12

CSE 1325

Week of 11/09/2020

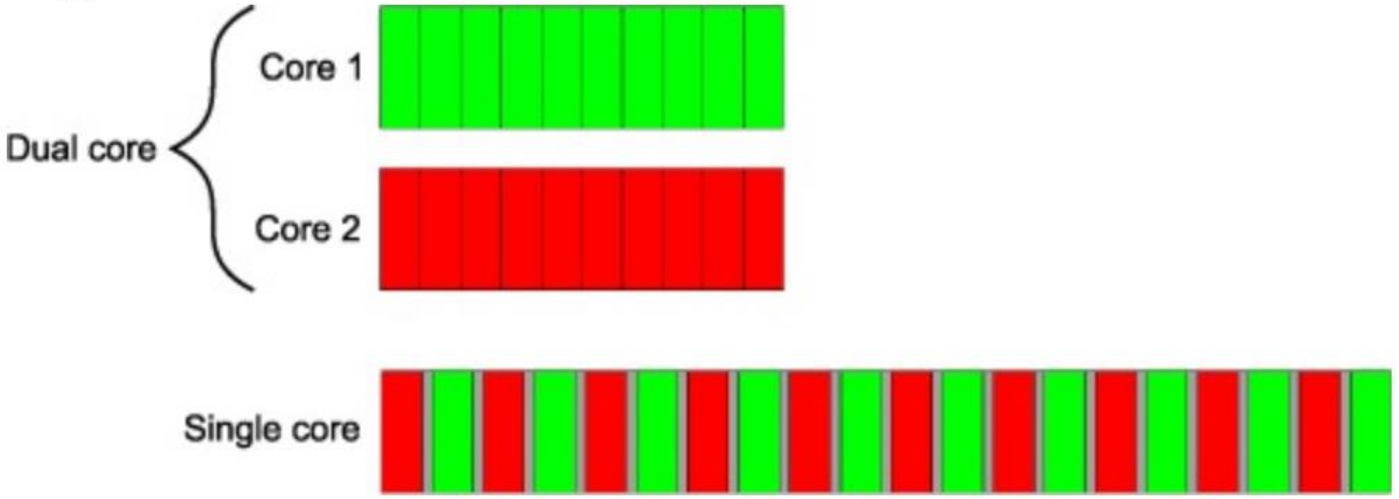
Instructor : Donna French

Multithreading

Most of today's computers, smartphones and tablets are typically multicore.

The most common level of multicore processor today

- dual core
- quad core



In multicore hardware systems, the hardware can put multiple processes to work simultaneously on different parts of your task; thereby, enabling the program to complete faster.

Multithreading

To take full advantage of multicore architecture, we need to write multithreaded applications.

When a program splits tasks into separate threads, a multicore system can run those threads in parallel.

When you run any program on a modern computer system, your program's tasks compete for the attention of the processor(s) with the operating system, other programs and other activities that the operating system is running on your behalf. All kinds of tasks are typically running in the background of your system.

Multithreading

Therefore, it is important to recognize that different runs of the same process may take different amounts of time and the various threads may run in different orders at different speeds.

There's also overhead inherent to multithreading itself. Simply dividing a task into two threads and running it on a dual core system does not guarantee that it will run twice as fast.

Multithreading

There is not guarantee of which threads will execute when and how fast they will execute regardless of how the program is designed or how the processors are laid out.

Concurrency

The ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

Process

A self-contained execution environment including its own memory space.

Thread

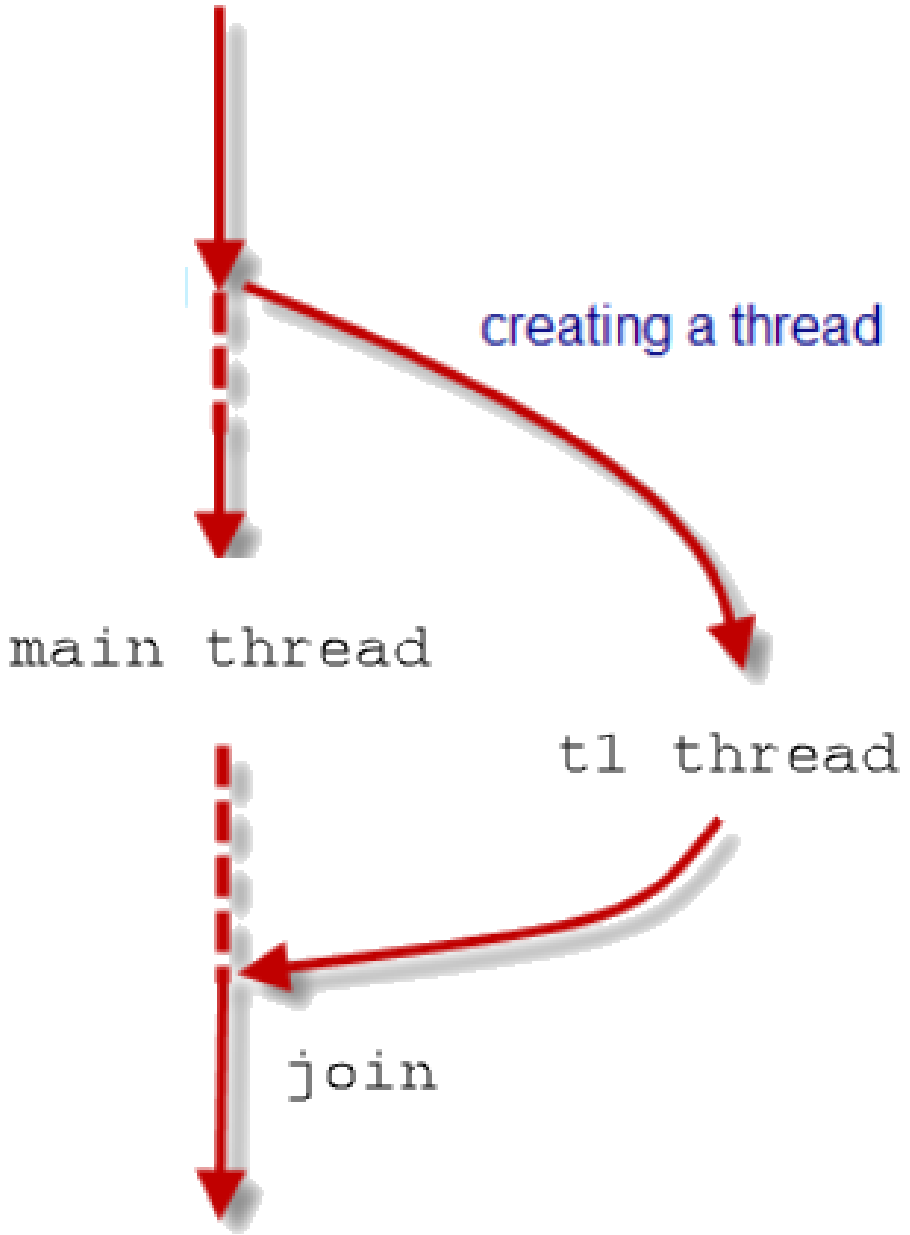
An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space.

Threads

Class to represent individual threads of execution.

A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space.

`main()` is a thread



Threads

Real World Examples of Threads

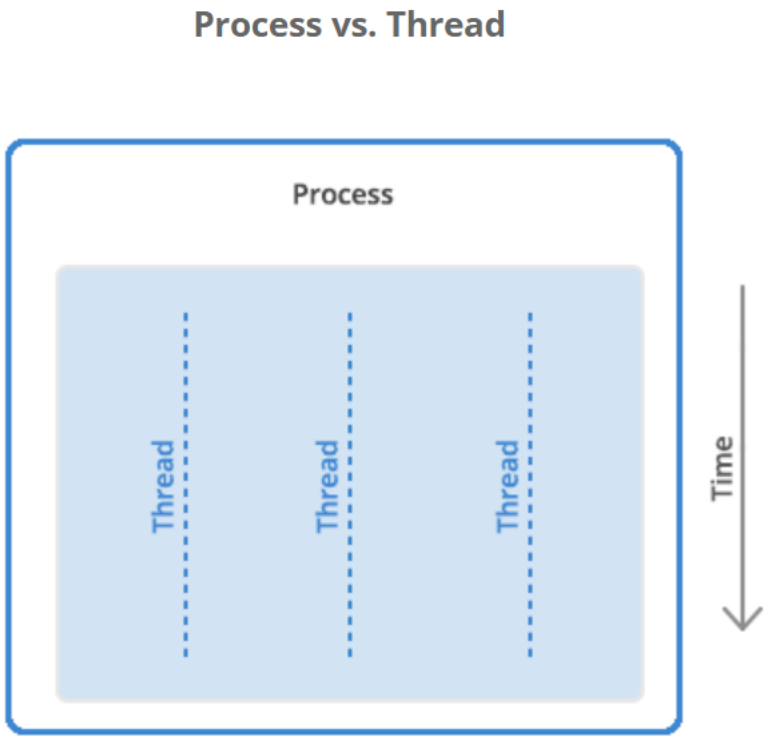
Text editor – one thread is accepting your typing, one thread is checking your spelling, one thread is occasionally saving your document. etc…

Video game – one thread is tracking your health, one thread is tracking your position, one thread is tracking your ammo, etc…

You – one thread is breathing, one thread is keeping your heart beating, one thread is falling asleep, one thread is halfway listening, etc…

Threads

A thread is the unit of execution within a process. A process can have anywhere from just one thread to many threads.

