Introduction to C++: Part 2

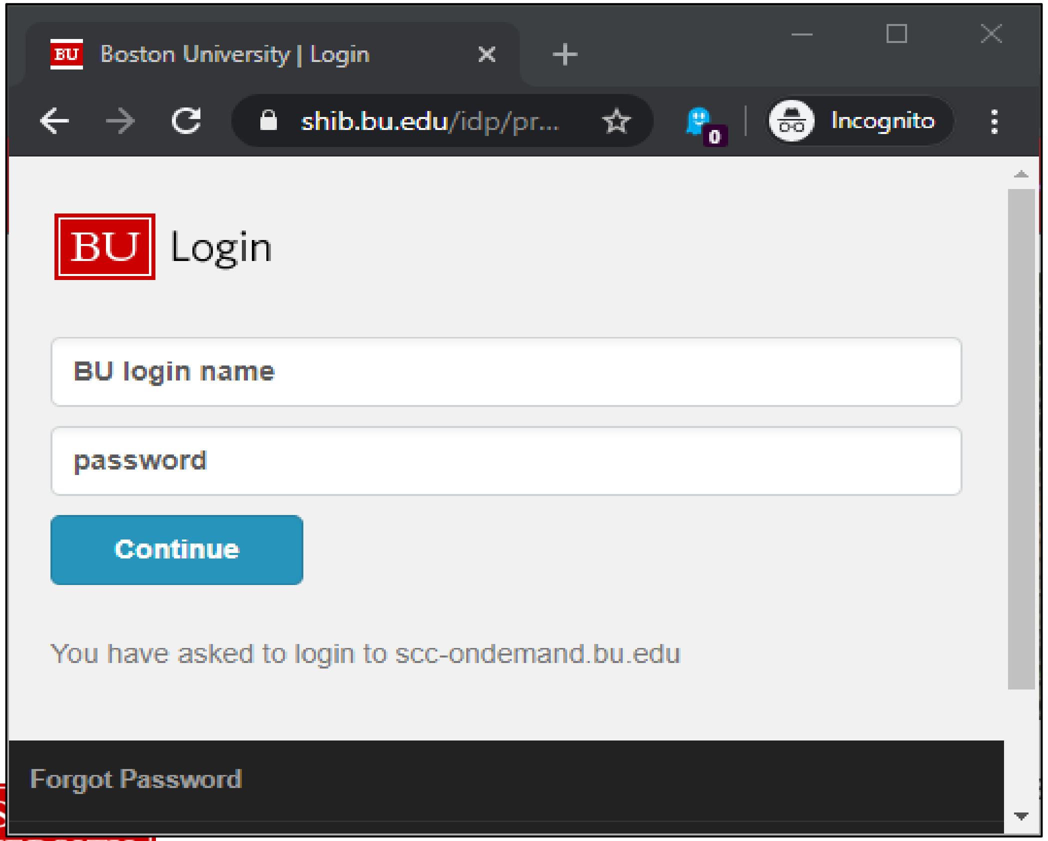
BOSTON UNIVERSITY

Tutorial Outline: Part 2

- Compiler Options
- References and Pointers
- Solve a Programming Problem
- Intro to the Standard Template Library
- Function Overloads
- Generic Functions

