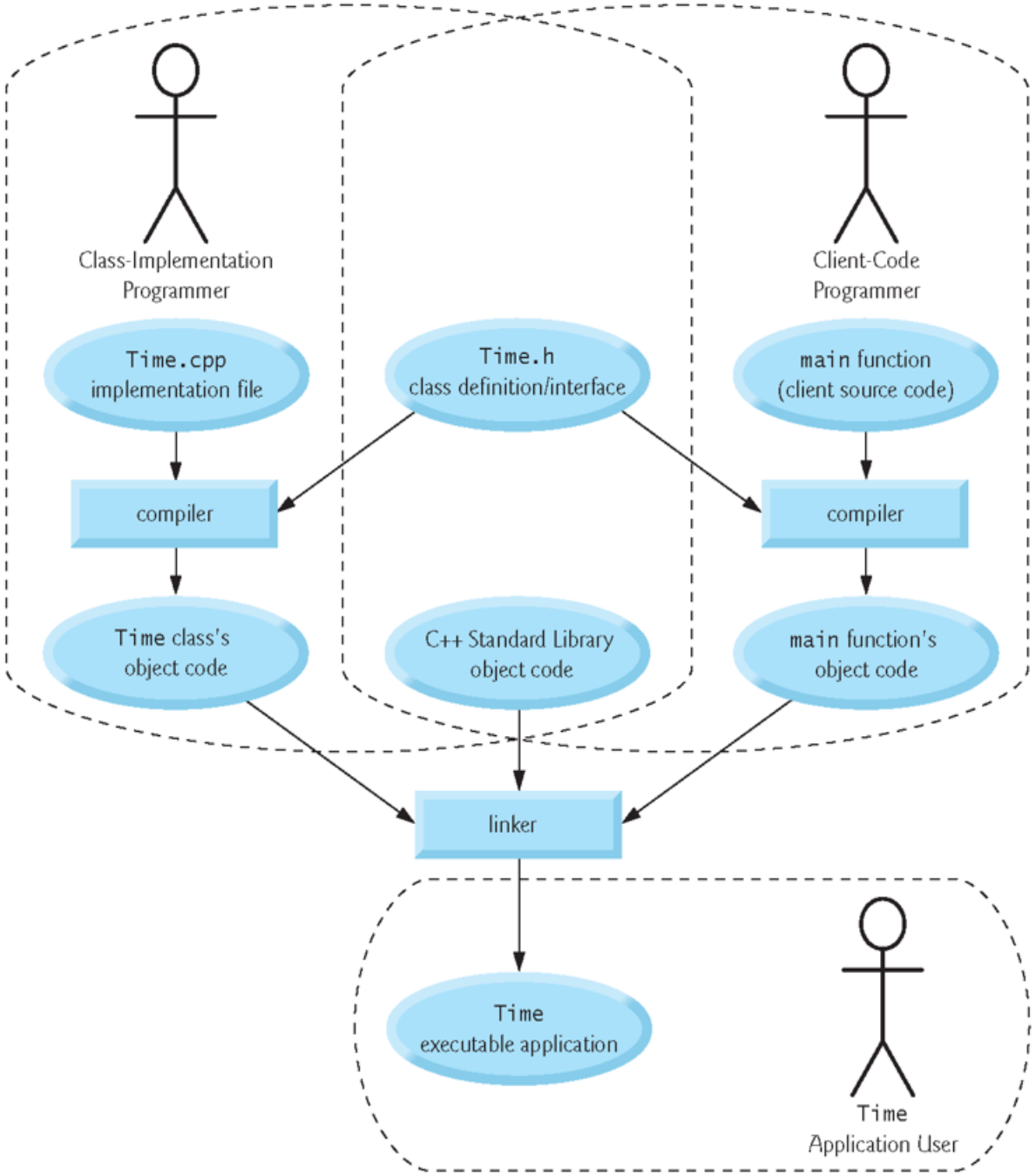
11

CSE 1325

Week of 11/02/2020

Instructor : Donna French




```
SRC1 = TimeTest.cpp
SRC2 = Time.cpp
OBJ1 = $(SRC1:.cpp=.o)
OBJ2 = $(SRC2:.cpp=.o)
EXE = $(SRC1:.cpp=.e)



CFLAGS = -g -std=c++11

all : $(EXE)

$(EXE): $(OBJ1) $(OBJ2)
        g++ $(CFLAGS) $(OBJ1) $(OBJ2) -o $(EXE)

$(OBJ1) : $(SRC1)
        g++ -c $(CFLAGS) $(SRC1) -o $(OBJ1)

$(OBJ2) : $(SRC2)
        g++ -c $(CFLAGS) $(SRC2) -o $(OBJ2)
```

```
// operator overload Demo

#include "Widget.h"

#include <iostream>

using namespace std;

int main(void)
{
   Widget W1{"Wiggy1", "red"};
   Widget W2{"Wiggy2", "blue"};

   cout << "Widgets are equivalent if they are the same color" << endl;

   if (W1 == W2)
       cout << "equivalent" << endl;
   else
       cout << "not equivalent" << endl;

   return 0;
}
```

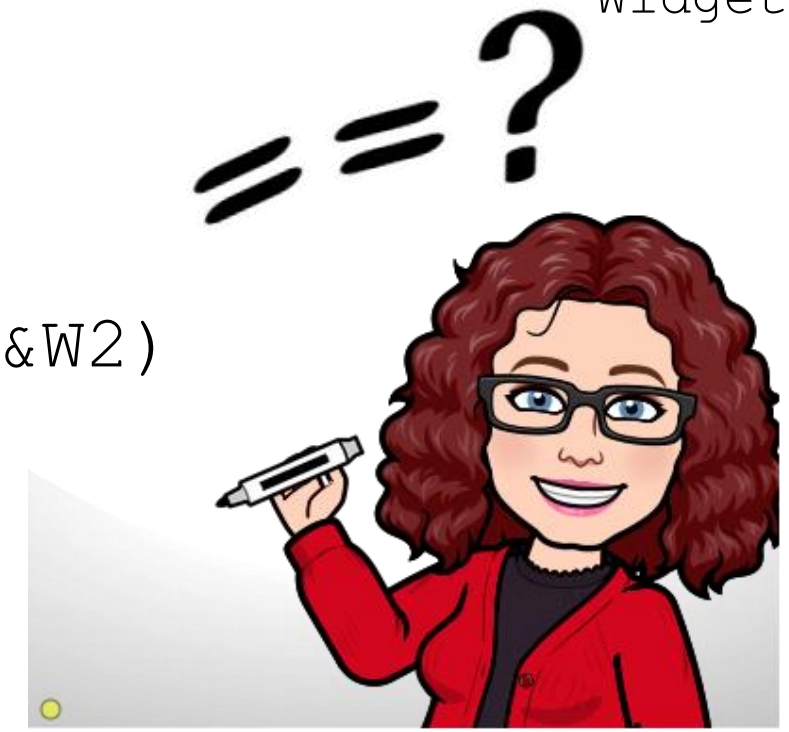
Widget.h

```
// Widget header file

#include <iostream>

class Widget
{
   public :
      Widget(std::string Name, std::string Coloring) : name{Name}, color{Coloring}
      {
      }

      std::string getName()
      {
         return name;
      }

      std::string getColor()
      {
         return color;
      }

   private :
      std::string name;
      std::string color;

};
```

```
Widget W1{"Wiggy1", "red"};
Widget W2{"Wiggy2", "blue"};
```

Widget.h

What would the operator overload function look like?




```
bool operator==(Widget &W1, Widget &W2)
{
    if (W1.color == W2.color)
        return true;
    else
        return false;
}
```

Widget.h

```
// Widget header file

#include <iostream>

class Widget
{
   friend bool operator==(Widget &, Widget &);

   public :
      Widget(std::string Name, std::string Coloring) : name{Name}, color{Coloring}
      {
      }

      std::string getName()
      {
         return name;
      }

      std::string getColor()
      {
         return color;
      }

   private :
      std::string name;
      std::string color;

};
```

```
bool operator==(Widget &W1, Widget &W2)
{
   if (W1.color == W2.color)
      return true;
}
```

```
// operator overload Demo

#include "Widget.h"

#include <iostream>

using namespace std;

int main(void)
{
   Widget W1{"Wiggy1", "red"};
   Widget W2{"Wiggy2", "blue"};

   cout << "Widgets are equivalent if they are the same color" << endl;

   cout << W1 << W2;

   if (W1 == W2)
       cout << "\n\nequivalent" << endl;
   else
       cout << "\n\nnot equivalent" << endl;

   return 0;
}
```

```
Widgets are equivalent if they are the same color

Widget name  : Wiggy1
Widget color : red
Widget name  : Wiggy2
Widget color : blue

not equivalent
```

Widget.h

What would the << overload function look like?


```
std::ostream& operator<<(std::ostream& output, const Widget& Wout)
{
   output << "\nWidget name  : " << Wout.name
          << "\nWidget color : " << Wout.color;

   return output;
}
```

Widget.h

```
// Widget header file

#include <iostream>

class Widget
{
   friend bool operator==(Widget &, Widget &);
   friend std::ostream& operator<<(std::ostream&, const Widget&);

   public :
      Widget(std::string Name, std::string Coloring) : name{Name}, color{Coloring}
      {
      }

      std::string getName()
      {
         return name;
      }

      std::string getColor()
      {
         return color;
      }

   private :
      std::string name;
      std::string color;
```

```
std::ostream& operator<<(std::ostream& output, const Widget& Wout)
{
   output << "\nWidget name  : " << Wout.name
          << "\nWidget color : " << Wout.color;

   return output;
}
```

Widget.cpp

```
#include <iostream>
#include "Widget.h"

std::string Widget::getName()
{
   return name;
}

std::string Widget::getColor()
{
   return color;
}

bool operator==(Widget &W1, Widget &W2)
{
   if (W1.color == W2.color)
      return true;
}

std::ostream& operator<<(std::ostream& output, const Widget& Wout)
{
   output << "\nWidget name  : " << Wout.name
          << "\nWidget color : " << Wout.color;

   return output;
}
```

Constructor with Default Arguments

Like other functions, constructors can specify default arguments.

Time.h

```
Time(int = 0, int = 0, int = 0);
```

Time.cpp

```
Time::Time(int hour, int minute, int second)
{
    setTime(hour, minute, second);
}
```

```
class Time
{
   public:
      Time(int = 0, int = 0, int = 0); // default constructor

      // set functions
      void setTime(int, int, int); // set hour, minute, second
      void setHour(int); // set hour (after validation)
      void setMinute(int); // set minute (after validation)
      void setSecond(int); // set second (after validation)

      // get functions
      unsigned int getHour() const; // return hour
      unsigned int getMinute() const; // return minute
      unsigned int getSecond() const; // return second

      std::string toUniversalString() const; // 24-hour time format string
      std::string toStandardString() const; // 12-hour time format string
   private:
      unsigned int hour{0}; // 0 - 23 (24-hour clock format)
      unsigned int minute{0}; // 0 - 59
      unsigned int second{0}; // 0 - 59
};
```

```
void Time::setTime(int h, int m, int s)
{
   setHour(h); // set private field hour
   setMinute(m); // set private field minute
   setSecond(s); // set private field second
}
```

```
// set hour value
void Time::setHour(int h)
{
   if (h >= 0 && h < 24)
   {
      hour = h;
   }
   else
   {
      throw invalid_argument("hour must be 0-23");
   }
}
```

```
// set minute value
void Time::setMinute(int m)
{
    if (m >= 0 && m < 60)
    {
        minute = m;
    }
    else
    {
        throw invalid_argument("minute must be 0-59");
    }
}
```

```
// set second value
void Time::setSecond(int s)
{
   if (s >= 0 && s < 60)
   {
      second = s;
   }
   else
   {
      throw invalid_argument("second must be 0-59");
   }
}
```

Exception Handling

C++ uses a throw statement to signal that an exception or error case has occurred.

To use a throw statement, simply use the `throw` keyword, followed by a value of any data type you wish to use to signal that an error has occurred.

Typically, this value will be an error code, a description of the problem, or a custom exception class.

```
throw -1;
   terminate called after throwing an instance of 'int'
   Aborted (core dumped)
```

```
throw "Catch it!!!";
   terminate called after throwing an instance of 'char const*'
   Aborted (core dumped)
```

Exception Handling

A `throw` statement acts as a signal that some kind of problem that needs to be handled has occurred.

Throwing exceptions is only one part of the exception handling process.

In C++, we use the **try** keyword to define a block of statements (called a **try block**). The try block acts as an observer looking for any exceptions that are thrown by any of the statements within the try block.

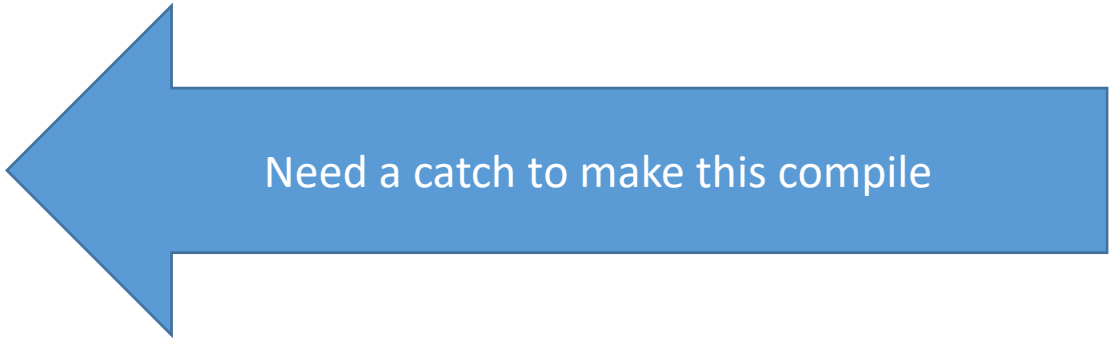
```
try
{
    throw "Catch it!!!";
}
```

Exception Handling

```
throw "Catch it!!!";
```

```
terminate called after throwing an instance of 'char const*'
Aborted (core dumped)
```