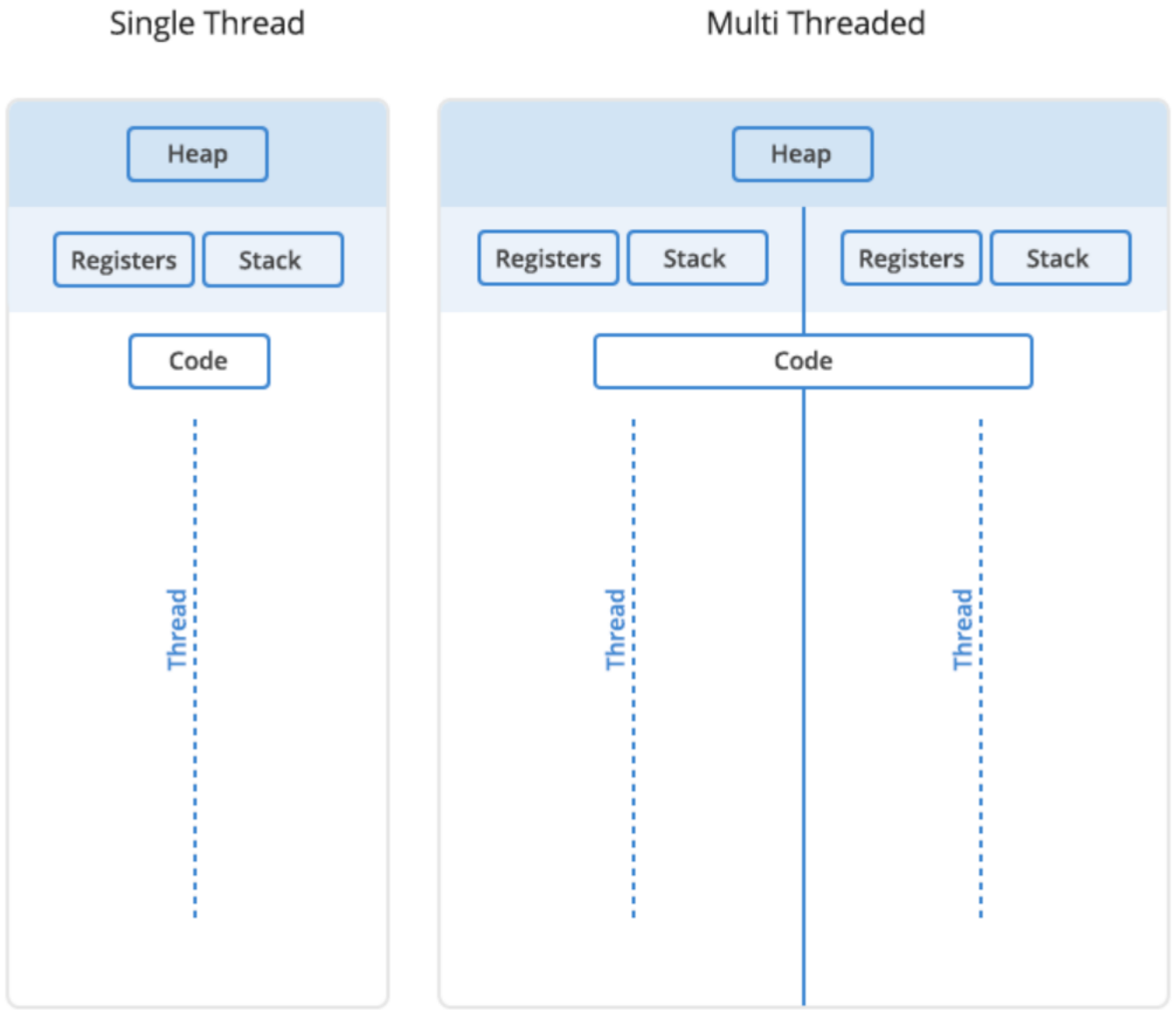


Threads

When a process starts, it is assigned memory and resources. Each thread in the process shares that memory and resources.

In single-threaded processes, the process contains one thread. The process and the thread are one and the same, and there is only one thing happening.

In multithreaded processes, the process contains more than one thread, and the process is accomplishing a number of things at the same time.


Single Thread
Heap
Registers
Stack
Code
Thread
Multi Threaded
Heap
Registers
Stack
Registers
Stack
Code
Thread
Thread

Threads

Two types of memory are available to a process or a thread

- the stack
- the heap

It is important to distinguish between these two types of process memory because

- each thread will have its own stack
- all the threads in a process will share the heap

Threads

must include `<thread>`

must be compiled with `-pthread`

```
#makefile for multithreaded C++ program
SRC = threadDemo.cpp
OBJ = $(SRC:.cpp=.o)
EXE = $(SRC:.cpp=.e)

CFLAGS = -g -std=c++11 -pthread

all : $(EXE)

$(EXE): $(OBJ)
	g++ $(CFLAGS) $(OBJ) -o $(EXE)

$(OBJ) : $(SRC)
	g++ -c $(CFLAGS) $(SRC) -o $(OBJ)
```

```
g++ threadDemo.cpp -pthread -g -std=c+11
```

Threads

To construct a thread, we instantiate a thread object by calling the thread initialization constructor.

This will construct a thread object that represents a new joinable thread of execution.

The new thread of execution calls the passed in function with the passed in arguments.

```
thread t1(threadT1, "Hello");
```

```
#include <iostream>
#include <thread>

using namespace std;

void threadFunction(string msg)
{
   cout << "threadFunction says " << msg << endl;
}

int main(void)
{
   //Construct a new thread and run it
   thread t1(threadFunction, "Hello");

   return 0;
}
```

Instantiate a `thread` object named `t1` using the initialization constructor.

Pass function "`threadFunction`" to the constructor along with parameter string "`Hello`".

The `thread` constructor will call `threadFunction` with parameter "`Hello`"

```
threadFunction("Hello");
```

```
#include <iostream>
#include <thread>

using namespace std;

void threadFunction(string msg)
{
   cout << "threadFunction says "
        << msg << endl;
}

int main(void)
{
   //Construct a new thread and run it
   thread t1(threadFunction, "Hello");

   return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./a.out
terminate called without an active exception
Aborted (core dumped)
```

```
Breakpoint 1, main () at threadDemo.cpp:13
13      {
(gdb) n
15              thread t1(threadFunction, "Hello");
(gdb)
[New Thread 0x7ffff6f4f700 (LWP 13497)]
threadFunction says Hello
[Thread 0x7ffff6f4f700 (LWP 13497) exited]
18              return 0;
(gdb)
15              thread t1(threadFunction, "Hello");
(gdb)
terminate called without an active exception

Thread 1 "a.out" received signal SIGABRT, Aborted.
0x00007ffff728e428 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:54
54      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb)

Program terminated with signal SIGABRT, Aborted.
The program no longer exists.
```

Threads

After the new thread has been launched,

```
thread t1(threadFunction, "Hello");
```

the initial thread (`main`) continues execution.

It does not wait for the new thread to finish and ends the program—possibly before the new thread has had a chance to run.

We need to add a call to `thread` member function `join` which will cause the calling thread (`main`) to wait for the thread associated with the `thread` object `t1`

```cpp
#include <iostream>
#include <thread>

using namespace std;

void threadFunction(string msg)
{
    cout << "threadFunction says "
         << msg << endl;
}

int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");
    t1.join();
    return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./a.out
threadFunction says Hello
student@cse1325:/media/sf_VM$
```

```
Breakpoint 1, main () at threadDemo.cpp:13
13      {
(gdb) n
15              thread t1(threadFunction, "Hello");
(gdb) n
[New Thread 0x7ffff6f4f700 (LWP 13520)]
threadFunction says Hello
[Thread 0x7ffff6f4f700 (LWP 13520) exited]
18              t1.join();
(gdb) n
20              return 0;
(gdb) n
15              thread t1(threadFunction, "Hello");
(gdb) n
21      }
(gdb) n
__libc_start_main (main=0x4011d5 <main()>, argc=1, argv=0x7fffffffe1f8,
    init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized out
    stack_end=0x7fffffffe1e8) at ../csu/libc-start.c:325
325     ../csu/libc-start.c: No such file or directory.
(gdb) n
[Inferior 1 (process 13516) exited normally]
```

Threads

An initialized thread object represents an active thread of execution and is joinable and has a unique thread id which we can obtain by calling `thread` member function `get_id()`.

```
thread t1(threadT1, "Hello");
thread t2(threadT2, "Hello");
thread t3(threadT3, "Hello");
thread t4(threadT4, "Hello");
thread t5(threadT5, "Hello");

cout << "t1's id is "
     << t1.get_id() << endl;
cout << "t2's id is "
     << t2.get_id() << endl;
cout << "main's id is "
     << this_thread::get_id()
     << endl;
```

```
t1's id is 140390222829312
t2's id is 140390214436608
main's id is 140390240180032
threadT5 says Hello     T5 i = 1
threadT4 says Hello     T4 i = 2
threadT3 says Hello     T3 i = 3
threadT2 says Hello     T2 i = 4
threadT1 says Hello     T1 i = 5
```

Threads

We can also ask the thread pointed at by `this` to give us its id.

```
void threadFunction(int x)
{
	cout << "My id = " << this_thread::get_id() << endl;
}

int main(void)
{
	int x = 0;
	thread::id main_tid = this_thread::get_id();
	thread t1(threadFunction, x);
	cout << "t1's id = " << t1.get_id() << endl;
	cout << "main's id = " << main_tid << endl;
	t1.join();

	return 0;
}
```

```
t1's id = 139717741209344
main's id = 139717759383360
My id = 139717741209344
```

Threads

A default-constructed (non-initialized) thread object is not joinable.

```
thread t6();

cout << "t6's id is "
     << t6.get_id()
     << endl;

t6.join();
```

```
student@cse1325:/media/sf_VM$ g++ threadDemo.cpp -g -std=c++11 -pthread
threadDemo.cpp: In function 'int main()':
threadDemo.cpp:53:30: error: request for member 'get_id' in 't6', which is of no
n-class type 'std::thread()'
  cout << "t6's id is " << t6.get_id() << endl;
                              ^

threadDemo.cpp:62:5: error: request for member 'join' in 't6', which is of non-c
lass type 'std::thread()'
  t6.join();
     ^
```

Threads

The act of calling `join()` cleans up any storage associated with the thread.

The `thread` object is no longer associated with the now-finished `thread` - it isn't associated with any `thread`.

This means that you can call `join()` only once for a given `thread`.

Once you've called `join()`, the `thread` object is no longer joinable.

```
int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");
    t1.join();
    t1.join();
    return 0;
}
```

```
terminate called after throwing an instance of
'std::system_error'
  what():  Invalid argument
Aborted (core dumped)
```

Threads

The arguments passed to the thread's function are passed by copy by default.

```
void threadFunction(int x)
{
	x++;
	cout << "x = " << x << endl;
}

int main(void)
{
	int x = 0;

	cout << "x before = " << x << endl;
	thread t1(threadFunction, x);
	cout << "x after = " << x << endl;

	t1.join();

	return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./thread1Demo.e
x before = 0
x after = 0
x = 1
```

Threads

By default, the arguments are *copied* into internal storage where they can be accessed by the newly created thread of execution, even if the corresponding parameter in the function is expecting a reference.

```
void threadFunction(int &x)
{
	x++;
	cout << "x = " << x << endl;
}

int main(void)
{
	int x = 0;

	cout << "x before = " << x << endl;
	thread t1(threadFunction, x);
	cout << "x after = " << x << endl;

	t1.join();

	return 0;
}
```

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 -pthread thread1Demo.cpp -o thread1Demo.o
In file included from thread1Demo.cpp:3:0:
/usr/include/c++/7/thread: In instantiation of ‘struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >’:
/usr/include/c++/7/thread:127:22:   required from ‘std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(int&); _Args = {int&}]’
thread1Demo.cpp:21:29:   required from here
/usr/include/c++/7/thread:240:2: error: no matching function for call to ‘std::thread::_Invoker<std::tuple<void (*)(int&), int>
>::_M_invoke(std::thread::_Invoker<std::tuple<void (*)(int&), int> >::_Indices)’
  operator()()
  ^~~~~~~~
/usr/include/c++/7/thread:231:4: note: candidate: template<long unsigned int ..._Ind> decltype (std::__invoke((_S_declval<_Ind>)()...)) std::thread::_Invoker<_Tuple>::_M_invoke(std::_Index_tuple<_Ind ...>) [with long unsigned int ..._Ind = {_Ind ...}; _Tuple = std::tuple<void (*)(int&), int>]
    _M_invoke(_Index_tuple<_Ind...>)
    ^~~~~~~~~
/usr/include/c++/7/thread:231:4: note:   template argument deduction/substitution failed:
/usr/include/c++/7/thread: In substitution of ‘template<long unsigned int ..._Ind> decltype (std::__invoke(_S_declval<_Ind>()...)) std::thread::_Invoker<std::tuple<void
(*)(int&), int> >::_M_invoke<_Ind ...>(std::_Index_tuple<_Ind1 ...>) [with long unsigned int ..._Ind = {0, 1}]’:
/usr/include/c++/7/thread:240:2:   required from ‘struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >’
/usr/include/c++/7/thread:127:22:   required from ‘std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(int&); _Args = {int&}]’
thread1Demo.cpp:21:29:   required from here
/usr/include/c++/7/thread:233:29: error: no matching function for call to ‘__invoke(std::__tuple_element_t<0, std::tuple<void (*)(int&), int> >, std::__tuple_element_t<1,
std::tuple<void (*)(int&), int> >)’
    -> decltype(std::__invoke(_S_declval<_Ind>()...))
                ~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~
In file included from /usr/include/c++/7/tuple:41:0,
                 from /usr/include/c++/7/bits/unique_ptr.h:37,
                 from /usr/include/c++/7/memory:80,
                 from /usr/include/c++/7/thread:39,
                 from thread1Demo.cpp:3:
/usr/include/c++/7/bits/invoke.h:89:5: note: candidate: template<class _Callable, class ... _Args> constexpr typename std::__invoke_result<_Functor, _ArgTypes>::type
std::__invoke(_Callable&&, _Args&& ...)
     __invoke(_Callable&& __fn, _Args&&... __args)
     ^~~~~~~~
/usr/include/c++/7/bits/invoke.h:89:5: note:   template argument deduction/substitution failed:
/usr/include/c++/7/bits/invoke.h: In substitution of ‘template<class _Callable, class ... _Args> constexpr typename std::__invoke_result<_Functor, _ArgTypes>::type
std::__invoke(_Callable&&, _Args&& ...) [with _Callable = void (*)(int&); _Args = {int}]’:
/usr/include/c++/7/thread:233:29:   required by substitution of ‘template<long unsigned int ..._Ind> decltype (std::__invoke(_S_declval<_Ind>()...))
std::thread::_Invoker<std::tuple<void (*)(int&), int> >::_M_invoke<_Ind ...>(std::_Index_tuple<_Ind1 ...>) [with long unsigned int ..._Ind = {0, 1}]’
/usr/include/c++/7/thread:240:2:   required from ‘struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >’
/usr/include/c++/7/thread:127:22:   required from ‘std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(int&); _Args = {int&}]’
thread1Demo.cpp:21:29:   required from here
/usr/include/c++/7/bits/invoke.h:89:5: error: no type named ‘type’ in ‘struct std::__invoke_result<void (*)(int&), int>’
makefile:14: recipe for target 'thread1Demo.o' failed
make: *** [thread1Demo.o] Error 1
```

Threads

To "pause" a thread's execution, it can be made to sleep by calling `sleep_for` with a parameter of a typedef from the `chrono` time library.

```
this_thread::sleep_for(chrono::seconds(10));
```

Use `#include <chrono>` in order to have access to `chrono`'s typedefs `hours`, `minutes` and `seconds`.

```
int main(void)
{
  thread t1(threadT1, "I'm thread T1");
  thread t2(threadT2, "I'm thread T2");
  thread t3(threadT3, 5);

  cout << "main() is napping for 5 seconds" << endl;
  this_thread::sleep_for(chrono::minutes(5));
  cout << "main() says I'M AWAKE " << endl;

  // Makes the main thread wait for the new thread to finish execution
  t1.join();
  t2.join();
  t3.join();

  return 0;
}
```

```
void threadT3(int seconds)
{
  cout << "threadT3 is napping for " << seconds
       << "seconds" << endl;
  this_thread::sleep_for(chrono::seconds(seconds));
  cout << "threadT3 says I'M AWAKE " << endl;
}
```


student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ ./threadsleepDemo.e
student@cse1325: /me...
10:26

Threads

Reentrant Function/Algorithm

A reentrant function/algorithm behaves correctly if called simultaneously by several threads.

Functions that are callable by several threads must be made reentrant. To make a function reentrant might require changes to the function interface or to the implementation.

Functions that access global state, like memory or files, have reentrance problems.

In C++, functions can be made reentrant by using `mutex`.

Threads

`mutex`

A `mutex` is a *lockable object* that is designed to signal when critical sections of code need exclusive access and prevents other threads with the same protection from executing concurrently and access the same memory locations.

Locking a `mutex` prevents other threads from locking it (exclusive access) until it is unlocked.

Must

```
#include <mutex>
```

```
int main(void)
{
  srand(time(NULL));

  thread asterisk_thread(print_it, 20, '*');
  thread dollar_thread(print_it, 20, '$');

  asterisk_thread.join();
  dollar_thread.join();

  return 0;
}
```

```
void print_it(int NumberOfCharsToPrint, char charToPrint)
{
  for (int i = 0; i < NumberOfCharsToPrint; ++i)
  {
     cerr << charToPrint;

     this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
  }

  cout << '\n';
}
```


student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ ./mutexDemo.e
student@cse1325: /me...
11:00

```
int main(void)
{
  srand(time(NULL));

  thread asterisk_thread(print_it, 20, '*');
  thread dollar_thread(print_it, 20, '$');

  asterisk_thread.join();
  dollar_thread.join();

  return 0;
}
```

```
void print_it(int NumberOfCharsToPrint, char charToPrint)
{
  for (int i = 0; i < NumberOfCharsToPrint; ++i)
  {
     cerr << charToPrint;

     this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
  }

  cout << '\n';
}
```

Why `cerr` and not `cout`?


Terminal - student@maverick: /media/sf_VM/CSE1325
File Edit View Terminal Tabs Help
student@maverick:/media/sf_VM/CSE1325$ ./mutex1Demo.

Threads

Threads don't do a very good job of sharing a resource.

`asterisk_thread` and `dollar_thread` were sharing the standard output stream and were not able to evenly or consistently take turns.

What if those two threads had been sharing a file and were tasked with updating that file?

The data in the file would be in a different order every time the process ran.

`mutex`

We can use a synchronization primitive called a `mutex` (*mut*ual *ex*clusion) to help alleviate this sharing issue.

Before accessing a shared resource, you lock the mutex associated with that resource and, when you have finished accessing the data structure, you unlock the mutex.

The Thread Library ensures that once one thread has locked a specific mutex, all other threads that try to lock the same mutex have to wait until the thread that successfully locked the mutex unlocks it.

mutex

Create a `mutex` by constructing an instance of `std::mutex`

```
mutex mtx;
```

If not `using namespace std`, then this would need to be `std::mutex`

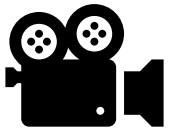
lock it with a call to the member function `lock()`

```
mtx.lock();
```

and unlock it with a call to the member function `unlock()`

```
mtx.unlock();
```

```
mutex mtx;              // construct mutex object

void print_it(int NumberOfCharsToPrint, char charToPrint)
{
      mtx.lock();

      for (int i = 0; i < NumberOfCharsToPrint; ++i)
      {
            cerr << charToPrint;
            this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
      }

      cout << '\n';

      mtx.unlock();
}
```



student@cse1325: /media/sf_VM

File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ ./mutexDemo.e
$
```

student@cse1325: /me...