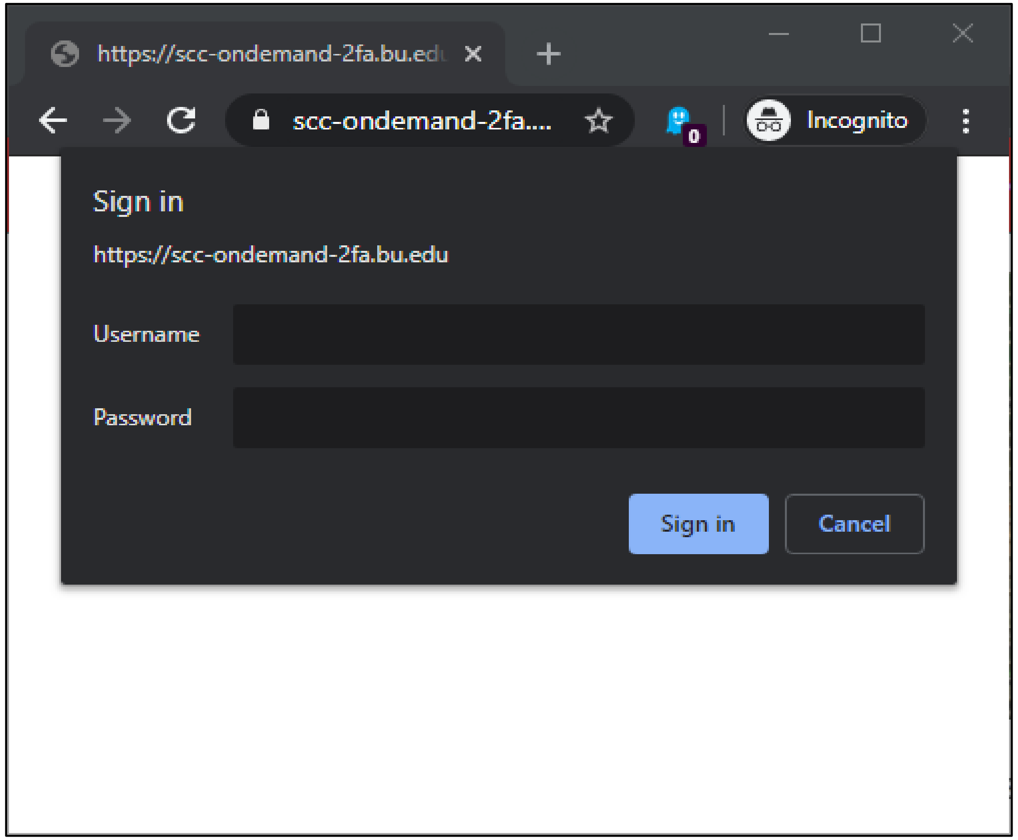
Existing SCC Account

1. Open a web browser
2. Navigate to http://scc-ondemand.bu.edu
3. Log in with your BU Kerberos Credentials

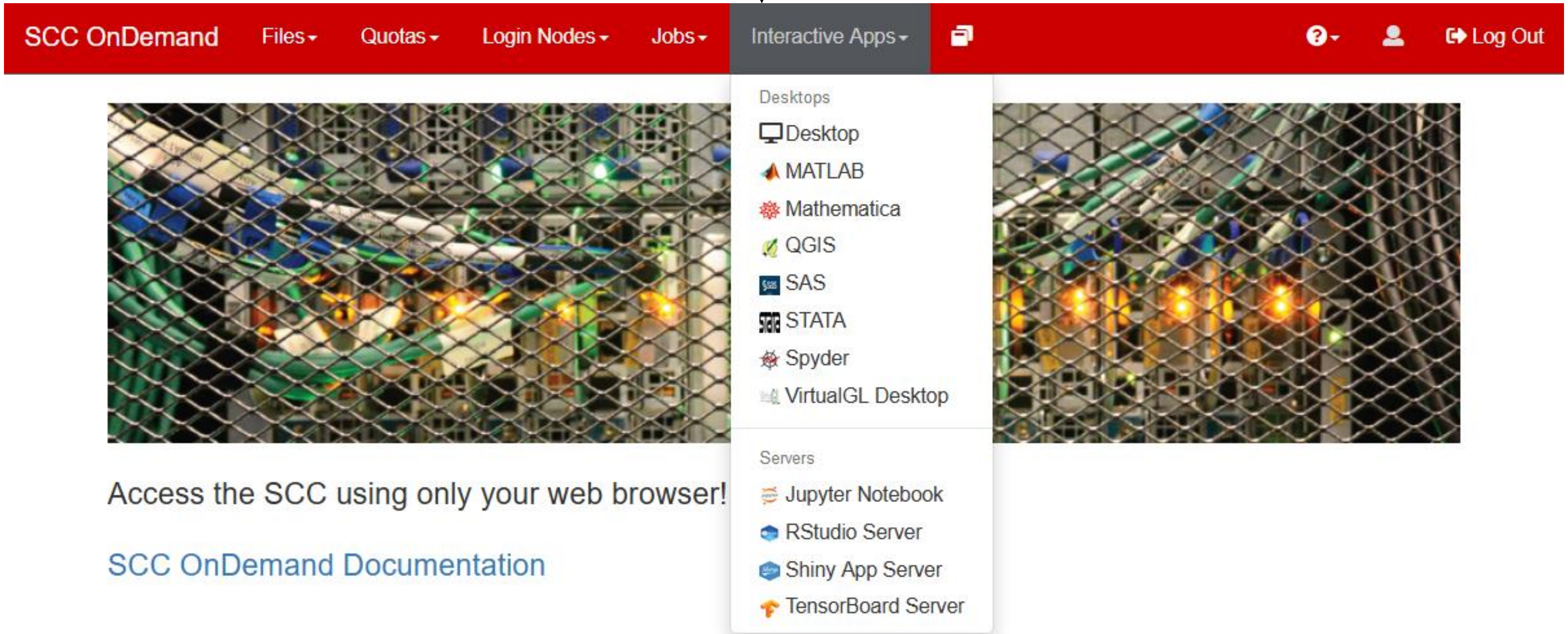


Temporary Tutorial Account

1. Open a web browser
2. Navigate to http://scc-ondemand-**2fa**.bu.edu
3. Log in with Tutorial Account



Click on Interactive Apps/Desktop



Interactive Apps

Desktops

- Desktop
- MATLAB
- Mathematica
- QGIS
- SAS
- STATA
- Spyder
- VirtualGL Desktop

Servers

- Jupyter Notebook
- RStudio Server
- Shiny App Server
- TensorBoard Server
- Webserver

Desktop

This app will launch an interactive desktop on a compute node.

List of modules to load (space separated)

eclipse/2019-06 gcc/8.3.0 | Select Modules

← eclipse/2019-06 gcc/8.3.0

Working Directory

Select Directory

The directory to start in. (Defaults to home directory.)

Initial command to run

xfce4-terminal

Number of hours

3

— 3

Number of cores

1

Number of gpus

0

Project

scv

Extra qsub options

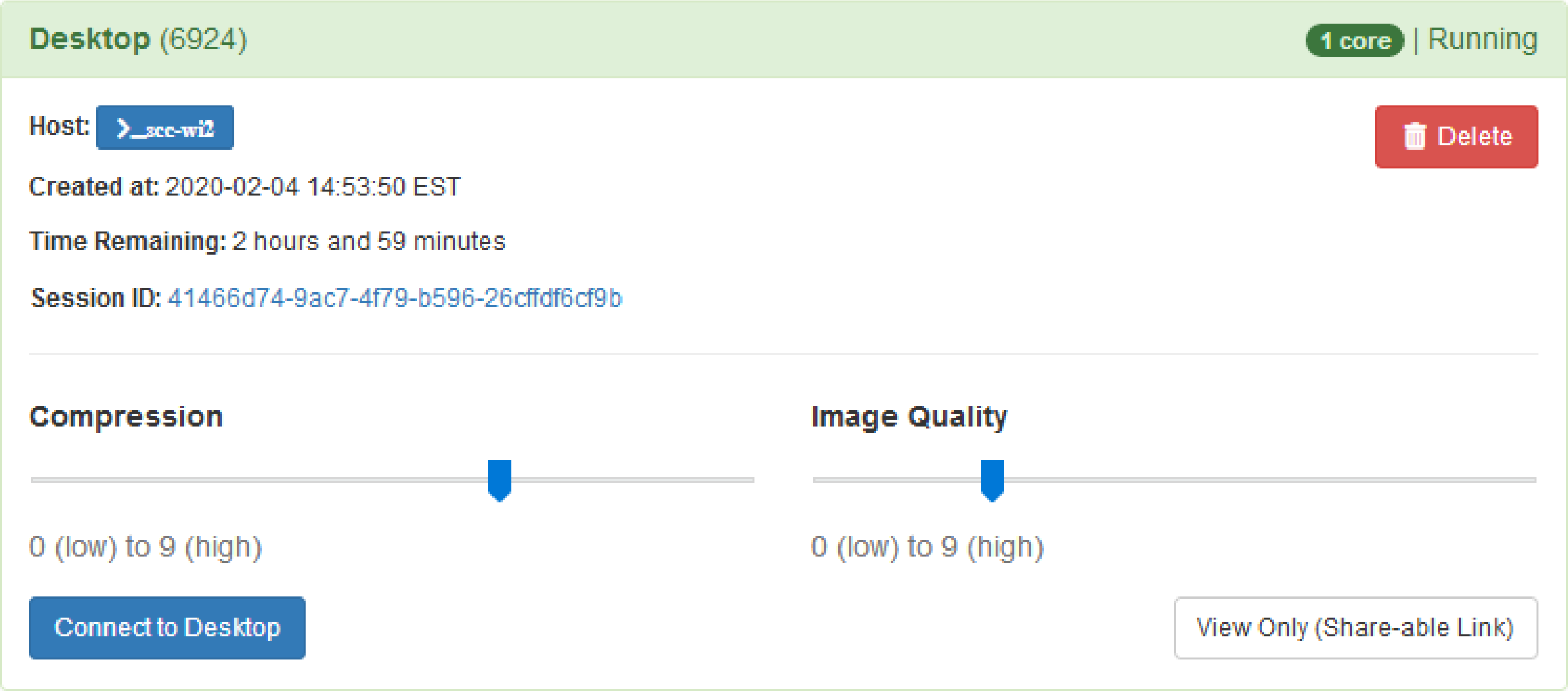
☐ I would like to receive an email when the session starts

