EMBEDDED SYSTEMS PROGRAMMING 2015-16

More About Languages

JAVA: ANNOTATIONS (1/2)

- Structured comments to source code (=**metadata**). They provide data about the code, but they are not part of the code itself
- Can be used
  - by the compiler to detect errors or suppress warnings
  - by software tools to generate documentation, code, ...
- Insert an annotation by prepending an "`@`"
- Not available in C and C++

JAVA: ANNOTATIONS (2/2)

Sample annotations used by the Java compiler

- **`@Deprecated`**: indicates that the annotated element should no longer be used
- **`@Override`**: informs the compiler that the element is meant to override an element declared in a superclass
- **`@SuppressWarnings`**: tells the compiler to suppress a set of specific warnings

COPY CONSTRUCTOR

- **Java, C++**

- The copy constructor is a special constructor used when a newly-instantiated object is a copy of an existing object

- First argument of the CC: <u>must</u> be a reference to an object of the same type as the one being constructed

COPY CONSTRUCTOR: C++

- A default CC is automatically generated by the compiler, but an user-provided CC is mandatory when the class
  - allocates memory dynamically,
  - owns non-shareable references, such as references to files

COPY CONSTRUCTOR: EXAMPLES (1/2)

- Java

```
class Pixel extends Point
{

...

    public Pixel(Pixel sourcepixel)   // Copy constructor
    {
        // Coordinates are copied by invoking the appropriate
        // constructor of the superclass
        super(sourcepixel.GetX(), sourcepixel.GetY());

        color = new byte[3];
        color[0] = sourcepixel.color[0];
        color[1] = sourcepixel.color[1];
        color[2] = sourcepixel.color[2];
    }

...

}
```

COPY CONSTRUCTOR: EXAMPLES (2/2)

- C++

```
Pixel::Pixel(Pixel& sourcepixel)
{
    // NOTE: the constructor of the base class cannot
    // be invoked: x and y must be copied explicitly
    SetX(sourcepixel.GetX());
    SetY(sourcepixel.GetY());

    color = new unsigned char [3];
    color[0] = sourcepixel.color[0];
    color[1] = sourcepixel.color[1];
    color[2] = sourcepixel.color[2];
}
```

JAVA: COPY CONSTRUCTOR VS. CLONING

- **Java**: an object can be copied by implementing either the copy constructor or the `clone()` method

```
Pixel px = new Pixel();

...

// If Pixel implements a copy constructor
Pixel copy1 = new Pixel(px);

// If Pixel implements the clone() method
Pixel copy2 = px.clone();
```

- However, **cloning is less flexible.**
  Example: `clone()` can't initialize blank `final` variables, while a constructor can

NESTED CLASS

- Java, C++
- A **nested class** is a class declared within the body of another class or interface; no special syntax
- A nested class is a member of its enclosing class
- A nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class

JAVA: INNER CLASS

- A non-static nested class is called an **inner class**

- Inner classes have access to members of the enclosing class, even if they are declared private; fields must be `final`

- Static nested classes are allowed access only through an object reference

- Inner classes cannot define static members

JAVA: EXAMPLES

- Instantiation of a static nested class

```
OuterClass.StaticNestedClass nestedObject =
 new OuterClass.StaticNestedClass();
```

- Instantiation of an inner class: instantiate the outer class first, then create the inner object within the outer object

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

JAVA: ANONYMOUS CLASS

- Inner class without a name
- Declaration coincide with instantiation, hence it must take place inside a method

```
// Instantiation of an anonymous class
// that implements the View.OnClickListener interface
bu.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Perform action on click
        tv.setText("Good job!");
    }
});
```

- An inner class declared inside a method (with or without a name) is called a “local class”

C++: NESTED CLASS

- A nested class can directly use names, type names, names of static members, and enumerators only from the enclosing class
- A nested class can be declared and then defined later
- The declaration/definition of a nested class do not cause any object to be instantiated: instantiation must be explicit
- `struct`s and `union`s can be nested as well

C++: EXAMPLE

```
class OuterClass
{
    class NestedClass1;      // forward declaration
    class NestedClass2;      // forward declaration
    class NestedClass1 {};   // definition of nested class

    /* ... */

    NestedClass1 n;          // instantiation of nested class
};

class OuterClass::NestedClass2 {}; // definition of nested class
```

C++: VIRTUAL FUNCTION

- Member function (=method) of a class, whose functionality can be overridden in its derived classes
- Declared with the `virtual` keyword
- Differently from plain overloading, calls are resolved at run time (more on this later)
- <u>Mandatory</u> when a base-class pointer is used to access an overridden method of the derived class

VIRTUAL FUNCTION: EXAMPLE

- Were `print()` not declared `virtual`, the method of the base class would be called in `main()`

```
#include <iostream>

class BaseClass
{
    public: virtual void print() {std::cout << "Base\n";}
};

class DerivedClass: public BaseClass
{
    // Override of the print() method
    public: void print() {std::cout << "Derived\n";}
};

int main(int argc, const char *argv[])
{
    // A derived-class object is assigned to a base-class pointer
    BaseClass * C = new DerivedClass();

    C->print();

    delete C; return 0;
}
```

ABSTRACT CLASS

- A class whose definition is incomplete.
  It cannot be instantiated: it can only be subclassed

- **Java**: abstract classes (and methods); interfaces

- **C++**: abstract classes; pure virtual methods

JAVA: INTERFACE

- Group of related methods with empty bodies (i.e., undefined methods)
- To be used, an interface must be **implemented** by a class

```
interface GeometricObject
{
    double Distance();   // