined in the **<iterator>** header to obtain pointers to the first element and just-past-the-end element of an array.

```
int *begin = std::begin(iarr); // begin == iarr
int *end = std::end(iarr); // end == iarr+6 (past the end!)
```

Iterators

More generally than pointers, we can use **iterators** for containers. These are souped up versions of pointers. All containers of the C++ Standard have iterators.

For now, we will restrict ourselves to iterators for vectors. A **std::vector** has **begin** and **end** member functions that return iterators that point to the first element and just past the last element, respectively.



Iterators

The proper type name of an iterator can be rather atrocious to write...

```
vector<string> vec { "cat", "dog", "squirrel", "bird", "fish" };

// itBegin is an iterator referencing "cat"
vector<string>::iterator itBegin = vec.begin();

// we can also use the begin function
// vector<string>::iterator itBegin = begin( vec );
```

Notice how long it is to write, especially if we include the **std::**...
std::vector<std::string>::iterator

Iterators

An **iterator** is effectively a well-behaved pointer. For example, **iterator**s can point to a location in memory, be incremented/decremented, observe pointer arithmetic in adding/subtracting from them, etc.

Iterators have similar syntax to pointers, with **++** and **- -** to increment/decrement, ***** to dereference, **->** to access a class member, etc.

```
// recall vec == { "cat", "dog", "squirrel", "bird", "fish" }
// itBegin points to "cat"

cout << *itBegin << endl; // print the string
++itBegin; // advance the iterator to "dog"
itBegin += 2; // now advance two more positions to "bird"
cout << itBegin->size(); // obtain size of "bird"
```

```
cat
4
```

Iterators

The **auto** keyword is able to detect the type of the right-hand side of a definition/initialization. It is a great shorthand. We can write:

```
auto iter = vec.begin(); // iter points to the first element of vec

auto iter2 = vec.end(); // iter2 points just past the last element of vec
iter2 -= 1; // now iter2 points to the last element of vec
```

Iterators

Suppose **v** is a **std::vector<int>** with **v == { 1, 2, 3, 4, 5, 6 }** .

With iterators, we can unlock the **insert** and **erase** member functions for an **std::vector**, without having to write all the moves out by hand:

```
// insert the value 10 just before v[2] == 3
v.insert( v.begin() + 2, 10 ); // now v == { 1, 2, 10, 3, 4, 5, 6 }

// erase the value at position v[4]
v.erase( v.begin() + 4 ); // now v == { 1, 2, 10, 4, 5, 6 }
```

Note that **v.begin() + i** points to **v[i]**.

Iterators

In general, to **insert** in a **std::vector**:

// inserts value just before iterator"s position
vectorName.insert(iteratorPosition, value);

And to **erase** in a **std::vector**:

// erases element at iterator"s position
vectorName.erase(iteratorPosition);

Iterators

We can also sort vectors! There is a **std::sort** function that accepts two iterator inputs. This function is defined in the **<algorithm>** header.

The first input specifies the beginning of the sort range; the second input specifies the position *just past* the end of the sorting range.

```
// sort v[0] through v[4]: v[5] and beyond not included!
sort( v.begin(), v.begin() + 5 ); // v == { 1, 2, 3, 4, 10, 5, 6 }

// sort the entire vector: v.end() not included
sort( v.begin(), v.end() ); // v == { 1, 2, 3, 4, 5, 6 , 10 }
```

Passing Arrays and Pointers to Functions

Without some other tricks, we must generally specify the array size as a parameter because the array will **decay into a pointer**.

```
// arr will be a pointer to the array of ints and sz tells us its size
void foo( const int* arr, size_t sz) {
    for (size_t i=0; i < sz; ++i) {
        cout << arr[i] << endl;
    }
}
```

We could call **foo** as

```
const size_t iSize = 3;
int iarr[iSize] = { 3, 4, 5 };

foo( iarr, iSize );
```

Passing Arrays and Pointers to Functions

Remarks: the array decays into **int ***, but to guard against possible modification given that pointers, like references, give direct access to an object, we store the pointer as a **const int ***.

This needing to specify the size for an array is another thing that makes **std::vector**s so nice.