Launch

← click

* The Desktop session data for this session can be accessed under the data root directory.

BOSTON UNIVERSITY



When your desktop is ready click *Connect to Desktop*

BOSTON UNIVERSITY

- Enter this command to create a directory in your home folder and to copy in tutorial files:

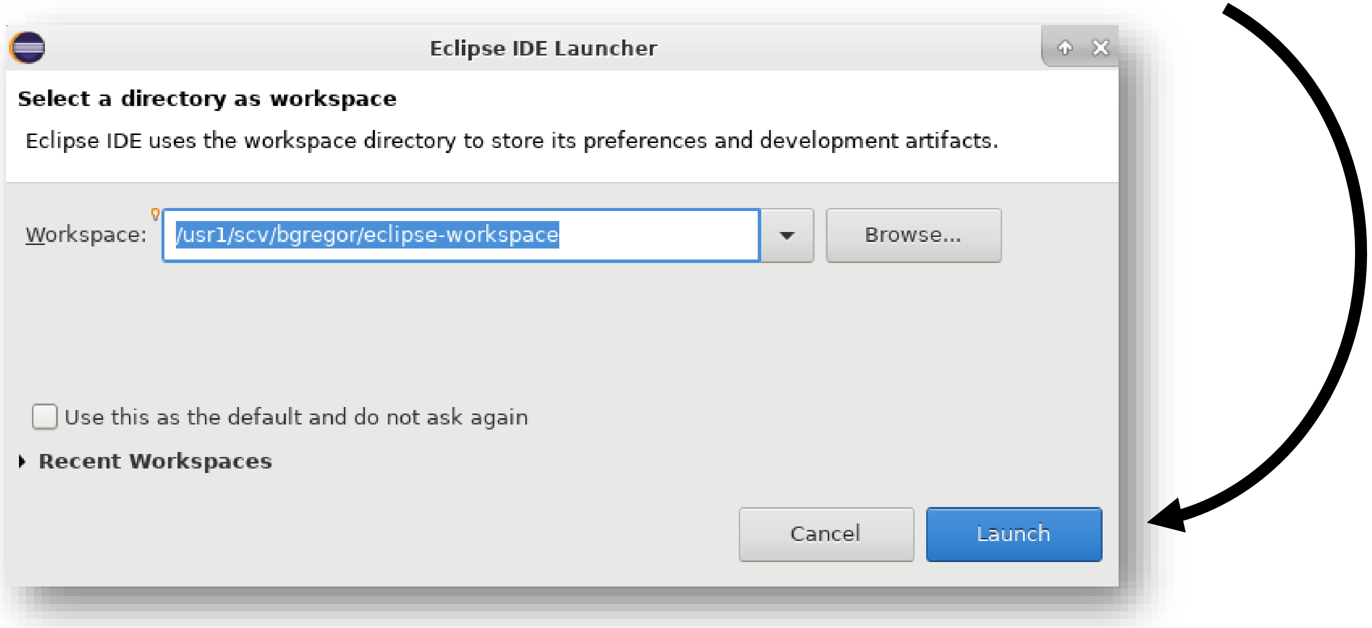
```
/net/scc2/scratch/intro_to_cpp_2.sh
```

Run the Eclipse software

- Start up the Eclipse development environment.