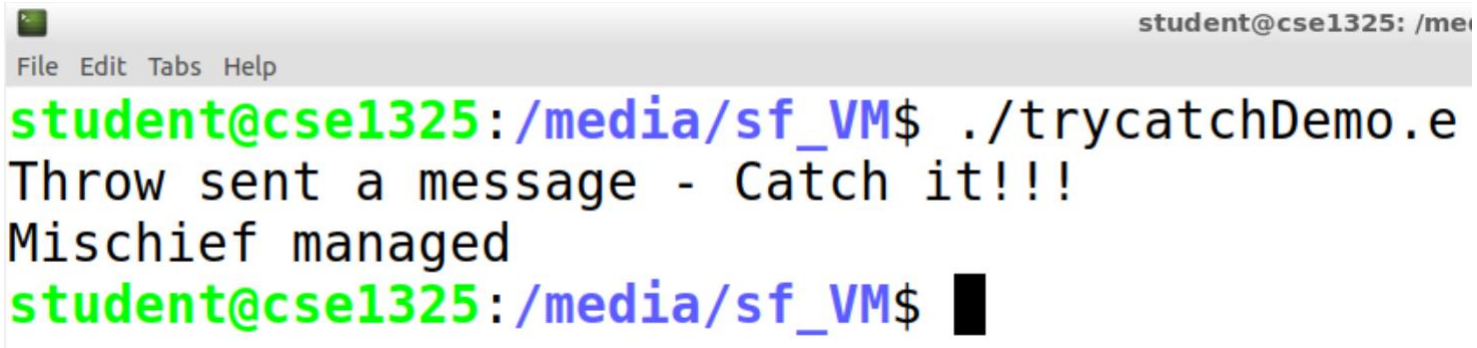
```
try
{
    throw "Catch it!!!";
}
```



Exception Handling

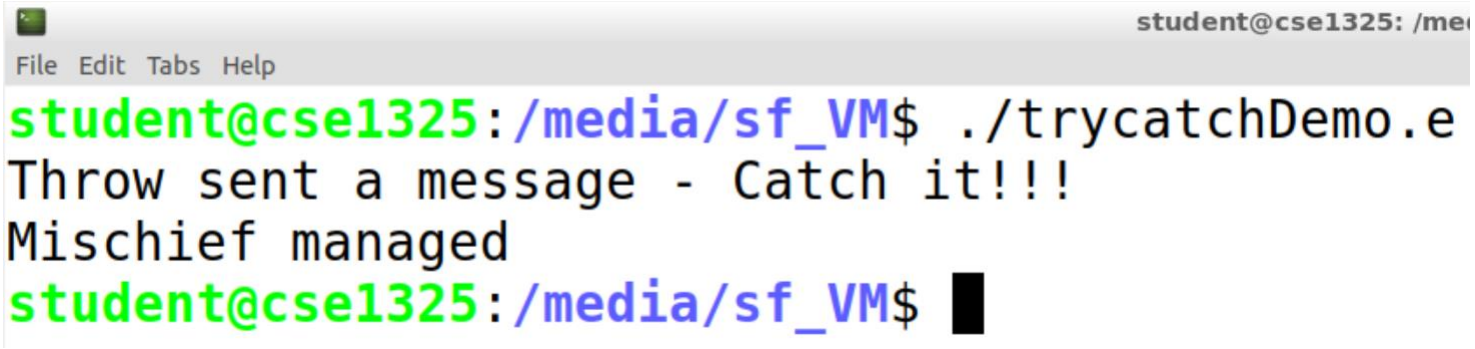
```
try
{
    throw "Catch it!!!";
}
catch(const char *message)
{
    cout << "Throw sent a message - " << message << endl;
}

cout << "Mischief managed" << endl;
```

Exception Handling



The program continues to run after the exception is handling and the final message "Mischief managed" is able to print.

Exception Handling




```
try
{
    throw "Catch it!!!";
}
catch(const char *message)
{
    cout << "Throw sent a message - " << message << endl;
}

cout << "Mischief managed" << endl;
```

```
try
{
	cout << "Before throw..." << endl;
	throw "Catch it!!!";
	cout << "After throw..." << endl;
}

catch(const char *message)
{
	cout << "Throw sent a message - " << message << endl;
}

cout << "Mischief managed" << endl;
```


```
student@cse1325:/media/sf_VM$ ./trycatchDemo.e
Before throw...
Throw sent a message - Catch it!!!
Mischief managed
student@cse1325:/media/sf_VM$
```

```
try
{
        int x = 100;
        throw "Catch it
}

catch(const char *message)
{
        cout << "Throw sent a message - " << message << endl;
}

cout << "Mischief managed " << x << " times" << endl;
```

Any variables declared in a `try` block are out of scope and not accessible outside of it.

```
g++ -c -g -std=c++11 trycatchDemo.cpp -o trycatchDemo.o
trycatchDemo.cpp: In function ‘int main()’:
trycatchDemo.cpp:24:33: error: ‘x’ was not declared in this scope
  cout << "Mischief managed " << x << " times" << endl;
                                 ^
makefile:14: recipe for target 'trycatchDemo.o' failed
make: *** [trycatchDemo.o] Error 1
```

Constructor with Default Arguments

A constructor that defaults all of its arguments is also a default constructor – a constructor that can be invoked with no arguments.

There can be at most one default constructor per class.

```
Time t1;
Time t2{2};
Time t3{21,34};
Time t4{12,25,42};
```

Default Memberwise Assignment

The assignment operator (=) can be used to assign an object to another object of the same class.

By default, the assignment is performed by memberwise assignment which is also called **copy assignment**.

Each data member of the object on the right of the = is assigned individually to the same data member in the object on the left of the =.

```
Time MyTime;                  //Instantiate objects MyTime and YourTime
Time YourTime;

MyTime.setTime(13, 27, 6); // change time

YourTime = MyTime;            // Set YourTime equal to MyTime


38              YourTime = MyTime;
(gdb) p MyTime
$1 = {hour = 13, minute = 27, second = 6}

(gdb) p YourTime
$2 = {hour = 0, minute = 0, second = 0}

(gdb) step

(gdb) p YourTime

$3 = {hour = 13, minute = 27, second = 6}
```

Check value of `MyTime` before executing assignment operator

Check value of `YourTime` before executing assignment operator

Execute the assignment operator

`YourTime` now has the same values as `MyTime`

Copy Constructor

Objects may be passed as function arguments and may be returned from functions.

The default is to pass by value.

Customized copy constructors are needed for classes whose data members contain pointers to dynamically allocated memory

```
48          displayTimeCopy("Displaying YourTime", YourTime);
(gdb) p &YourTime
$11 = (Time *) 0x7fffffffe0b0
(gdb) step
```

Address of `YourTime` in `main()`

```
displayTimeCopy (message="Displaying YourTime", time=...) at TimeTest.cpp:18
18      {
(gdb) p &time
$12 = (Time *) 0x7fffffffe008
```

Address of `time` in `displayTimeCopy()`

```
void displayTimeCopy(string message, Time time)
```

Object `time` will be received by `displayTimeCopy()` as a copy (pass by value)

Address of `YourTime` in `main()` is not the same as the address of `time` in `displayTimeCopy()`

The copy constructor made a copy of `YourTime`

```
47          displayTime("Displaying YourTime", YourTime);
(gdb) p &YourTime
$9 = (Time *) 0x7fffffffe0b0
```

Address of `YourTime` in `main()`

```
displayTime (message="Displaying YourTime", time=...) at TimeTest.cpp:11
11    {
(gdb) p &time
$10 = (const Time *) 0x7fffffffe0b0
```

Address of `time` in `displayTime()`

```
void displayTime(const string& message, const Time& time)
```

Object `time` will be received by `displayTimeCopy()` as an address (pass by reference)

Address of `YourTime` in `main()` is the same as the address of `time` in `displayTime()`

Copy Constructor

When an object is passed by value, C++ creates a new object and uses a **copy constructor** to copy the original object's values into the new object.

Uses memberwise assignment/copy assignment.

If you do not provide a copy constructor for your classes, C++ will create a public copy constructor for you.