Passing Arrays and Pointers to Functions

The **std::time** function can be documented as follows:

```
/**
This function computes the number of seconds, S, that have elapsed
since 00:00 on January 1, 1970, UTC time.

@param timePointer a pointer to a time_t variable. If the pointer is not
nullptr, the time_t variable that it references will be changed to the value
of S.

@return the value S as described above.
*/
time_t time( time_t *timePointer);
```

Passing Arrays and Pointers to Functions

time_t is a special integer type for use in the **std::time** function counting the seconds.

nullptr is a pointer literal: it points nowhere and can readily be converted to any pointer type.

Passing Arrays and Pointers to Functions I

We can use **std::time** to write a simple timer for doing addition problems.

```
// Obtain number of problems to do
cout << "How many problems? ";
int numProblems = 0;
cin >> numProblems;

// Obtain range of numbers in sums
cout << "How big can the numbers get? ";
int biggest;
cin >> biggest;

// number of problems user answers correctly
int numCorrect = 0;

// set start time to now, end time to 0 initially
time_t start = time ( nullptr ), end = 0;
```

Passing Arrays and Pointers to Functions II

```
// for each problem
for ( int i = 1; i <= numProblems; ++i ) {
// pick two random int values in range and their sum
    int num1 = rand() % biggest + 1; // from 1 to biggest in value
    int num2 = rand() % biggest + 1; // from 1 to biggest in value
    int correct = num1+num2;

    // make user compute sum
    cout << num1 << "+" << num2 << " = ";
    int userAnswer = 0; // user answer
    cin >> userAnswer;
    if ( userAnswer == correct ) { // if correct
        ++numCorrect; // increase correct count
    }

}
```

Passing Arrays and Pointers to Functions III

```
// update end time
time ( &end ); // pass pointer to end to update its value

// compute the total time and give user results
time_t totalTime = end - start;
cout << "In "<< totalTime << " seconds, you answered " <<
    numCorrect << " problems correctly out of "<< numProblems << ".";
```

Range For Loop

There is another loop structure called the **range for** loop, new to the C++ Standard. It makes for very slick and easy to read traversals of containers.

Suppose that **salaries** is a **std::vector<double>** storing the salaries of employees of a company. We can print out all of the salaries with:

```
for ( double salary : salaries ) { // for each salary listed
    cout << salary <<endl; // print its value
}
```

Range For Loop

The general syntax for a **range-based for loop** is:

for (variableType variableName : containerName) {
*   do this stuff;*
}

It means to go through every element of the container, give that element a name so it can be used, use it by that name, and go on to the next element, etc.

Range For Loop

Expanded out,

```
for ( double salary : salaries ) {
    cout << salary << endl;
}
```

means

```
// start at begin but do not include end
for ( auto iter = salaries.begin(); iter != salaries.end(); ++iter) {
    double salary = *iter; // create local variable called salary
    cout << salary << endl; // and print it
}
```

Range For Loop

Now suppose **roster** is a **std::vector<std::string>** storing the names of members of a baseball team. We can print out all of their names with

```
for ( const string& name : roster ) {
    cout << name << endl;
}
```

Note: we use **const string&** for efficiency and const-correctness. We do not want to make a copy of every **std::string** in **roster** - a reference suffices. And we are not modifying the name, so we use **const**.

Range For Loop

In many cases, we can use **auto**: beware that auto does not adhere to making references to const without our explicitly writing so.

```
// okay
for ( const auto& name : roster ) {
    cout << name << endl;
}

// PROBLEM: inefficient: makes copies of each string!
for ( auto name : roster ) {
    cout << name << endl;
}
```

Ternary (Conditional) Operator

The **ternary** or **conditional operator** allows us to write certain if/else branching processes rather neatly. It has the following syntax:

(condition to check) ? (what to return if true) : (what to return if false)

For example, we can assign **z** to the maximum of **x** and **y**:

```
int x = 3, y = 4, z = 0;

z = ( (x < y) ? y : x );
```

When **(x<y)** is true, the output is **y**; otherwise the output is **x**.

Ternary (Conditional) Operator

We generally put parentheses around this operator; it has a very low priority in order of operations (**operator precedence**) and the parentheses around the entire expression are important to avoid unintentional bugs. For example,

```
int x = 3, y = 4;
cout << ( x < y ) ? y : x; // will print 1 to the screen, not 4
```

Template Functions

Here we very briefly consider the notion of templates. Consider writing functions that will compute the maximum between two variables of the same type. It can quickly get repetitive:

```
int max(int x, int y) {
   return ( x < y ) ? y : x;
}
float max(float x, float y) {
   return ( x < y ) : y : x;
}
const string& max( const string& x, const string& y) {
   return ( x < y ) ? y : x;
}
const vector<double>& max( const vector<double>& x,
   const vector<double>& y) {
   return ( x < y ) ? y : x;
}
```

Template Functions

Remarks: the **std::vector** templated class has a comparison operator. Two vectors are compared lexicographically by element, just like **std::string**s with their **char**s.

The logic is identical in all cases; note that for objects, we use references instead of values because we wish to be efficient.

Templates allow us to instruct the compiler how to define a range of functions based on arbitrary input parameters.

Template Functions

Here's how we can write a **max** template function to manage all input types (this is the same implementation as in the C++ Standard):

```
/**
@tparam T the value type we are comparing

@param x the first value
@param y the second value

@return the maximum of x and y based upon <
*/

template<typename T>
const T& max( const T& x, const T& y) {
   return (x < y) ? y : x;
}
```

Template Functions

A template function is documented similarly to a non-template function, but we also specify what the generic argument represents with **tparam** descriptors.

We tell the compiler we are creating a template with the **template** keyword.

This template applies to an arbitrary parameter type that we call T (could use any name), and the **typename** tells the compiler that T can be any type.

The template parameters appear inside of angled brackets < ... >

Template Functions

Because **max** could be called on an arbitrary input type, we prepare for the worst (the input being a large class object) and accept **const T&** arguments as opposed to **T**.

Returning **const T&** is more efficient since it references the return value rather than having to copy/move it. In certain edge cases, it can be dangerous: if the output is referencing a variable that is only local to the function, that local variable may already be destroyed by the time we use it.

Template Functions

Given the templated definition for **max**, we can now write code:

```
cout << max(5,7) << endl;
cout << max(14.4, -8.2) << endl;

string s1("apple");
string s2("orange");
cout <<max(s1, s2);
```

```
7
14.4
orange
```

Template Functions

For templated functions, the compiler is often able to deduce the type of the input variables without our explicitly writing them.

```
max(5,7); // calls max<int>(5,7);
```

We can also be explicit:

```
max<int>(5,7); // also okay
```

At times, being explicit is necessary to avoid ambiguities such as

```
max(3, 11.22); // ERROR: call max<int> or max<double>?
```

```
max<int>(3, 11.22); // okay, calls max<int> and converts 11.22 to 11 (int)
```

Template Functions

Consider printing all the elements within a **std::vector** of a given type:

```
/**
Prints all elements of a vector

@tparam T the type of elements the vector stores

@param collection the vector storing the elements
*/
template<typename T>
void printElements( const vector<T>& collection ) {
    for ( const auto& i : collection ) { // for each i in the vector
        cout << i <<endl; // print it
    }
}
```

Remark: we use **const auto &** because **T** could be a class type and we wish to be efficient and const correct!

Template Functions

The general syntax for defining a template with a single template parameter is

template <typename T>
returnType functionName(comma separated list of arguments) {
*   instructions;*
}

Remark: unlike ordinary functions, a template function should generally be defined in a header file!

Summary

- Functions allow us to write cleaner code, segregating long blocks of code that are repeated, and providing a means to invoke those blocks of code throughout the program.
- Functions should be documented carefully by listing what they do, what their inputs are, and what their outputs are.
- Functions are **declared** with their name, inputs and outputs; they are **defined** with a body.
- Functions, unless returning **void**, must **return** a value.
- References are necessary for a function to modify its arguments and for efficiency in passing class objects.
- **Const** correctness applies for function arguments, allowing the programmer to promise not to modify a local variable.
- Functions can be **overloaded** whereby they take different arguments, with a best match sought; they cannot be overloaded based on their return types.
- Arrays **decay to pointers** when passed to functions.

Summary

- Iterators behave similarly to pointers and allow for simpler code to **insert**/**remove** elements of containers or to even **sort** them!
- The **auto** keyword can pick up on the type of the right-hand side of a definition.
- A **range for loop** traverses a container without having to worry about indexing.
- The **ternary** conditional operator can be used to shorten simple if/else structures.
- **Function templates** can be used to cut down on writing nearly identical code many times.
- Templates should generally be defined in header files, not in cpp files.