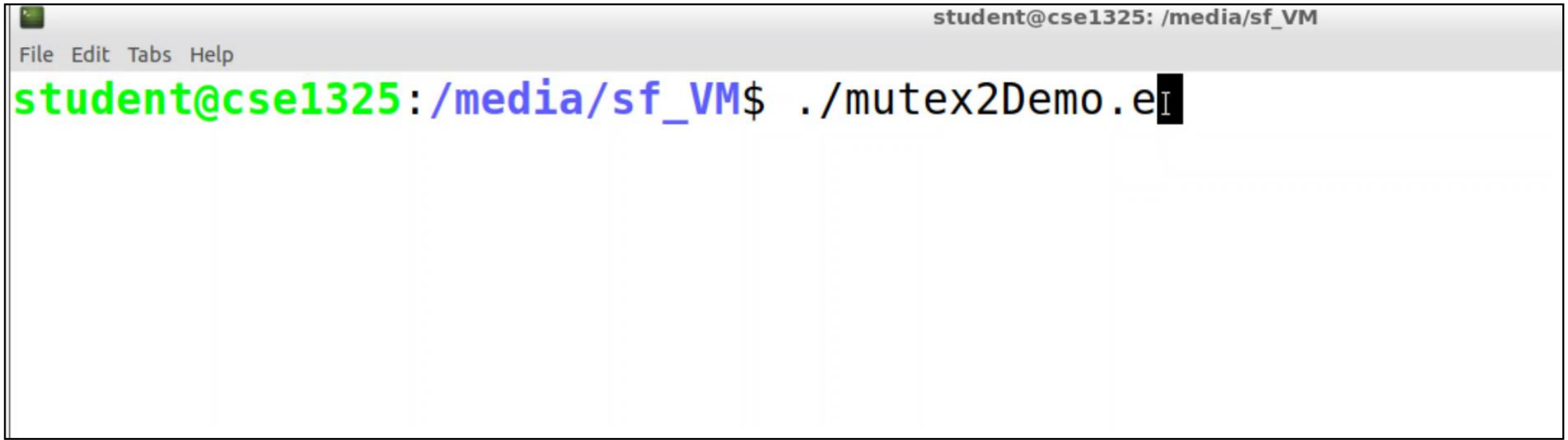
12:31

```
void print_it(char charToPrint)
{
   cerr << charToPrint;

   cerr << charToPrint;
}
```

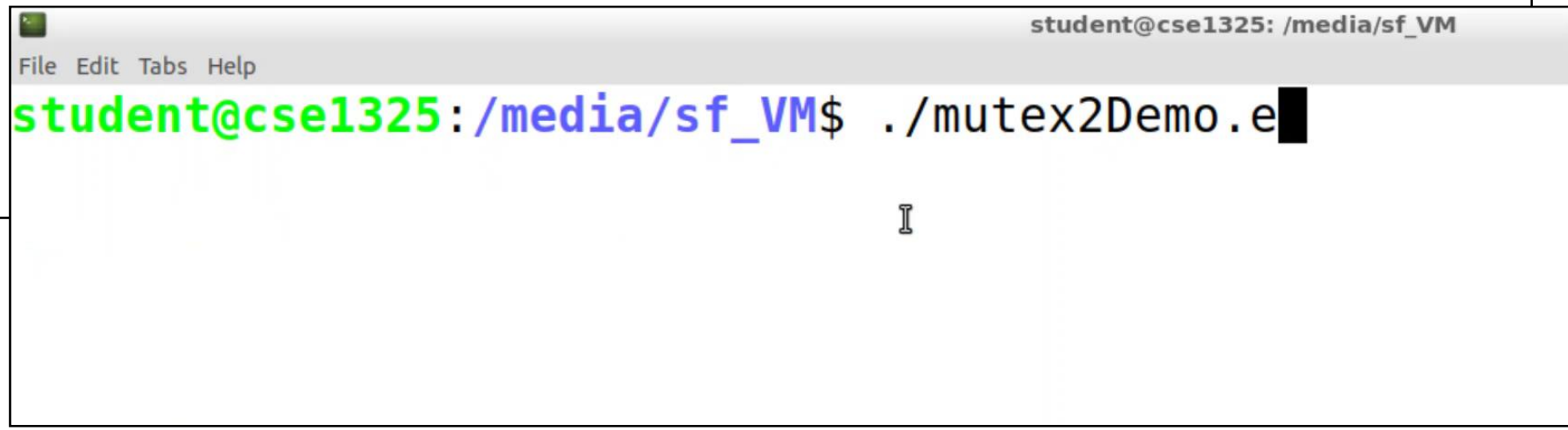
```
int main(void)
{
  vector<thread> Alphabet;

  for(int i = 0; i < 26; ++i)
  {
     char Letter = i+65;
     Alphabet.push_back(std::thread(print_it, Letter));
  }

  for (thread& it : Alphabet)
  {
     it.join();
  }

  cout << endl;

  return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./mutex2Demo.e
GGHHIIFFJJKKLLMMEENNOOPPQQRRSSDDTTUUVVWWXXYYZZCCBBAA
student@cse1325:/media/sf_VM$
```

```
void print_it(char charToPrint)
{
   this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

   cerr << charToPrint;

   this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

   cerr << charToPrint;
}
```



```
void print_it(char charToPrint)
{
   mtx.lock();

   this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

   cerr << charToPrint;

   this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

   cerr << charToPrint;

   mtx.unlock();

}
```



```
int main(void)
{
   vector<thread> Numbers;
   srand(time(NULL));

   for(int i = 10; i < 100; ++i)
   {
      Numbers.push_back(std::thread(print_it, i));
   }

   for(thread& it : Numbers)
   {
      it.join();
   }

   return 0;
}
```

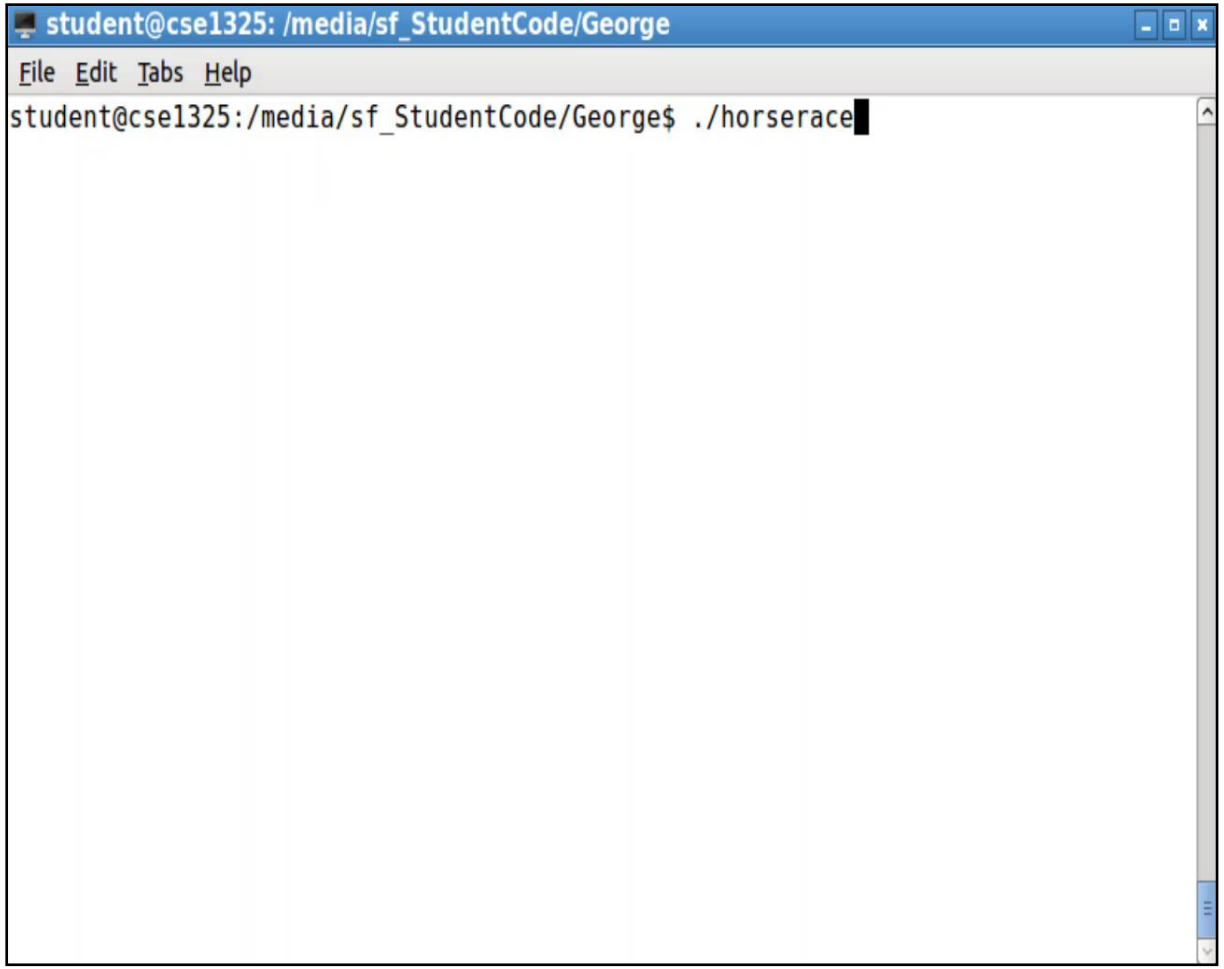
```
void print_it(int NumberToPrint)
{
   static int counter = 1;

   cerr << NumberToPrint << '-';

   this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

   if (!(counter++ %10))
      cout << endl;

}
```