It is possible to create a customized copy constructor.

Copy Constructor

```
Time MyTime;
MyTime.setTime(13, 27, 6); // change time
Time YourTime(MyTime);
```

```
27          Time YourTime(MyTime);
```


```
(gdb) p YourTime
$2 = {hour = 13, minute = 27, second = 6}

(gdb) p MyTime
$3 = {hour = 13, minute = 27, second = 6}
```

Debug Note : debug does not step into the default copy constructor

Time.cpp


```
Time::Time(unsigned int h, unsigned int m, unsigned int s)
      : hour{h}, minute{m}, second{s}
{
}
```


```
Time::Time(const Time &timeCopy)
    : hour{timeCopy.hour}, minute{timeCopy.minute}, second{timeCopy.second}
{
}
```

Time.h

```
Time(unsigned int=0, unsigned int=0, unsigned int=0);
Time(const Time &);
```

```
27       Time YourTime(MyTime);
(gdb) step
Time::Time (this=0x7fffffffdff8, timeCopy=...) at Time.cpp:17
17     : hour{timeCopy.hour}, minute{timeCopy.minute},second{timeCopy.second}
```

```
Time::Time(const Time &timeCopy)
```


```
Time::Time(const Time timeCopy)
    : hour{timeCopy.hour}, minute{timeCopy.minute}, second{timeCopy.second}
{
}
```

Time.h:13:19: error: invalid constructor; you probably meant ‘Time (const Time&)’


```
Time::Time(Time &timeCopy)
    : hour{timeCopy.hour}, minute{timeCopy.minute}, second{timeCopy.second}
{
}
```

/media/sf_VM/TimeTest.cpp:27: undefined reference to `Time::Time(Time const&)'

```
Time YourTime(MyTime);
```

Debug Note : Debug will show you the class's definition when you use `ptype`

```
25          Time MyTime;
(gdb) ptype Time
type = class Time {
  private:
    unsigned int hour;
    unsigned int minute;
    unsigned int second;

  public:
    Time(unsigned int, unsigned int, unsigned int);
    void setTime(int, int, int);
    std::__cxx11::string toUniversalString(void) const;
    std::__cxx11::string toStandardString(void) const;
}
```

Debug Note : Debug will step into the constructor which allows you to see the default parameter values, the "this" pointer and the member initializer list.

```
25          Time MyTime;

Time::Time (this=0x7fffffffdff8, h=0, m=0,
s=0) at Time.cpp:12
12          : hour{h}, minute{m}, second{s} {};
```

Destructors

A destructor is another type of special member function.

The name of the destructor for a class is the tilde character (~) followed by the class name.

```
~Time();
```

The tilde operator is the bitwise complement operator so it is logical that the destructor is the complement of the constructor.

A destructor may not specify parameters or a return type.

Destructors

A class's destructor is called implicitly when an object is destroyed.

An object is destroyed when program execution leaves the scope in which that object was instantiated.

The destructor itself does not actually release the object's memory but it does perform termination housekeeping before the object's memory is reclaimed by the system in order to reused to hold new objects.

Destructors

Receives no parameters and returns no value

May not specify a return type—not even void

A class may have only one destructor

Destructor overloading is not allowed

It is a syntax error to attempt to pass arguments to a destructor, to specify a return type for a destructor (even `void` cannot be specified), to return values from a destructor or to overload a destructor.

If the programmer does not explicitly provide a destructor, the compiler creates an “empty” destructor

When Constructors and Destructors Are Called

Constructors and destructors are called implicitly by the compiler

- Order of these function calls depends on the order in which execution enters and leaves the scopes where the objects are instantiated

Destructor calls are made in the reverse order of the corresponding constructor calls

Storage classes of objects can alter the order in which destructors are called

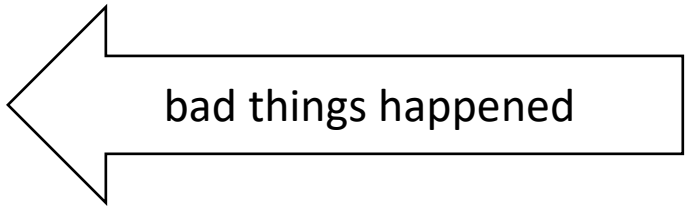
When Constructors and Destructors Are Called Global Scope

Constructors are called before any other function (including `main`)

The corresponding destructors are called when `main` terminates

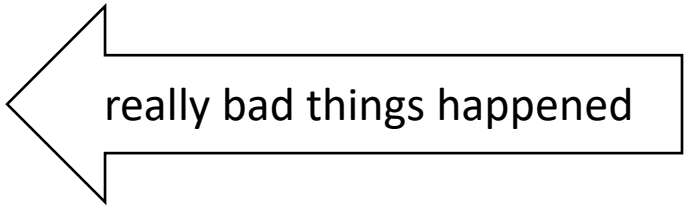
If the function `exit()` is used

- the program is forced to terminate immediately
- the destructors of local objects are not executed
- used to terminate a program when a fatal unrecoverable error is detected



If the function `abort()` is used

- performs similarly to function `exit`
- forces the program to terminate immediately without allowing programmer-defined cleanup code of any kind to be called
- usually used to indicate an abnormal termination of the program



When Constructors and Destructors Are Called Non-`static` Local Objects

Constructor is called when the object is defined.

Corresponding destructor is called when execution leaves the object's scope.

Constructors and destructors are called each time execution enters and leaves the scope of the object.

Destructors are not called if the program terminates with an `exit()` or `abort()`.

When Constructors and Destructors Are Called `static` Local Objects

Constructor is called only once when execution first reaches where the object is defined.

Corresponding destructor is called when `main` terminates or when `exit()` is called.

Destructors are not called if the program terminates with an `abort()`.

When Constructors and Destructors Are Called

Global	non-static	static
constructors are called before any other function (including `main`)	constructors are called each time execution enters and leaves the scope of the object	constructor is called only once when execution first reaches where the object is defined
destructors are called when `main` terminates – not called with `exit()` or `abort()`	destructors are called when `main` terminates or when execution leaves the object's scope – not called with `exit()` or `abort()`	destructor is called when `main` terminates or when `exit()` is called – not called with `abort()`
destroyed in the reverse order of their creation	destroyed in the reverse order of the constructor calls	destroyed in the reverse order of their creation

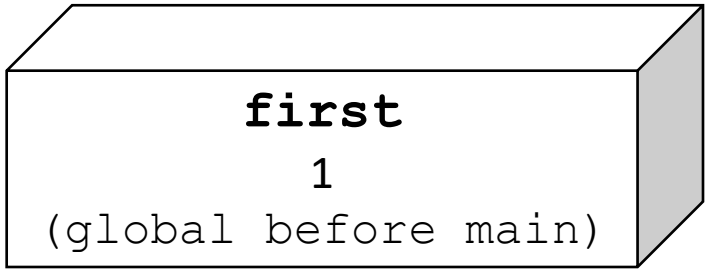
```
#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy
{
  public:
    CreateAndDestroy(int, std::string); // constructor
    ~CreateAndDestroy();                // destructor

  private:
    int objectID;                       // ID number for object
    std::string message;                // message describing object
};

#endif
```

CreateAndDestroy.h

```cpp
#include <iostream>
#include "CreateAndDestroy.h"

// constructor sets object's ID number and descriptive message
CreateAndDestroy::CreateAndDestroy(int ID, std::string messageString)
   : objectID{ID}, message{messageString}
   {
      std::cout << "\tObject " << objectID << "   constructor runs   "
                << message << std::endl;
   }

// destructor
CreateAndDestroy::~CreateAndDestroy()
{
   std::cout << "\tObject " << objectID << "   destructor runs    "
             << message << std::endl;
}
```