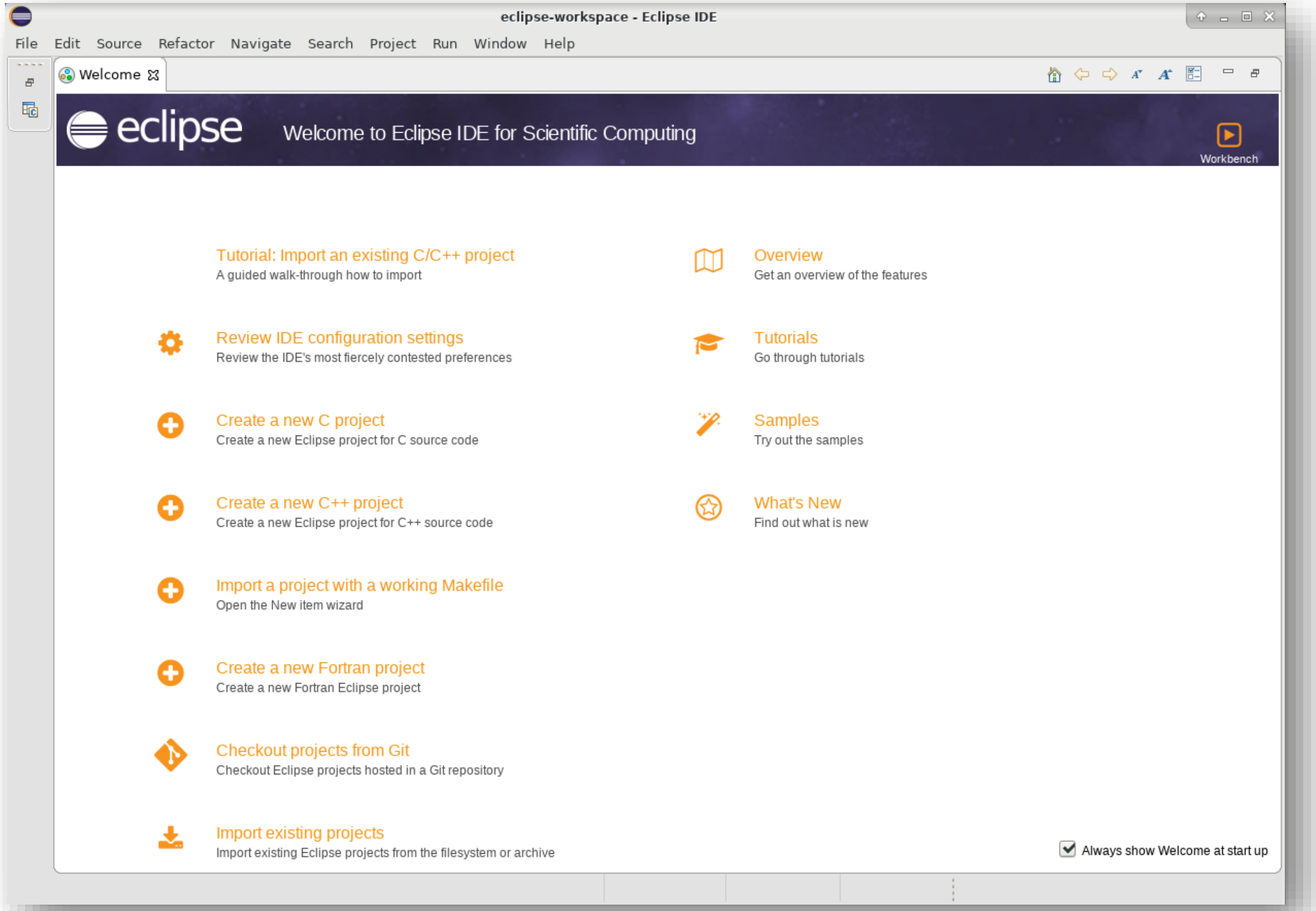
```
eclipse &
```

- When this window appears just click the Launch button:



Run the Eclipse software

- When this window appears just leave it be for now.



Compiler Options

- The g++ compiler has a **vast** array of options:
  - https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc/Invoking-GCC.html#Invoking-GCC
  - This is typical for compiled languages.

- Turning on optimizations makes the compiler work harder to produce code that will execute faster.
  - What happens in optimization? https://en.wikipedia.org/wiki/Optimizing_compiler

Compiler Options (for g++ 8.3.0)

- Common flags:
  - **-g** Support for debugging. Sometimes not completely effective with (any) optimization turned on.
  - **-std=c++11** Enable C++11 standards (on by default in 8.3.0)
    - These work too with 8.3.0: c++14, c++17
    - If you want newer support (say c++20) use a newer g++, accessed thru the SCC gcc modules.
  - **-Og** Optimize but don't do anything that will cause issues while running the debugger.
  - **-O, -O2, -O3** Produce optimized code. The higher numbers let the compiler try more strategies to generate code. They are less likely to have an impact.
    - Can be combined with **-g** but makes debugging more difficult.
  - **-ffast-math -funsafe-math-optimizations** May produce code that does not conform to IEEE standards for floating point computations. Try it with your program and see if it has any impact on accuracy and/or speed.
  - **-march=sandybridge** On the SCC, allow for some special CPU instructions (AVX) to be generated for some code that may result in better performance. Use the qsub option "-l avx" to run code compiled with this flag.

Using Compiler Options

- An IDE like Eclipse will apply these for you when building.
- On the command line (for a single source file program):

Debug

```
g++ -o my_program -g my_source.cpp
```

Debug with optimizations

```
g++ -o my_program -g -Og my_source.cpp
```

Release

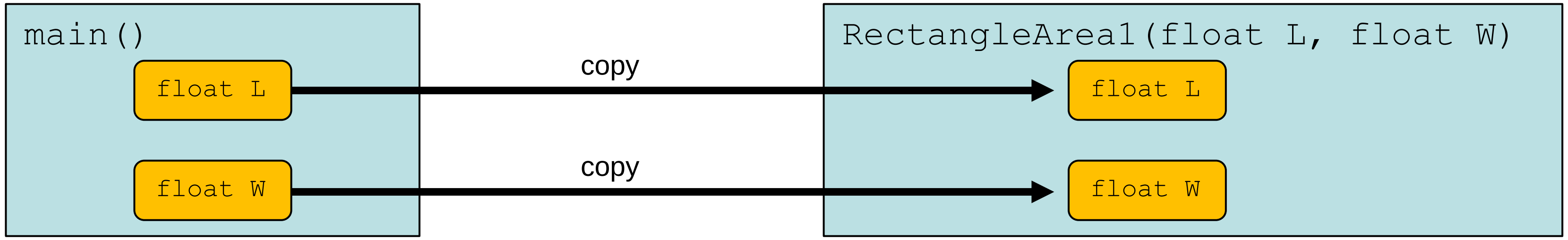
```
g++ -o my_program -O3 my_source.cpp
```

BOSTON UNIVERSITY

Tutorial Outline: Part 2

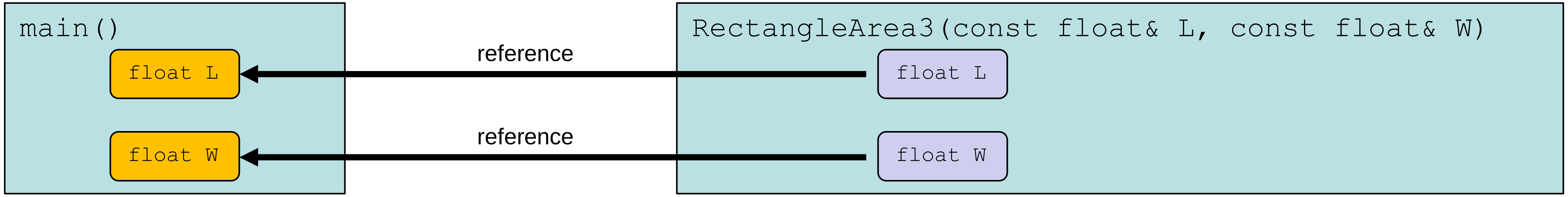
- Compiler Options
- References and Pointers
- Solve a Programming Problem
- Intro to the Standard Template Library
- Function Overloads
- Generic Functions

Pass by Value



- C++ defaults to *pass by value* behavior when calling a function.
- The function arguments are **copied** when used in the function.
- Changing the value of L or W in the RectangleArea1 function does **not** effect their original values in the main() function
- When passing objects as function arguments it is important to be aware that potentially large data structures are automatically copied!