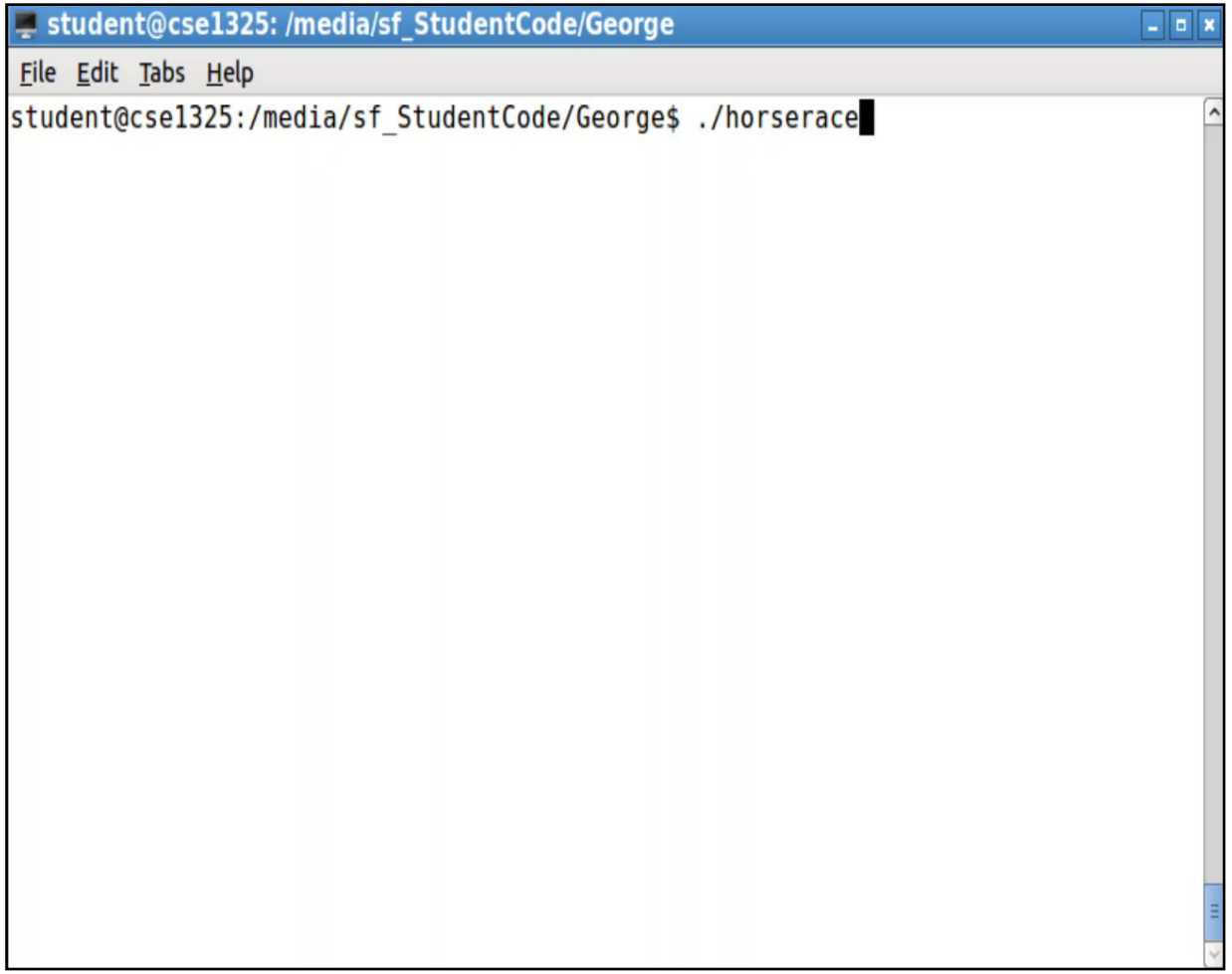
```
student@cse1325:/media/sf_VM$ ./mutex3Demo.e
16-17-15-18-19-14-20-21-22-23-13-24-26-27-28-25-29-30-31-12-32-33-34-35-36-40-37-39-4
2-41-38-59-48-56-43-52-58-55-46-49-60-11-53-54-45-57-62-44-47-50-61-51-63-64-65-66-67
-68-69-70-71-72-73-74-75-76-77-78-79-80-81-82-83-84-85-86-10-87-88-89-90-91-92-93-94-
95-96-97-98-99-



student@cse1325:/media/sf_VM$
```


student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ ./mutex3Demo.e
student@cse1325: /me...
Screenshot.txt
17:21

```
mutex Kitty;

void print_it(int NumberToPrint)
{
   static int counter = 1;

   Kitty.lock();

   cerr << NumberToPrint << '-';

   this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

   if (!(counter++ %10))
     cout << endl;

   Kitty.unlock();
}
```


student@cse1325: /
File Edit Tabs Help
student@cse1325:/media/sf_VM$ ./mutex3Demo.e


student@cse1325: /media/sf_StudentCode/George
File Edit Tabs Help
student@cse1325:/media/sf_StudentCode/George$ ./horserace


student@cse1325: /media/sf_StudentCode/George
File Edit Tabs Help
student@cse1325:/media/sf_StudentCode/George$ ./horserace

Multithreading Vocabulary

Stack – Scratch memory for a thread

Heap – Memory shared by all threads for dynamic allocation

Concurrency – Performing 2 or more algorithms simultaneously

Process – A self-contained execution environment including its own memory space

Thread – An independent path of execution within a process, running concurrently with other threads within a shared memory space

Reentrant – An algorithm that can be paused while executing and then safely executed by a different thread

mutex – A mutual exclusion object that prevents two properly written threads from concurrently accessing a critical resource

Standard Template Library (STL)
C++ Standard Library

The Standard Library defines powerful, template-based, reuseable components that implement many common data structures and algorithms used to process those data structures.

Three key components of the Standard Library

- Containers (templatized data structures)
- Iterators
- Algorithms

Standard Template Library (STL)
C++ Standard Library

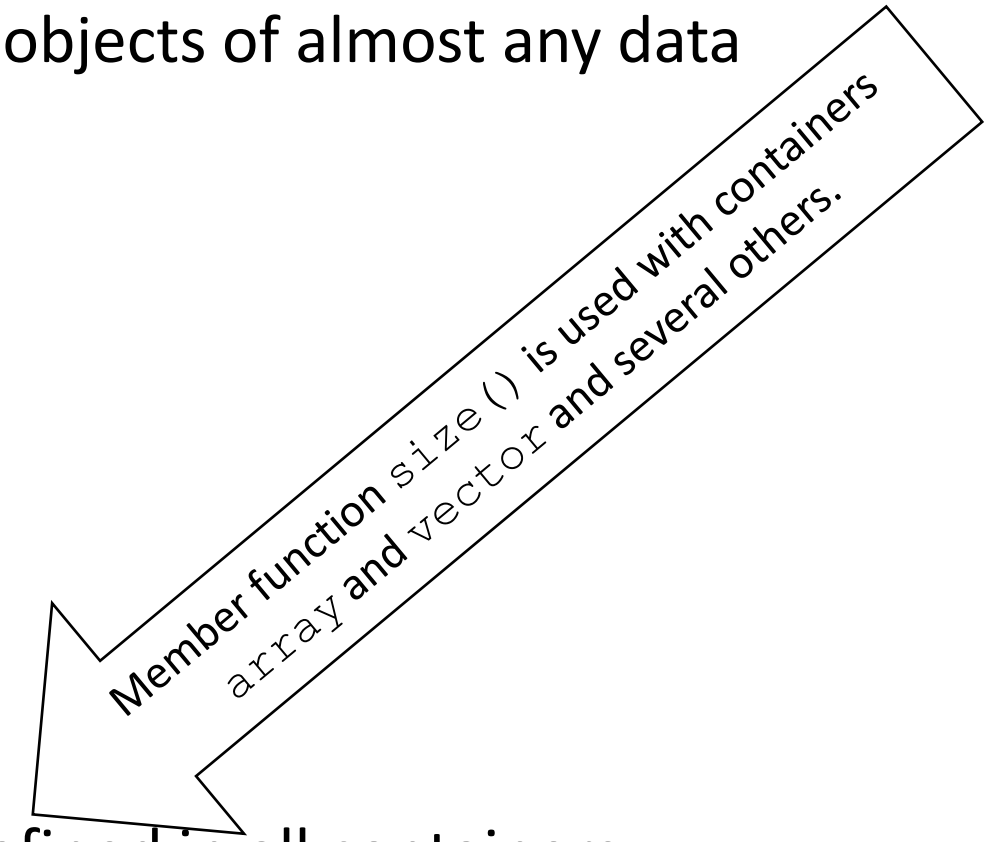
Containers are data structures capable of storing objects of almost any data type.

Three styles of container classes

- first-class containers
- container adapters
- near containers

Each container has associated member functions

- a subset of those member functions are defined in all containers



Container Classes

- first-class containers
  - sequence containers
  - associative containers
- container adapters
  - constrained version of sequence containers
- near containers
  - containers that exhibit some but not all capabilities of the first-class containers
  - used for performing high-speed mathematical vector operations

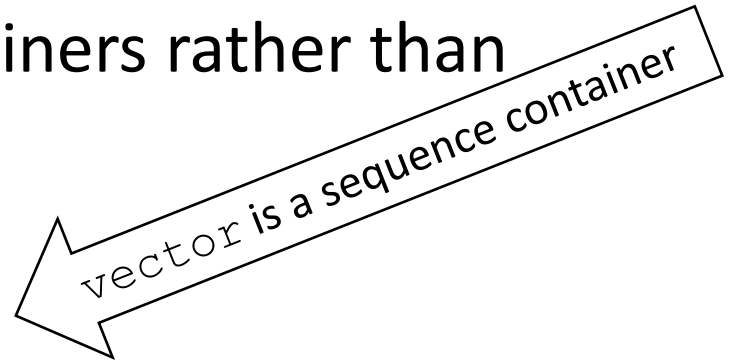
Container Classes

When an object is inserted into a container, a copy of the object is made.

The object should provide a copy constructor and copy assignment operator (custom or default).

It is usually preferable to reuse Standard Library containers rather than developing custom templatized data structures.

`vector` is typically satisfactory for most applications.



Associative Containers

The associative containers provide direct access to store and retrieve elements via keys (often called search keys).

The four ordered associative containers are

```
multiset,
set,
multimap
map
```

Each of these maintains its keys in sorted order.

Associative Containers

Class `map` provides operations for manipulating values associated with keys (these values are sometimes referred to as mapped values).

The primary difference between a `multimap` and a `map` is that a `multimap` allows duplicate keys with associated values to be stored and a `map` allows only *unique keys* with associated values.

In addition to the common container member functions, *ordered associative containers* also support several other member functions that are specific to associative containers.

`map`

The `map` *associative container* (from header `<map>`) performs fast storage and retrieval of *unique keys* and *associated values*.

The elements' ordering is determined by a comparator function object.

For example, in an integer map, elements can be sorted in ascending order by ordering the keys with comparator function object `less<int>`.

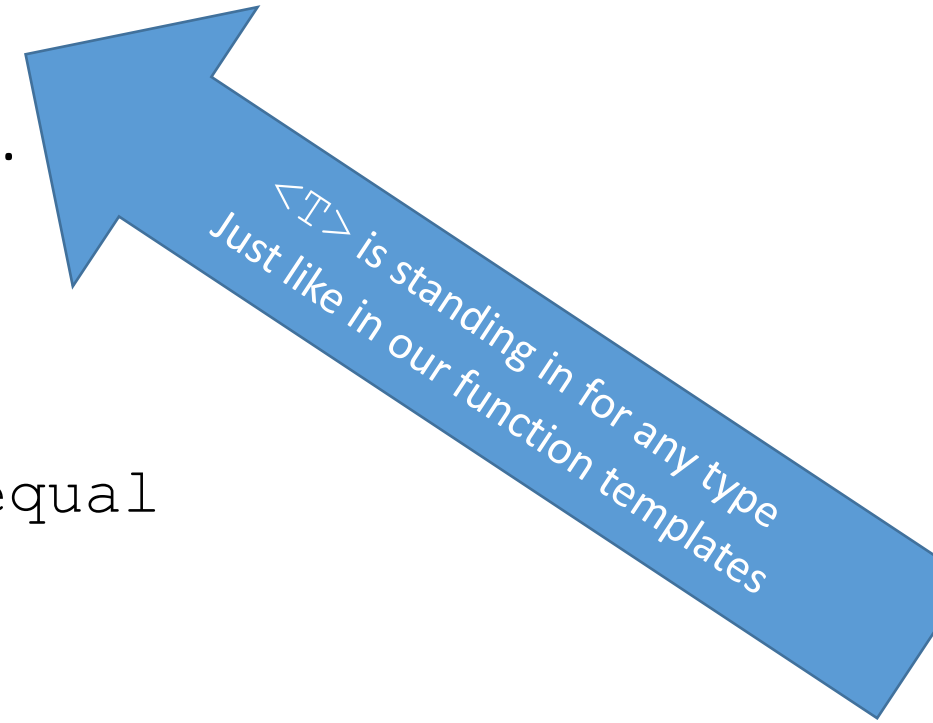
No two elements in the container can have equivalent *keys*.

map

The data type of the keys in all ordered associative containers must support comparison based on the comparator function object—keys sorted with `less<T>` must support comparison with operator <.

The default comparator function object is `less<T>` which sorts the key using <.

The comparator function object `greater<T>` sorts using >.



Other builtin comparator function objects

`equal_to`	`greater`	`less`
`not_equal_to`	`greater_equal`	`less_equal`

It is possible to create your own comparator function object.

map

A **map** is a set where each element is a pair, called a key/value pair.

The key is used for sorting and indexing the data and must be unique.

The value is the actual data.

Duplicate keys are *not* allowed—a single value can be associated with each key.

This is called a one-to-one mapping.

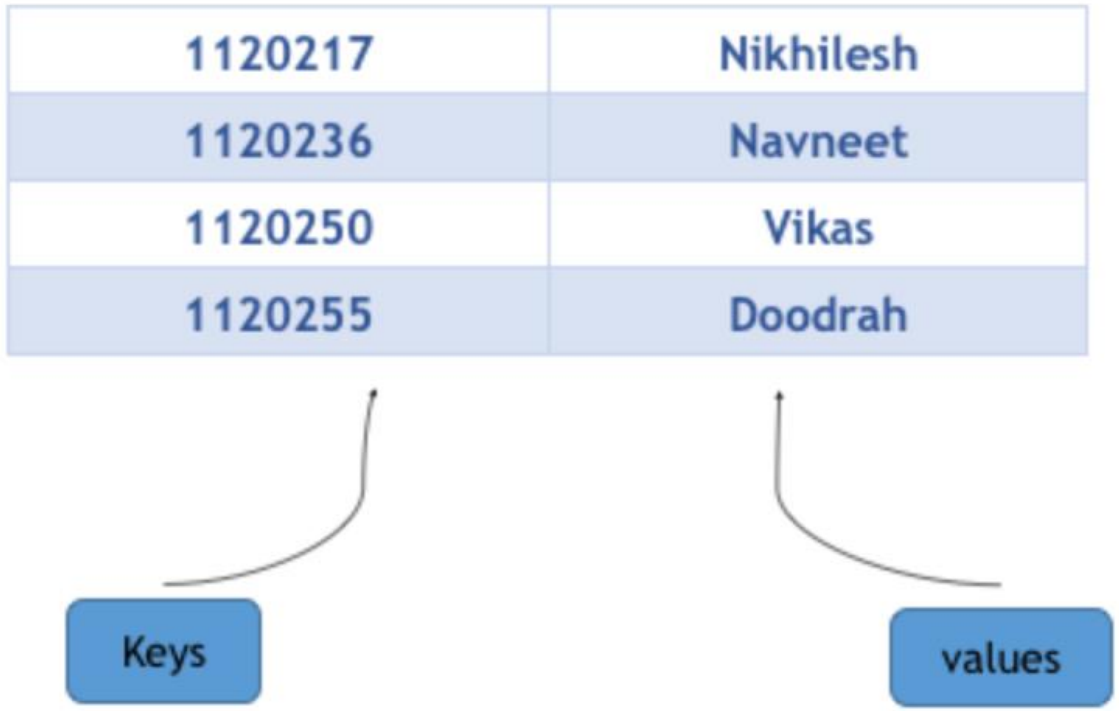
map

A map of students where **id number** is the key and **name** is the value can be represented graphically as

Notice that keys are arranged in ascending order.

Maps always arrange its keys in sorted order.

Here the keys are of string type; therefore, they are sorted lexicographically.



map

For example, a company that uses unique employee numbers, such as 100, 200 and 300, might have a map that associates employee numbers with their telephone extensions—4321, 4115 and 5217, respectively.

With a map you specify the key and get back the associated data quickly.

Providing the key in a map's subscript operator [ ] locates the value associated with that key in the map.

map

To create a map

```
map<int, double, less<int>> MyMap;
```

`MyMap` is the name of our `map` object.

Key type is `int`

Type of key's associated value is `double`.

map's elements will be sorted in ascending order using the function object `less<int>`. Ascending order is the default for a `map` so `less<int>` can be omitted.

map

To insert a new key-value pair

```
MyMap.insert(make_pair(15, 2.7));
```

`MyMap` is the name of the map and `insert()` is a member function of map

`insert()` adds a new key-value pair to the `map`

`make_pair()` is a member function of `map` that uses the types specified for the keys and values to create a key-value pair object.

```
map<int, double> MyMap;

MyMap.insert(make_pair(15, 2.7));
MyMap.insert(make_pair(30, 111.11));
MyMap.insert(make_pair(5, 1010.1));
MyMap.insert(make_pair(10, 22.22));
MyMap.insert(make_pair(25, 33.333));
MyMap.insert(make_pair(5, 77.54));
MyMap.insert(make_pair(20, 9.345));
MyMap.insert(make_pair(15, 99.3));
```

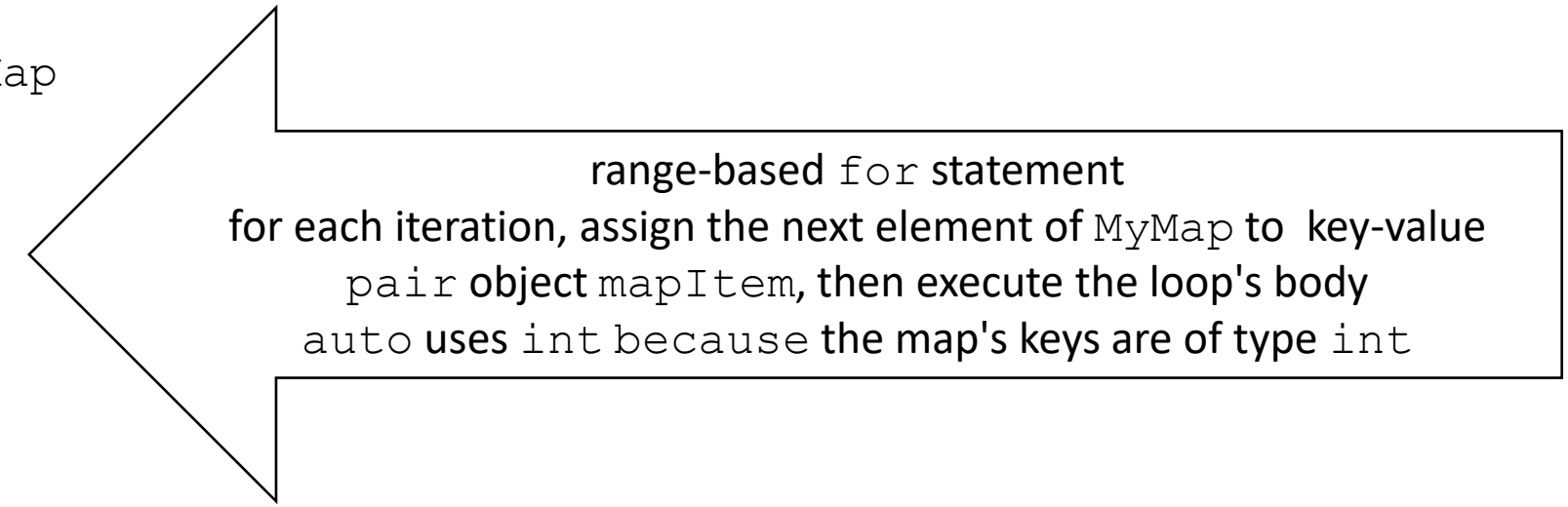
dup ignored – no update

dup ignored – no update

```
(gdb) p MyMap
$5 = std::map with 6 elements = {
  [5] = 1010.1,
  [10] = 22.219999999999999,
  [15] = 2.7000000000000002,
  [20] = 9.3450000000000006,
  [25] = 33.332999999999998,
  [30] = 111.11
}
```

mapDemo.cpp

```
map<int, double> MyMap;

MyMap.insert(make_pair(15, 2.7));
MyMap.insert(make_pair(30, 111.11));
MyMap.insert(make_pair(5, 1010.1));
MyMap.insert(make_pair(10, 22.22));
MyMap.insert(make_pair(25, 33.333));
MyMap.insert(make_pair(5, 77.54)); // dup ignored
MyMap.insert(make_pair(20, 9.345));
MyMap.insert(make_pair(15, 99.3)); // dup ignored

cout << "MyMap contains:\nKey\tValue\n";

// walk through elements of MyMap
for (auto mapItem : MyMap)
```



```
(gdb) p MyMap
$5 = std::map with 6 elements = {
  [5] = 1010.1,
  [10] = 22.219999999999999,
  [15] = 2.7000000000000002,
  [20] = 9.3450000000000006,
  [25] = 33.332999999999998,
  [30] = 111.11
}
```

```
(gdb) p mapItem
$4 = {
  first = 5,
  second = 1010.1
}
```