CreateAndDestroy.cpp

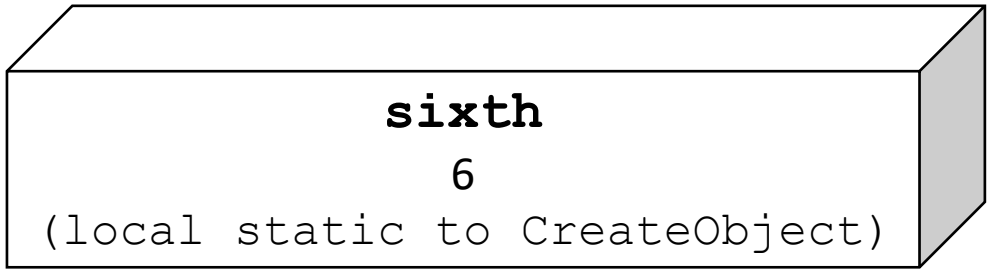
```
#include <iostream>
#include "CreateAndDestroy.h"

using namespace std;

void CreateObject(); // prototype

// Construct a global object
CreateAndDestroy first{1, "(global before main)"};
```

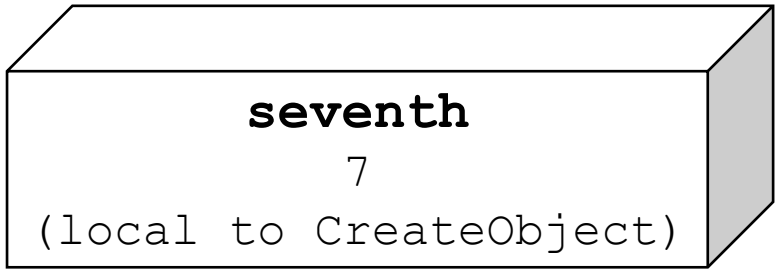



```
CreateAndDestroy::CreateAndDestroy(int ID, std::string messageString)
   : objectID{ID}, message{messageString}
   {
      std::cout << "\tObject " << objectID << "   constructor runs   "
                << message << std::endl;
   }
```

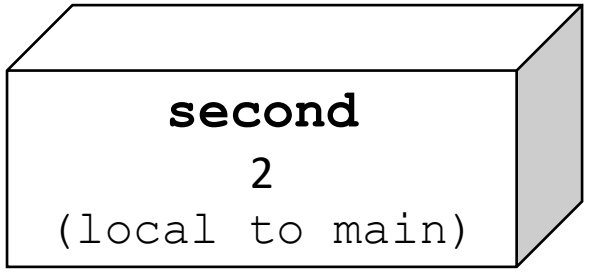
CreateAndDestroyTest.cpp

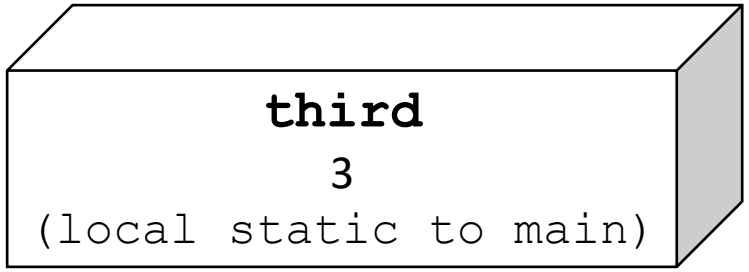
```
// function to create objects
void CreateObject()
{
   cout << "\nCreateObject() start\n" << endl;

   CreateAndDestroy fifth{5, "(local to CreateObject)"};

   static CreateAndDestroy sixth{6, "(local static to CreateObject)"};

   CreateAndDestroy seventh{7, "(local to CreateObject)"};

   cout << "\nCreateObject() finish\n" << endl;
}
```

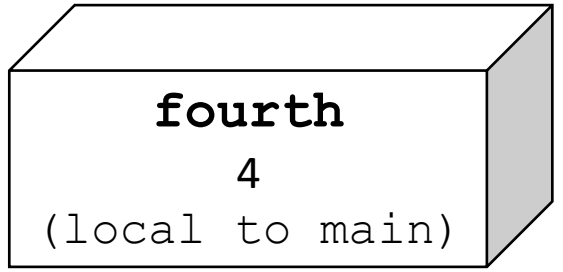


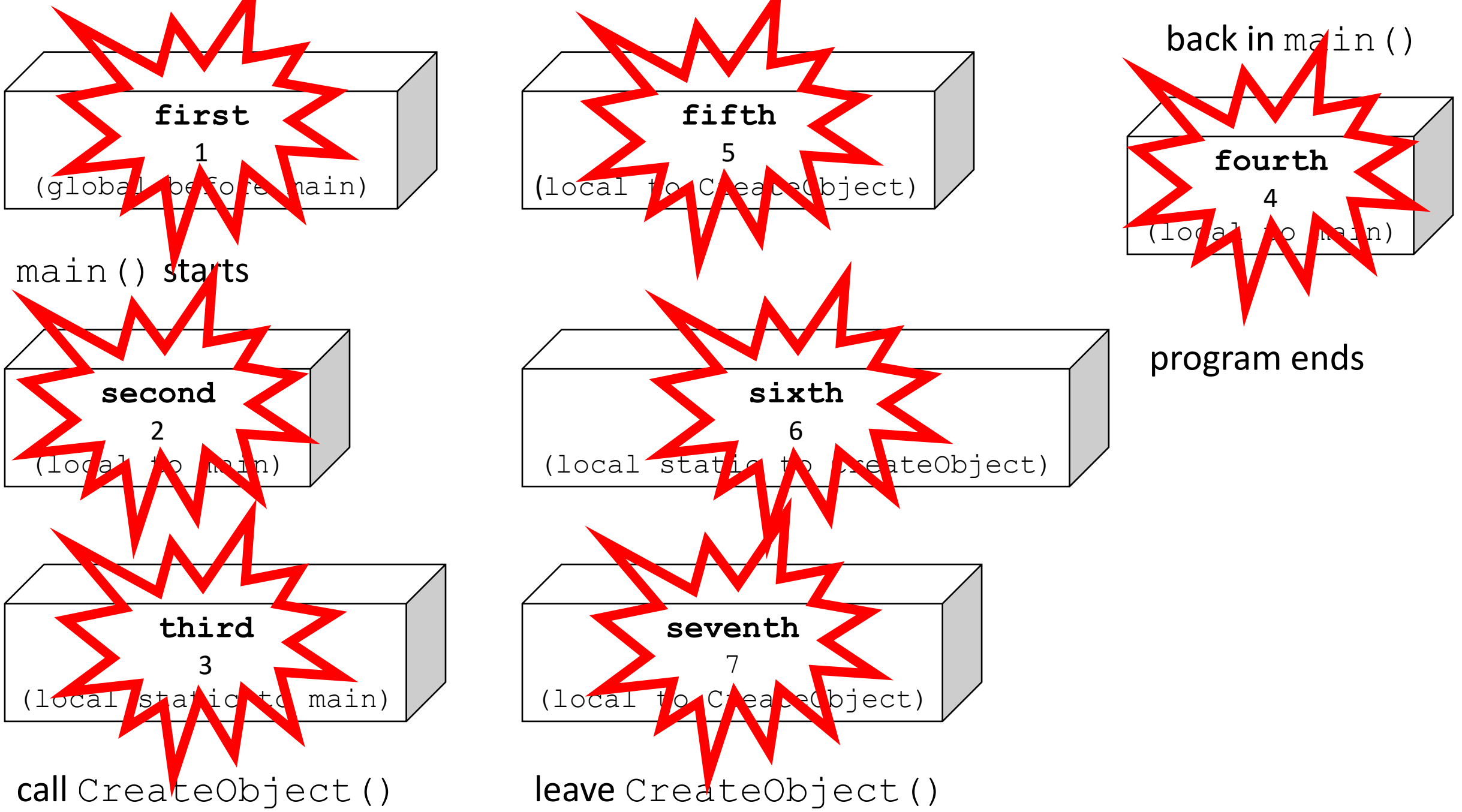


```
int main(void)
{
   cout << "\nmain() starts\n" << endl;

   CreateAndDestroy second{2, "(local to main)"};

   static CreateAndDestroy third{3, "(local static to main)"};

   cout << "\ncalling CreateObject()\n" << endl;
   CreateObject();
   cout << "\nback from CreateObject()\n" << endl;

   CreateAndDestroy fourth{4, "(local to main)"};

   cout << "\nmain() finish\n" << endl;
}
```