Pass by Reference



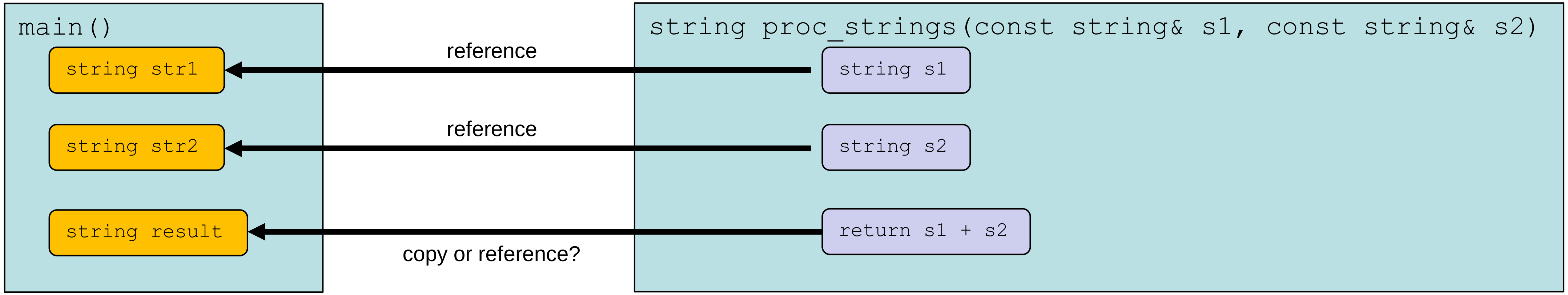
- *Pass by reference* behavior is triggered when the & character is used to modify the type of the argument.

- Pass by reference function arguments are **NOT** copied. Instead the compiler sends a *pointer* to the function that references the memory location of the original variable. The syntax of using the argument in the function does not change.

- The *const* modifier can be used to prevent changes to the original variable in main().

BOSTON UNIVERSITY

- In C++ arguments to functions can be objects…
  - Example: Consider a string variable containing 1 million characters (approx. 1 MB of RAM).
    - Pass by value requires a copy – 1 MB
    - pass by reference requires 8 bytes.



- Returning references is allowed but the reference'd value must be in memory. Here – the new string is local to the function. Don't return a reference to it, that string will get cleaned up when the function is done! Return as a copy.
  - But…compilers will help here….

BOSTON UNIVERSITY

Rules of thumb for function/method arguments

- Basic types (int, float, etc) just pass by value unless you need to use them to return values.
  - int - 4 bytes
  - int& - 8 bytes (64-bit memory address)

- Pass all objects by reference.
  - use the *const* modifier in the function definition whenever appropriate to protect yourself from accidentally modifying variables.

Tutorial Outline: Part 2

- Compiler Options
- References and Pointers
- Solve a Programming Problem
- Intro to the Standard Template Library
- Function Overloads
- Generic Functions

Looping

Loop variable

Loop if true

Change applied to loop variable after each iteration

```
for (int i = 0 ; i < 10; ++i)
{
        // ++i means "add 1 to the value of index"
        cout << i << " " ;
}
```

- Loop with a “for” loop, referencing the value of vec using brackets.
- 1st time through:
  - i = 0
  - Print its value
  - i gets incremented by 1
- 2nd time through:
  - i = 1
  - Etc
- After last time through
  - Loop exits
  - Loop variable *i* was declared with the loop – it is NOT available after the loop!

BOSTON UNIVERSITY

Problem 1 from Project Euler

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

```
answer <- 0
for (i in 1:999) {
  if (i %% 3 == 0 | i %% 5 == 0)
            answer <- answer + i
}
print(answer)
```

Solution in R

```
numsum = 0

for i in range(1000):
    if (i%3 == 0 or i%5==0):
        numsum += i

print(f'The sum is: {numsum}')
```

Solution in Python

```
n = 1:(999/3);
N = 1:(999/5);
multiples_3 = 3.*n;
multiples_5 = 5.*N;
allmultiples = [multiples_3 multiples_5];
answer = sum(unique(allmultiples));
fprintf('The answer is %.0d\n',answer)
```

Solution in Matlab (no loops)

Let's work out a C++ version of this.

BOSTON UNIVERSITY

- Start an Eclipse project
- Implement the solution in the main() routine.
- Got that working? Move it to a function that takes the max integer as an argument.

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

Answer: 233168.

```
// C++ if/else statement
if (boolean condition1) {
    // do this if true
} else if (condition2) {
    // rather do this if
    // true
} else {
    // the default
}
```

```
// C++ if statement
if (condition) {
   // do this if true
}
```

- Arithmetic: + - * / % ++ --
- Logical: && (AND) || (OR) ! (NOT)
- Example: `x || !y` is “x OR NOT y”
- Comparison: == > < >= <= !=

```
for (int i = 0 ; i < 10; ++i)
{
        cout << i << " " ;
}
```