mapItem is an iterator object of type key-value *MyMap*

```
cout << "MyMap contains:\nKey\tValue\n";

for (auto mapItem : MyMap)
{
   cout << mapItem.first << '\t' << mapItem.second << '\n';
}
```

```
MyMap contains:
Key   Value
5     1010.1
```

mapDemo.cpp

```
// walk through elements of MyMap
for (auto mapItem : MyMap)
{
   cout << mapItem.first << '\t'
        << mapItem.second << '\n';
}

MyMap[25] = 9999.99; // use subscripting to change value for key 25
MyMap[40] = 8765.43; // use subscripting to insert value for key 40

cout << "\nAfter subscript operations, \nMyMap contains:\nKey\tValue\n";

for (auto mapItem : MyMap)
{
   cout << mapItem.first << '\t'
        << mapItem.second << '\n';
}
```

```
(gdb) p MyMap
$1 = std::map with 6 elements = {[5] = 10
22.219999999999999,
  [15] = 2.7000000000000002, [20] = 9.345
  [25] = 33.332999999999998, [30] = 111.1

30 MyMap[25] = 9999.99; // use [] to chan
31 MyMap[40] = 8765.43; // use [] to inse

(gdb) p MyMap
$2 = std::map with 7 elements = {[5] = 1010.1, [1
  [15] = 2.7000000000000002, [20] = 9.345000000000
  [25] = 9999.9899999999998, [30] = 111.11, [40] =
```

```
MyMap contains:
Key  Value
5    1010.1
10   22.22
15   2.7
20   9.345
25   33.333
30   111.11
```

```
After [] operations,
MyMap contains:
Key  Value
5    1010.1
10   22.22
15   2.7
20   9.345
25   9999.99
30   111.11
40   8765.43
```

map2Demo.cpp

```
#include <iostream>
#include <map>
#include <sstream>

using namespace std;

string MapToString(map<string, double> & maptostring)
{
  ostringstream MapString;

  for (auto mapItem : maptostring)
  {
    MapString << mapItem.first << "\t" << mapItem.second << endl;
  }

  return MapString.str();
}

int main (void)
{
  map<string,double> GradeMap{ {"A",78.52} , {"B",85.94} , {"C",90.57} };

  string MapString = MapToString(GradeMap);

  cout << MapString; // Print the result
}
```

map2Demo.cpp

```
int main (void)
{
  map<string,double> GradeMap{ {"A",78.52} , {"B",85.94} , {"C",90.57} };

  string MapString = MapToString(GradeMap);

  cout << MapString; // Print the result
}
```

```
(gdb) p GradeMap
$2 = std::map with 3 elements =
  ["A"] = 78.519999999999996,
  ["B"] = 85.939999999999998,
  ["C"] = 90.569999999999993
}

(gdb) p MapString
$3 = "A\t78.52\nB\t85.94\nC\t90.57\n"

A   78.52
B   85.94
C   90.57
```

`map<string, double>`

Key is a string – letter grade
Value is a double – numeric grade

```
string MapToString(map<string, double> & maptostring)
{
  ostringstream MapString;

  for (auto mapItem : maptostring)
  {
    MapString << mapItem.first << "\t" << mapItem.second << endl;
  }

  return MapString.str();
}
```

pass by reference

#include <sstream>

range based for

```
A 78.52
B 85.94
C 90.57
```

```
25 string MapString = MapToString(GradeMap);

(gdb) step

MapToString (maptostring=std::map with 3
elements = {...}) at map2Demo.cpp:10
```

```
(gdb) p mapItem
$6 = {
  first = "A",
  second = 78.519999999999996}

(gdb) p mapItem
$8 = {
  first = "B",
  second = 85.939999999999998}

(gdb) p mapItem
$11 = {
  first = "C",
  second = 90.569999999999993}
```

map2Demo.cpp

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

using namespace std;

int main(void)
{
    map<string, int> mapOfWords;
    string SearchWord;
    int SearchValue;
```

`mapOfWords` is a `map` with `string` keys and `int` values.

`SearchWord` will be used to search for keys in our map
`SearchValue` will be used to search for values in our map

`map3Demo.cpp`

`map` function `empty()`

```
if (mapOfWords.empty())
  cout << "Our map is empty" << endl;

if (!mapOfWords.empty())
  cout << "Our map is NOT empty" << endl;
```

returns `true` if `map` size is `0`, else returns `false`

Never throws an exception

`map3Demo.cpp`

```
if (mapOfWords.empty())
   cout << "Our map is empty" << endl;

cout << "Inserting 'earth'" << endl;
mapOfWords.insert(make_pair("earth", 2));

cout << "Inserting 'moon'" << endl;
mapOfWords.insert(make_pair("moon", 4));

if (!mapOfWords.empty())
   cout << "Our map is NOT empty" << endl;

cout << "Inserting 'sun'" << endl;
mapOfWords["sun"] = 3;
```

```
Our map is empty

Inserting 'earth'

Inserting 'moon'

Our map is NOT empty

Inserting 'sun'

(gdb) p mapOfWords
$2 = std::map with 3
elements = {
  ["earth"] = 2,
  ["moon"] = 4,
  ["sun"] = 3
}
```

map3Demo.cpp

`map` function `size()`

```
mapOfWords.size()
```

does not take any parameters

returns the number of elements in the map

Never throws an exception

```
cout << "Key\tValue" << endl;
for (auto mapItem : mapOfWords)
{
      cout << mapItem.first << "\t" << mapItem.second << endl;
}

cout << "Our map has " << mapOfWords.size() << " elements" << endl;
```

```
Key  Value
earth    2
moon     4
sun      3
Our map has 3 elements
```

map3Demo.cpp

```
(gdb) p mapOfWords
$3 = std::map with 3 elements = {
  ["earth"] = 2,
  ["moon"] = 4,
  ["sun"] = 3
}


Breakpoint 3, main () at map3Demo.cpp:38
38        mapOfWords["earth"] = 1;


(gdb) p mapOfWords
$3 = std::map with 3 elements = {
  ["earth"] = 1,
  ["moon"] = 4,
  ["sun"] = 3
}
```

map3Demo.cpp

operator[] vs insert function

If specified key already existed in map then operator [] will silently change its value; whereas, insert will not replace an already added key. Instead, it returns if element was added or not.

```
// Will replace the value of already added key i.e. earth
mapOfWords["earth"] = 1;
```

Whereas, for insert member function,

```
// fails and returns false in object's second member
if (mapOfWords.insert(make_pair("earth", 1)).second == false)
{
   cout << "Element with key 'earth' not inserted because
            already existed" << endl;
}
```

```
(gdb) p mapOfWords
$2 = std::map with 3 elements = {["earth"] = 2, ["moon"] = 4, ["sun"] = 3}

38    mapOfWords["earth"] = 1;

(gdb) p mapOfWords
$3 = std::map with 3 elements = {["earth"] = 1, ["moon"] = 4, ["sun"] = 3}



41    if (mapOfWords.insert(make_pair("earth", 1)).second == false)

43            cout << "Element with key 'earth' not inserted because already
existed" << endl;
```

Element with key 'earth' not inserted because already existed

```
(gdb) p mapOfWords
$4 = std::map with 3 elements = {["earth"] = 1, ["moon"] = 4, ["sun"] = 3}
(gdb)
```

map3Demo.cpp

`map` function `at()`

```
mapOfWords.at(key)
```

takes in *key*

returns a reference to the key's value

If an exception is thrown, there are no changes in the container.

It throws out_of_range if *key* is not the key of an element in the map.

```
// at() will throw an exception when given key out of range
try
{
   cout << "The value for 'earth' is " << mapOfWords.at("earth") << endl;
   cout << "The value for 'venus' is " << mapOfWords.at("venus") << endl;
}
catch (out_of_range& say)
{
   cerr << "An out of range exception was thrown by "
        << say.what() << endl;
}
```

```
The value for 'earth' is 1
An out of range exception was thrown by map::at
```

operator[] vs at function

operator[] does not do range checking. If you access a key using the indexing operator [] that is not currently a part of a map, then it automatically adds a key for you

at() member function does range checking and throws an exception when you are trying to access a nonexisting element.

```
// [] operator does not check bounds and does not throw exception
cout << "The value for 'venus' is " << mapOfWords["venus"] << endl;
```

```
The value for 'venus' is 0
```

```
(gdb) p mapOfWords
$1 = std::map with 4 elements = {["earth"] = 1, ["moon"] = 4, ["sun"] = 3,
  ["venus"] = 0}
```

`map` function `find()`

```
mapOfWords.find(key)
```

takes in key

returns an iterator to the element if key is found, else returns `end`

If an exception is thrown, there are no changes in the container.

```
for (auto mapItem : mapOfWords)
{
    cout << mapItem.first << "\t" << mapItem.second << endl;
}

cout << "Enter a word to find " << endl;
cin >> SearchWord;

// Searching element in map by key.
if (mapOfWords.find(SearchWord) != mapOfWords.end())
    cout << "word '" << SearchWord << "' found" << endl;
else
    cout << "word '" << SearchWord << "' not found" << endl;

if (mapOfWords.find("mars") == mapOfWords.end())
    cout << "word 'mars' not found" << endl;
```

map3Demo.cpp

```
earth          1
moon           4
sun            3
venus          0
```

```
if (mapOfWords.find(SearchWord) != mapOfWords.end())
   cout << "word '" << SearchWord << "' found" << endl;
else
   cout << "word '" << SearchWord << "' not found" <<
endl;

if (mapOfWords.find("mars") == mapOfWords.end())
   cout << "word 'mars' not found" << endl;
```

```
Enter a word to find
moon

word 'moon' found

word 'mars' not found
```

map3Demo.cpp

`map` function `count()`

```
mapOfWords.count(key)
```

takes in key

returns `1` if key is found, else returns `0`

If an exception is thrown, there are no changes in the container.