CreateAndDestroyTest.cpp

```
   Object 1   constructor runs   (global before main)

main() starts

   Object 2   constructor runs   (local to main)
   Object 3   constructor runs   (local static to main)

calling CreateObject()


CreateObject() start

   Object 5   constructor runs   (local to CreateObject)
   Object 6   constructor runs   (local static to CreateObject)
   Object 7   constructor runs   (local to CreateObject)

CreateObject() finish
```

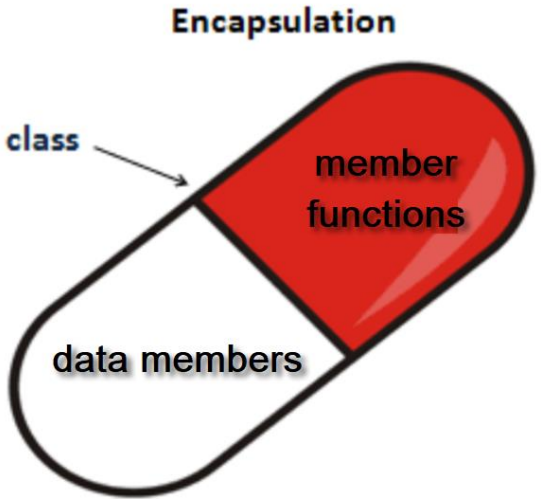
CreateAndDestroyTest.cpp

```
   Object 7   destructor runs    (local to CreateObject)
   Object 5   destructor runs    (local to CreateObject)

back from CreateObject()

Object 4   constructor runs   (local to main)

main() finish

   Object 4   destructor runs    (local to main)
   Object 2   destructor runs    (local to main)
   Object 6   destructor runs    (local static to CreateObject)
   Object 3   destructor runs    (local static to main)
   Object 1   destructor runs    (global before main)
```

CreateAndDestroyTest.cpp

```
Constructing Machine Bugs Bunny
Destroying Machine Bugs Bunny
Constructing Machine Cecil Turtle
Destroying Machine Bugs Bunny
Destroying Machine Cecil Turtle
Constructing Machine Daffy Duck
Destroying Machine Bugs Bunny
Destroying Machine Cecil Turtle
Destroying Machine Daffy Duck
Constructing Machine Elmer Fudd
Destroying Machine Elmer Fudd
Constructing Machine Fog Horn
Destroying Machine Bugs Bunny
Destroying Machine Cecil Turtle
Destroying Machine Daffy Duck
Destroying Machine Elmer Fudd
Destroying Machine Fog Horn
```

```
Pick a Snack Machine

0. Exit
1. Machine Bugs Bunny
2. Machine Cecil Turtle
3. Machine Daffy Duck
4. Machine Elmer Fudd
5. Machine Fog Horn
6. Add a new machine

Enter choice 0
Destroying Machine Bugs Bunny
Destroying Machine Cecil Turtle
Destroying Machine Daffy Duck
Destroying Machine Elmer Fudd
Destroying Machine Fog Horn
```



Encapsulation

Classes wrap attributes and member functions into objects created from those classes – an object's attributes and member functions are intimately related.

Objects may communicate with one another, but they are not normally allowed to know how other objects are implemented.

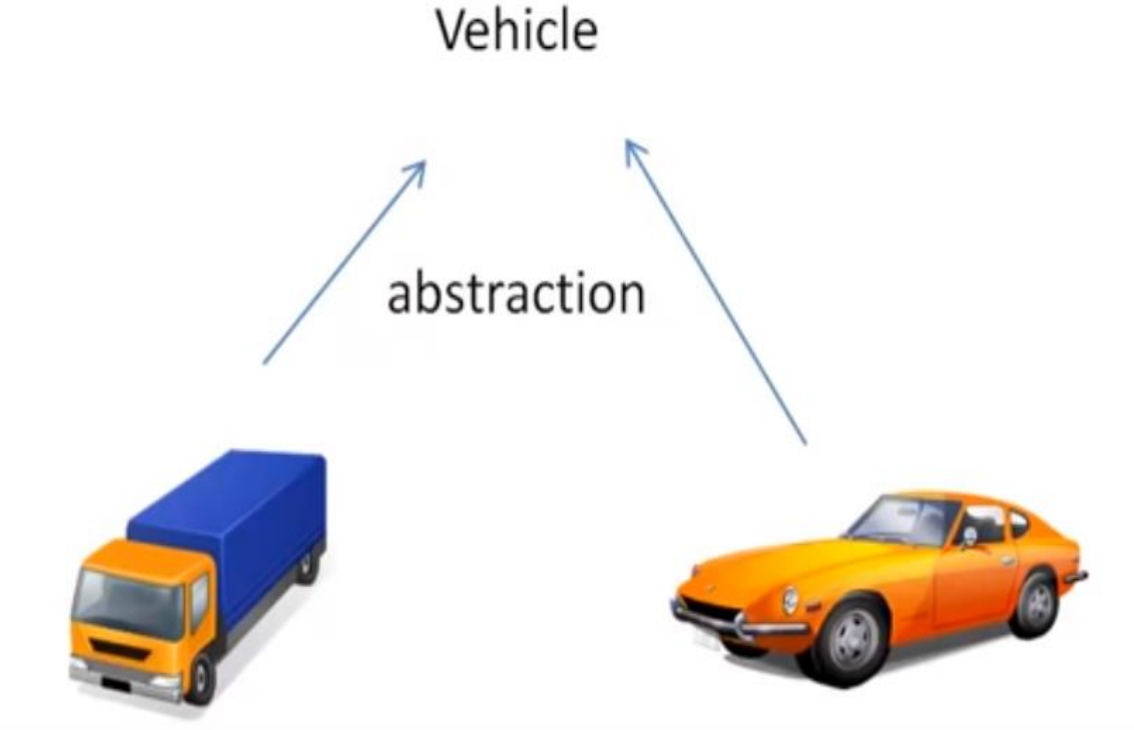
Encapsulation is the technique of information hiding - implementation details are hidden within the objects themselves.