Pieces of C++ you'll need

BOSTON UNIVERSITY

Stepping back a bit

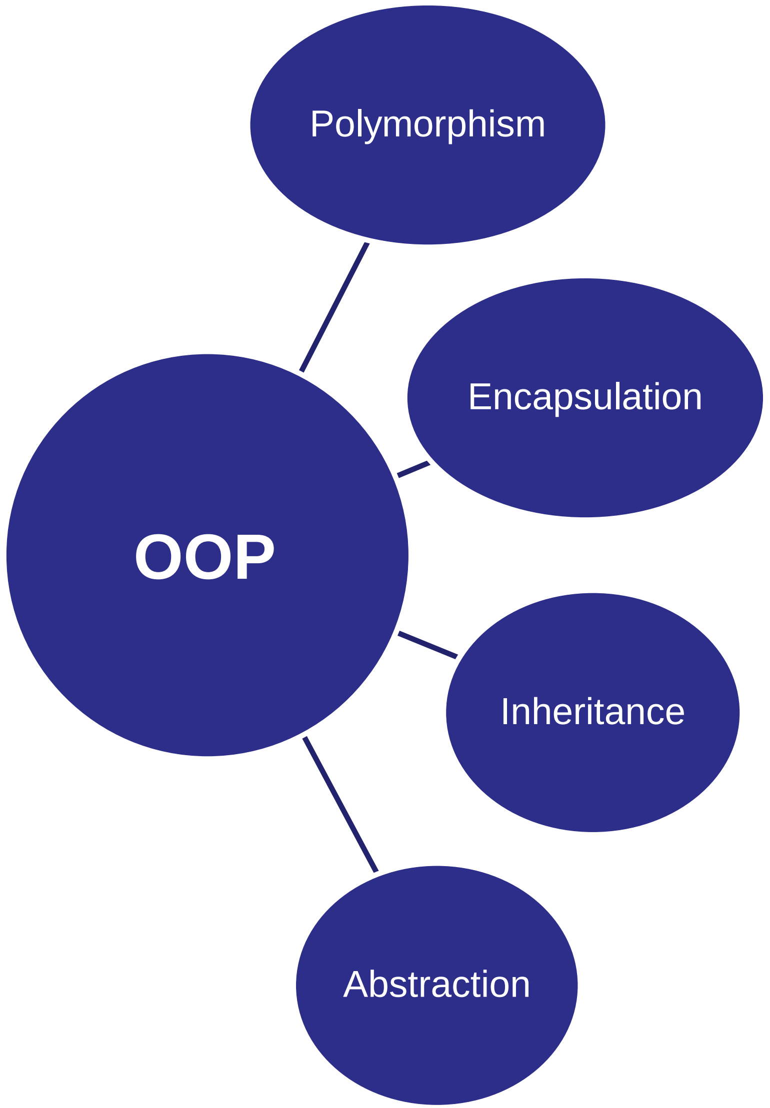
- Summary so far:
  - Basics of C++ syntax
  - Declaring variables
  - Defining functions
  - Using the IDE

- As an object-oriented language C++ supports a core set of OOP concepts.

- Knowing these concepts help with understanding some of the underlying design of the language and how it operates in your programs.

The formal concepts in OOP

- The core concepts in addition to classes and objects are:
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction



BOSTON UNIVERSITY

Core Concepts

- Encapsulation
  - Bundles related data and functions into a class
- Inheritance
  - Builds a relationship between classes to share class members and methods
- Abstraction
  - The hiding of members, methods, and implementation details inside of a class.
- Polymorphism
  - The application of the same code to multiple data types

Core Concepts in this tutorial

- Encapsulation
  - Demonstrated by writing some classes
- Inheritance
  - Write classes that inherit (re-use) the code from other classes.
- Abstraction
  - Design and setup of classes, discussion of the Standard Template Library (STL).
- Polymorphism
  - Function overloading, template code, and the STL

Tutorial Outline: Part 2

- Compiler Options
- References and Pointers
- Solve a Programming Problem
- Intro to the Standard Template Library
- Function Overloads
- Generic Functions

The Standard Template Library

- The STL is a large collection of containers and algorithms that are part of C++.
  - It provides many of the basic algorithms and data structures used in computer science.

- As the name implies, it consists of generic code that you specialize as needed.

- The STL is:
  - Well-vetted and tested.
  - Well-documented with lots of resources available for help.

Containers

- There are 16 types of containers in the STL (C++11):

Container	Description
array	1D list of elements.
vector	1D list of elements
deque	Double ended queue
forward_list	Linked list
list	Double-linked list
stack	Last-in, first-out list.
queue	First-in, first-out list.
priority_queue	1st element is always the largest in the container

Container	Description
set	Unique collection in a specific order
multiset	Elements stored in a specific order, can have duplicates.
map	Key-value storage in a specific order
multimap	Like a map but values can have the same key.
unordered_set	Same as set, sans ordering
unordered_multiset	Same as multisetset, sans ordering
unordered_map	Same as map, sans ordering
unordered_multimap	Same as multimap, sans ordering

BOSTON UNIVERSITY

Specifying the Type

- The STL is implemented entirely in header (.h) files. When used in your program, the compiler generates the required code as needed.
- You must tell the compiler what sort of types STL containers will hold.

```
#include <vector>
#include <unordered_map>
#include <tuple>
using namespace std ;

vector<int> v(3);  // Declare a vector of integers

// A map with string keys that holds doubles.
unordered_map<string, double> my_map ;
// insert a value
my_map["xyz"] = 2.0 ;
// get a value
auto val = my_map["xyz"] ;

// A tuple containing an int and a character
tuple<int,char> tpl (10,'x');
int z = get<0>(tpl) ; // retrieve the int at location 0
```

BOSTON UNIVERSITY

Algorithms

- There are [85+ of these](#).
  - Example: find, count, replace, sort, is_sorted, max, min, binary_search, reverse
- Algorithms manipulate the data stored in containers but is not tied to STL containers
  - These can be applied to your own collections or containers of data
- Example:

```
vector<int> v(3);                // Declare a vector of 3 elements.
v[0] = 7;
v[1] = 3;
v[2] = v[0] + v[1];              // v[0] == 7, v[1] == 3, v[2] == 10
reverse(v.begin(), v.end()) ;  // v[0] == 10, v[1] == 3, v[2] == 7
```

- The implementation is hidden and the necessary code for reverse() is generated from templates at compile time.

vector<T>

- A very common and useful class in C++ is the vector class. Access it with:

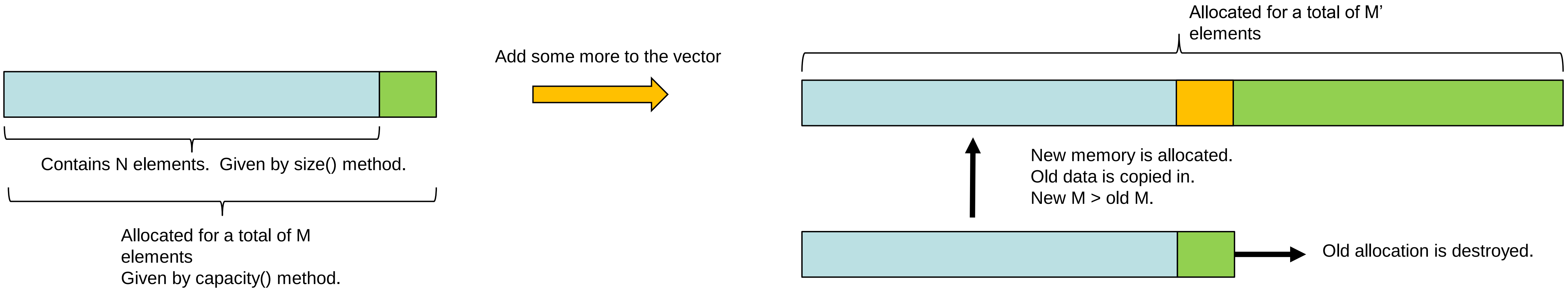
```
#include <vector>
// optional
using namespace std ;
```

- Vector has many methods:
  - Various constructors
  - Ways to iterate or loop through its contents
  - Copy or assign to another vector
  - Query vector for the number of elements it contains or its backing storage size.
- Example usage: `vector<float> my_vec ; // an empty float vector`
- Or: `vector<float> my_vec(50) ; // ready for 50 elements`

vector<T>

- Hidden from the programmer is the *backing store*
- Object oriented design in action!

- This is how the vector stores its data internally.



Construction and Destruction

- A special function called the *constructor* is called when an object is created.

- This is used to initialize an object:
  - Load values into member variables
  - Open files
  - Connect to hardware, databases, networks, etc.

- The *destructor* is called when an object goes *out of scope*.
- Example:

```
void function() {
    ClassOne c1 ;
    // stuff happens…
}
```

- Object *c1* is created when the program reaches the first line of the function and destroyed when the program leaves the function.

BOSTON UNIVERSITY

Scope

- Scope is the region where a variable is valid.
- Constructors are called when an object is created.
- Destructors are called automatically when a variable is out of scope.

```
int main() { // Start of a code block
      // in main function scope
      float x ;  // No constructors for built-in types
      ClassOne c1 ;   // c1 constructor ClassOne() is called.
      if (1){  // Start of an inner code block
            // scope of c2 is this inner code block
            ClassOne c2 ; //c2 constructor ClassOne() is called.
      }   // c2 destructor  is called.
      ClassOne c3 ;  // c3 constructor ClassOne() is called.
}   // leaving program, call destructors for c3 and c1 in that order
    // variable x: no destructor for built-in type
```

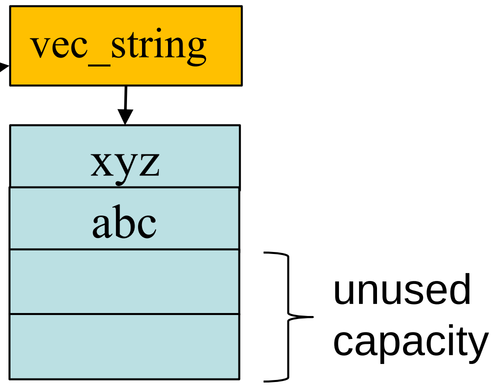
Destructors

- vector<t> can hold most types of objects:
  - Primitive (aka basic) types: int, float, char, etc.
  - Objects: string, your own classes, file objects, etc.
  - Pointers: int*, string*, etc.
  - But NOT references!
- When a vector is destroyed:
  - If it holds primitive types or pointers it just deallocates its backing store.
  - If it holds objects it will call each object's destructor before freeing its backing store.

Vector of Objects Destruction

```
void function(string a, string b) { {
        vector<string> vec_string = {a,b} ;
        // do something with the vector
} // leaving, call the vec_string destructor

// …somewhere in the program…
str_function("xyz","abc")
```



- String "abc" is destroyed first
- Then "xyz"
  - i.e. in reverse order
- Then *vec_string*

vector<t> with objects

- Select an object in a vector.
- The members and methods can be accessed directly.

- Elements can be accessed with brackets and an integer starting from 0.

```
// a vector with memory preallocated to
// hold 1000 objects.
vector<MyClass> my_vec(1000);

// Now make a vector with 1000 MyClass objects
// that are initialized using the MyClass constructor
vector<MyClass> my_vec2(1000,MyClass(arg1,arg2));

// Access an object's method.
my_vec2[100].some_method() ;
// Or a member
my_vec2[10].member_integer = 100 ;


// Clear out the entire vector
my_vec2.clear()
// but that might not re-set the backing store…
// Let's check the docs:
// http://www.cplusplus.com/reference/vector/vector/clear/
```

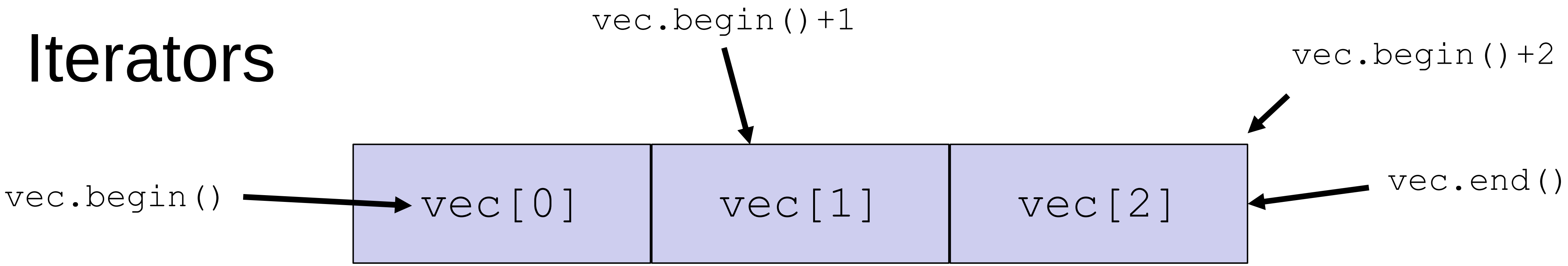
BOSTON UNIVERSITY

Looping

```
for (int index = 0 ; index < vec.size() ; ++index)
{
        // ++index means "add 1 to the value of index"
        cout << vec[index] << " " ;
}
```

- Loop with a “for” loop, referencing the value of vec using brackets.
- 1st time through:
  - index = 0
  - Print value at vec[0]
  - index gets incremented by 1
- 2nd time through:
  - Index = 1
  - Etc
- After last time through
  - Index now equal to vec.size()
  - Loop exits
- Careful! Using an out of range index will likely cause a memory error that crashes your program.

Iterators



- Iterators are generalized ways of keeping track of positions in a container.
- 3 types: forward iterators, bidirectional, random access
- Forward iterators can only be incremented (as seen here)
- Bidirectional can be added or subtracted to move both directions
- Random access can be used to access the container at any location
  - Bracket indexing [ ] is an example of random access.

BOSTON UNIVERSITY

Looping

```
for (vector<int>::iterator itr = vec.begin(); itr != vec.end() ; ++itr)
{
      cout << *itr << " " ;
      // iterators are pointers!
}
```

- Loop with a “for” loop, referencing the value of vec using an **iterator** type.
- `vector<int>::iterator` is a type that iterates through a vector of int’s.
- 1st time through:
  - itr points at 1st element in vec
  - Print value pointed at by itr: *itr
  - itr is incremented to the next element in the vector
- Iterators are very useful C++ concepts. They work on any STL container!
  - **No need to worry about the # of elements!**
  - Exact iterator behavior depends on the type of container but they are **guaranteed** to always reach every value.

BOSTON UNIVERSITY

Looping

```
for (auto itr = vec.begin() ; itr != vec.end() ; ++itr)
{
        cout << *itr << " " ;
}
```

- Let the *auto* type asks the C++ compiler to figure out the iterator type automatically.

```
for (auto itr = vec.begin(), auto vec_end = vec.end() ; itr != vec_end ; ++itr)
{
        cout << *itr << " " ;
}
```

- An extra modification: Assigning the vec_end variable avoids calling vec.end() on every loop.
  - Save yourself a function call.

Looping

```
for(const auto &element : vec)
{
        cout << element << " " ;
}
```

- Another iterator-based loop: iterator behavior and accessing an element are handled automatically by the compiler
- Uses a reference so the element is not copied.
- The ***const auto &*** prevents changes to the element in the vector.
- If you don't use *const* then the loop can update the vector elements via the reference.

- Less typing == less chance for program bugs.

Iterator notes

- There is very small performance penalty for using iterators…but are they safer to use.
- They allow substitution of one container for another (list<> for vector<>, etc.)
- With your own template code you can write a function that accepts any STL container type.

```
template<typename T>
void dump_string(T &t)
{   // print the contents of any STL container
    for( auto itr=t.begin() ; itr!=t.end() ; itr++) {
        cout << *itr << endl;
    }
}
```

```
list<float> lst ;
lst.push_back(-5.0) ;
lst.push_back(12.0) ;
vector<double> vec(2) ;
vec[0] = 1.0 ;
vec[1] = 2.0 ;

dump_string<list<float> >(lst) ;
dump_string<vector<double> >(lst) ;
```

BOSTON UNIVERSITY

STL Demo

- Open project *STL_Demo*

- Let's walk through the functions with the debugger and see some vectors in action.

Tutorial Outline: Part 2

- Compiler Options
- References and Pointers
- Solve a Programming Problem
- Intro to the Standard Template Library
- Function Overloads
- Generic Functions

Function overloading

- The same function can be implemented multiple times with different arguments.

- This allows for special cases to be handled, or specialized behavior for different types.

- cout and the << operator are an example of function overloading
  - << is just a function.

```
float sum(float a, float b) {
     return a+b ;
}

int sum(int a, int b) {
     return a+b ;
}
```

BOSTON
UNIVERSITY

Function overloading

- Overloaded functions are differentiated by their arguments and not the return type.
  - The number of arguments and their types can be varied.

- The compiler will decide which overload to use depending on the types of the arguments.

- If it can't decide a compile-time error will occur.

```
float sum(float a, float b) {
    return a+b ;
}

int sum(int a, int b) {
    return a+b ;
}
```

BOSTON UNIVERSITY

C++ Templates (aka generics)

- Generic code is code that works on multiple different data types but is only coded once.
- In C++ this is called a *template*.

- A C++ template is implemented entirely in a header file to define generic classes and functions.

- The actual code is generated **by the compiler** wherever the template is used in your code.
  - There is NO PENALTY when your code is running!

C++ Templates (aka generics)

- Template code should be placed in header (.h) files.
- A source code file (.cpp) is not needed for template code.
- Expect longer compile times – the compiler has to do a lot more work.
- Executing code created by templates is often **much** faster when compiler optimizations are turned on.

BOSTON UNIVERSITY

Sample template function

- The template is started with the keyword *template* and is told it'll handle a type which is referred to as *T* in the code.
  - Templates can be created with multiple different types, not limited to just one.
  - You don't have to use *T*, any non-reserved word will do.

```
// In a header file
template <typename T>
T sum_template (T a, T b) {
    return a+b ;
}
```

- Specialize the template to the type you want to use.

```
// Then call the function in a
// source file:
float x=1.0 ;
float y=2.0 ;
auto z=sum_template<float>(x,y) ;
```

BOSTON UNIVERSITY

An Example

- Open the project *Overloads_and_templates*
- This is an example of simple function overloads and a template function.
- New for 2022: Check out C++ Insights.
  - Let’s go here: https://cppinsights.io/s/302c7276

BOSTON UNIVERSITY

When to use function overloading and templates?

- When it makes your code easier to use, maintain, write, or debug!
  - Overloads are easier to use effectively.

- Templating everything in your code does not make it better, just harder to develop.
  - Longer compiles, harder to debug, etc.

- More experienced C++ programmers should use these features where appropriate.