```
cout << "Key 'mars' ";
if (mapOfWords.count("mars"))
   cout << " is part of our map" << endl;
else
   cout << " is not part of our map" << endl;


cout << "Enter a word to search for in our map ";
cin >> SearchWord;

cout << "Key '" << SearchWord << "'";

if (mapOfWords.count(SearchWord))
   cout << " is part of our map" << endl;
else
   cout << " is not part of our map" << endl;
```

earth	1
moon	4
sun	3
venus	0

```
Key 'mars'  is not part of our map
Enter a word to search for in our map earth
Key 'earth' is part of our map
```

map3Demo.cpp

`count()` vs `find()` function

Since a `map` can only have at most one key, `count()` will essentially stop after one element has been found. However, in view of more general containers such as `multimaps` and `multisets`, `find` is strictly better if you only care whether some element with this key exists, since it can really stop once the first matching element has been found.

In general, both `count()` and `find()` will use the container-specific lookup methods (tree traversal or hash table lookup), which are always fairly efficient. It's just that `count()` has to continue iterating until the end of the equal-range, whereas `find()` does not.

If you just want to find whether the key exists or not, and don't care about the value, it is better to use `count()` as it returns only an integer. `find()` returns an iterator, thus by using `count()`, you will save the construction of an iterator.

```
map<char, string> MyPets;

MyPets.insert(make_pair('A', "Appa"));
MyPets.insert(make_pair('S', "Sylvester"));
MyPets.insert(make_pair('S', "Shade"));
MyPets.insert(make_pair('J', "Josie"));

for (auto Pet : MyPets)
{
     cout << Pet.first << '\t' << Pet.second << endl;
}
```

Duplicate key - rejected!

```
student@cse1325:/media/sf_VM$ ./map4Demo.e
A     Appa
J     Josie
S     Sylvester
student@cse1325:/media/sf_VM$
```

```
map<char, string> MyPets;

MyPets['A'] = "Appa";
MyPets['S'] = "Sylvester";
MyPets['S'] = "Shade";
MyPets['J'] = "Josie";

for (auto Pet : MyPets)
{
     cout << Pet.first << '\t' << Pet.second << endl;
}
```

Duplicate key but [] overwrites!

```
student@cse1325:/media/sf_VM$ ./map4Demo.e
A     Appa
J     Josie
S     Shade
student@cse1325:/media/sf_VM$
```

```
map<string, string> MyPets;

MyPets.insert(make_pair("AP", "Appa"));
MyPets.insert(make_pair("SY", "Sylvester"));
MyPets.insert(make_pair("SH", "Shade"));
MyPets.insert(make_pair("JO", "Josie"));

for (auto Pet : MyPets)
{
     cout << Pet.first << '\t' << Pet.second << endl;
}
```

```
student@cse1325:/media/sf_VM$ ./map4Demo.e
AP    Appa
JO    Josie
SH    Shade
SY    Sylvester
student@cse1325:/media/sf_VM$
```

Standard Template Library (STL)
C++ Standard Library

The Standard Library defines powerful, template-based, reuseable components that implement many common data structures and algorithms used to process those data structures.

Three key components of the Standard Library

- Containers (templatized data structures)
- Iterators
- Algorithms

iterator

An iterator is an object that is pointing to some element in a range of elements (such as a container like `map`) that has the ability to iterate through the elements of that range.

An **iterator** is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented. With many classes (particularly lists and the associative classes), iterators are the primary way elements of these classes are accessed.

Iterators provide an easy way to step through the elements of a container class without having to understand how the container class is implemented.

iterator

An iterator is best visualized as a pointer to a given element in the container, with a set of overloaded operators to provide a set of well-defined functions:

*	Dereferencing the iterator returns the element that the iterator is currently pointing at.
++	Moves the iterator to the next element in the container. Most iterators also provide -- to move to the previous element.
==	Basic comparison operators to determine if two iterators point to the same element.
!=	To compare the values that two iterators are pointing at, dereference the iterators first, and then use a comparison operator.
=	Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is pointing at, dereference the iterator first, then use the assign operator.

iterator

Each container includes four basic member functions for use with = (assignment)

begin() returns an iterator representing the beginning of the elements in the container.

end() returns an iterator representing the element just past the end of the elements.

cbegin() returns a const (read-only) iterator representing the beginning of the elements in the container.

cend() returns a const (read-only) iterator representing the element just past the end of the elements.

iterator

`end()` returns an iterator representing the element just past the end of the elements. It points to a non-existent element that is used to determine when the end of a container is reached.

`cend()` returns a const (read-only) iterator representing the element just past the end of the elements.

`end()/cend()` do not point to the last element in the list – they point just pas the end.

This is done primarily to make looping easy: iterating over the elements can continue until the iterator reaches `end()`.

iterator

All containers provide (at least) two types of iterators:

container::iterator provides a read/write iterator

container::const_iterator provides a read-only iterator

Constant iterators cannot be used the container needs to be changed.

```
#include <iostream>
#include <vector>

int main(void)
{
    std::vector<int> vect{0,1,2,3,4,5};

    std::vector<int>::const_iterator it;

    it = vect.begin();

    while (it != vect.end())
    {
        std::cout << *it << " ";
        ++it; // and iterate to the next element
    }

    std::cout << '\n';
}
```

What if `vect.begin() + 1`?

`1 2 3 4 5`

What if `it = vect.end()-1` and `it != vector.begin()` and `-it`?

`5 4 3 2 1` – don't get 0

What if we don't use `vect.end()-1` and we just use `vect.end()`?

Print garbage and then numbers

```
0 1 2 3 4 5
```

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vect{0,1,2,3,4,5};

    std::vector<int>::iterator it;

    it = vect.begin();

    while (it != vect.end())
    {
        *it *= 3;
        std::cout << *it << " ";
        ++it; // and iterate to the next element
    }

    std::cout << '\n';
}
```

```
0 3 6 9 12 15
```

iterator

`map` uses a bidirectional iterator which means it supports ++ and –

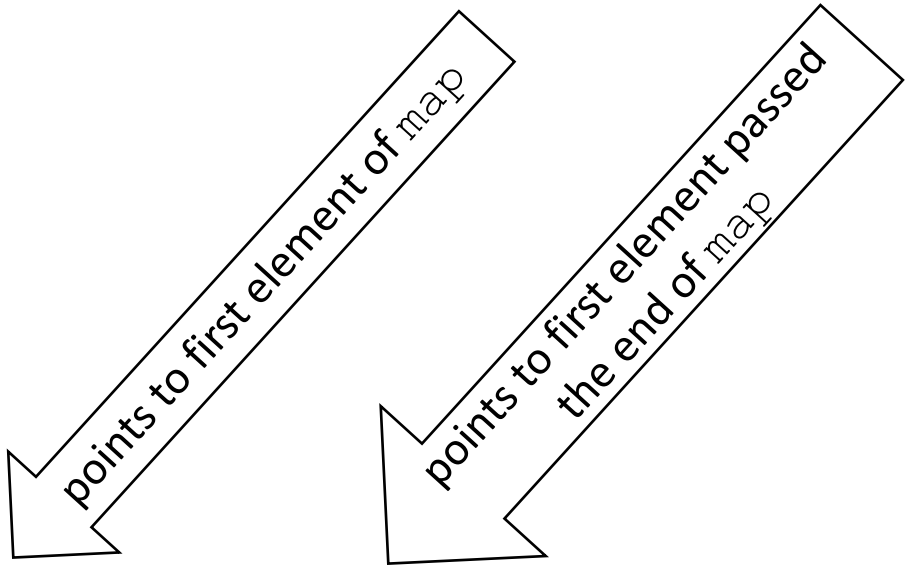
`map` iterators can be

dereferenced using `*`

accessed using `->`

compared using `==`

tested for inequality using `!=`



`map` iterators have member functions `begin()` and `end()`

```
cout << "Key\tValue" << endl;

map<string, int>::iterator it = mapOfWords.begin();

while(it != mapOfWords.end())
{
  cout << it->first << "\t"
       << it->second << endl;

   it++;
}
```

iterator

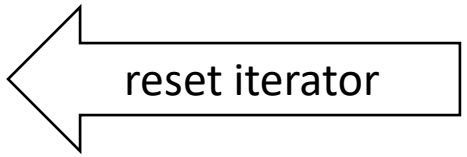
start a beginning of `map`

end when iterator goes one past the end of the `map`

```
Key   Value
earth 1
moon  4
sun   3
```

map3Demo.cpp

```
cout << "Enter a value to search for in our map ";
cin >> SearchValue;

it = mapOfWords.begin();
```




```
while(it != mapOfWords.end())
{
   if (it->second == SearchValue)
   {
      cout << "Key " << it->first << " has value " << it->second << endl;
   }
   it++;
}
```

```
Key   Value
earth 1
moon  4
sun   3

Enter a value to search for in our map 4
Key moon has value 4
```

map3Demo.cpp

```
void PrintVector(vector<int> PV)
{
	cout << "\nMyVector contains\n" << endl;

	for (auto it : PV)
		cout << it << ' ';
	cout << endl;
}
```

```
void PrintVector(vector<int> PV)
{
	cout << "\nMyVector contains\n" << endl;

	for (vector<int>::iterator it = PV.begin(); it != PV.end(); ++it)
	{
		cout << *it << ' ' ;
	}
	cout << endl;
}
```

```
cout << "Key\tValue" << endl;

map<string, int>::iterator it = mapOfWords.begin();

while(it != mapOfWords.end())
{
   if (it->second != false)
   {
      cout << it->first << "\t"
           << it->second << endl;
   }
   it++;
}
```

iterator

start a beginning of `map`

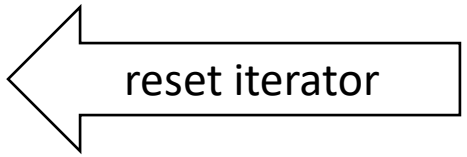
end when iterator goes one past the end of the `map`

eliminates "added" `venus`

```
Key   Value
earth 1
moon  4
sun   3
```

map3Demo.cpp

```
cout << "Enter a value to search for in our map ";
cin >> SearchValue;

it = mapOfWords.begin();
```




```
while(it != mapOfWords.end())
{
   if (it->second == SearchValue)
   {
      cout << "Key " << it->first << " has value " << it->second << endl;
   }
   it++;
}
```

```
Key   Value
earth 1
moon  4
sun   3

Enter a value to search for in our map 4
Key moon has value 4
```

map3Demo.cpp