Abstraction

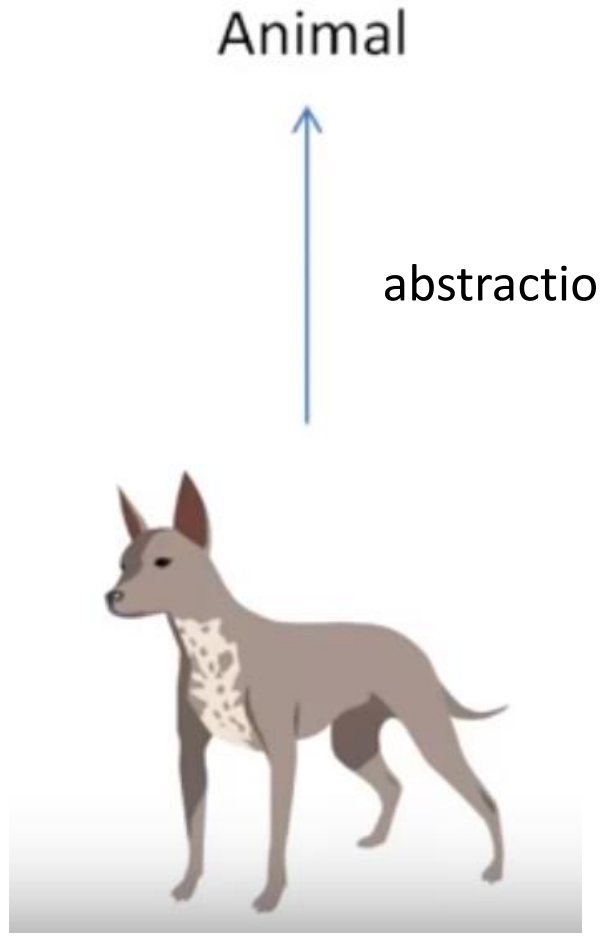
Abstraction is the concept of describing something in simpler terms, i.e abstracting away the details, in order to focus on what is important.

Abstraction is used to reduce complexity and allow efficient design and implementation of complex software systems.

Abstraction is the act of representing essential features without including the background details or explanations.

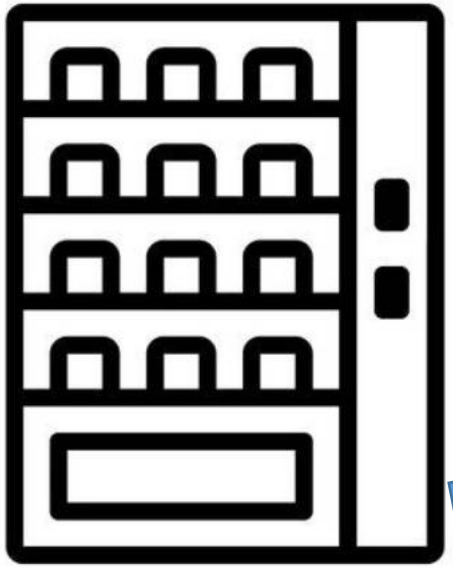


Vehicle is an abstraction of truck and car.



Animal is an abstraction of dog.

Vending Machine



Snack Machine

Abstraction

Abstraction

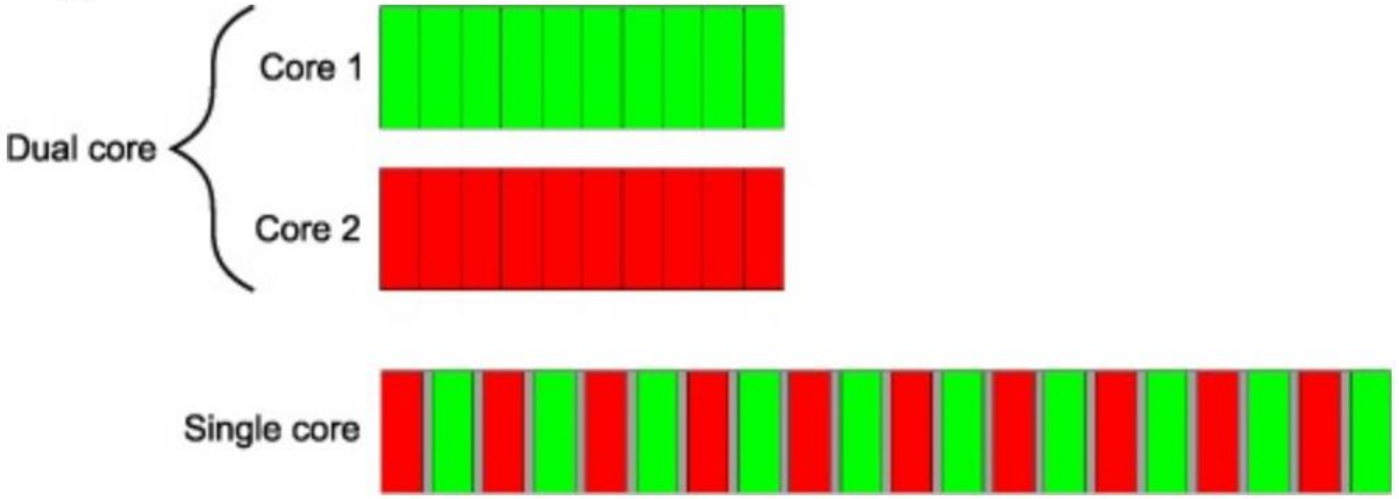
Coke Machine

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

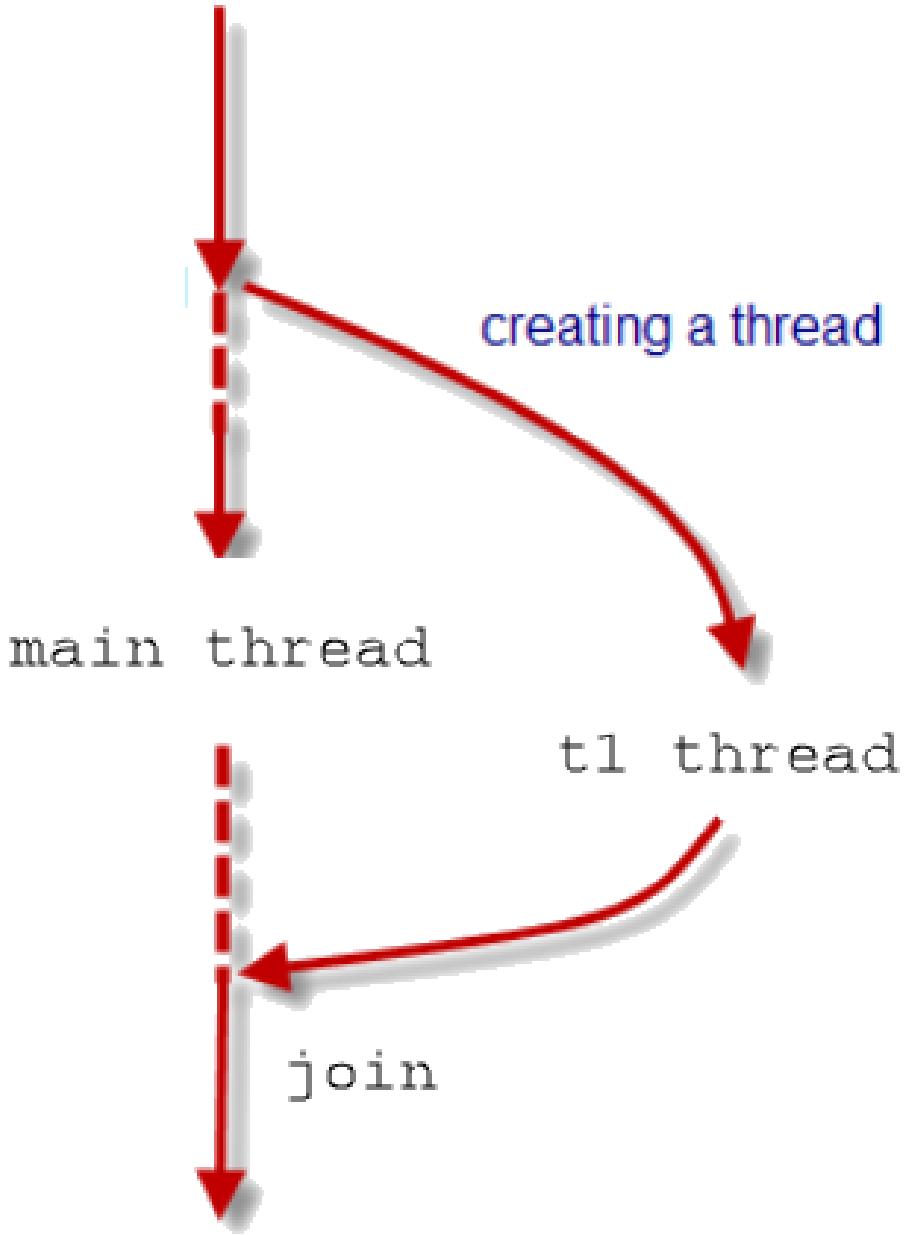
An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space.

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

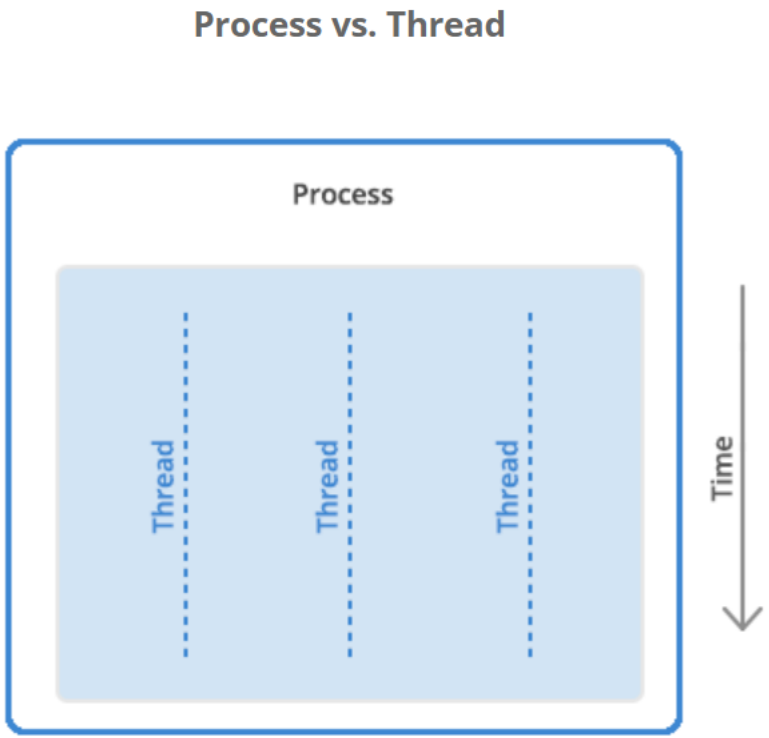
Text editor – one thread is accepting your typing, one thread is checking your spelling, one thread is occasionally saving your document. etc…

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.

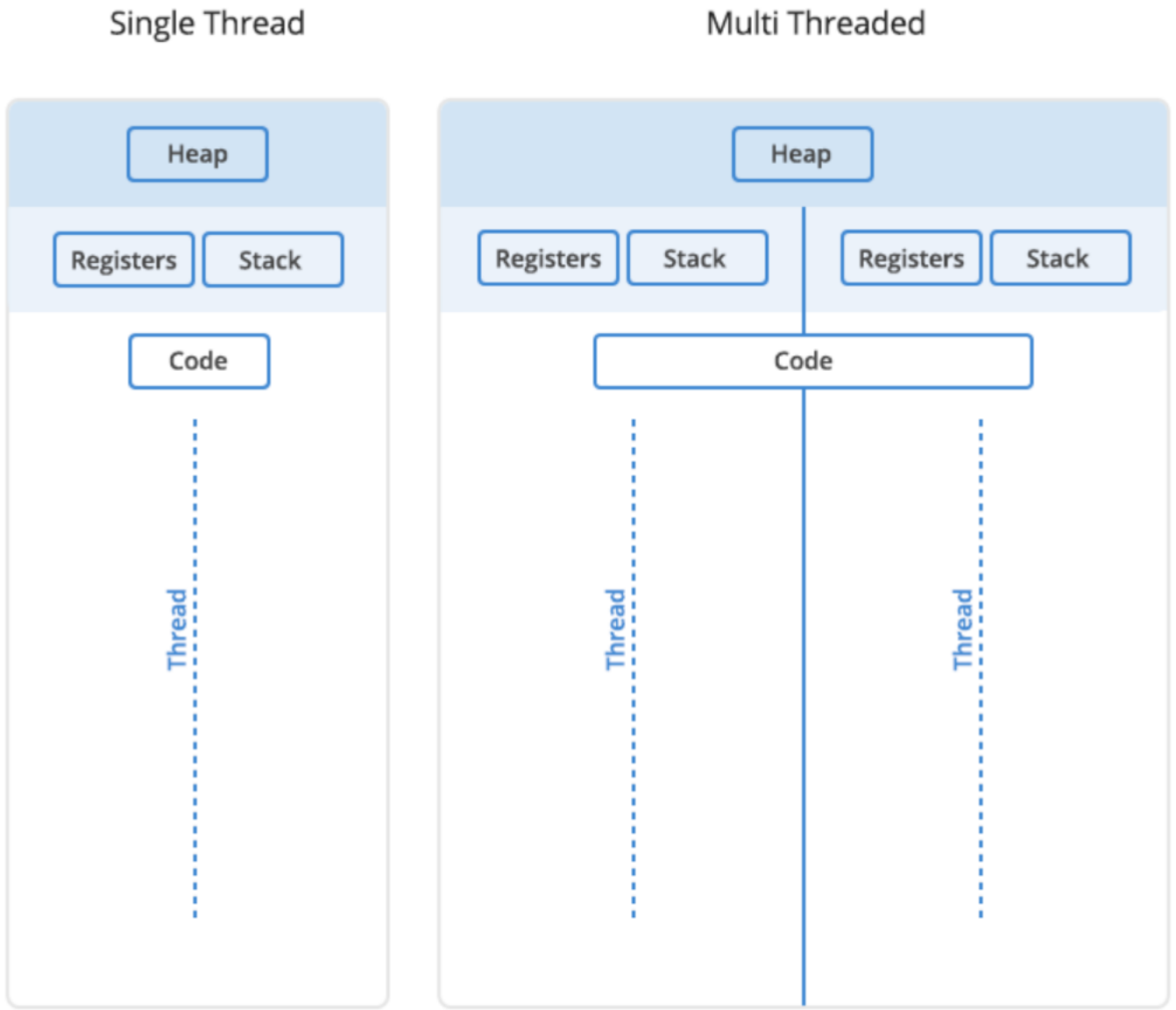


Threads

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
```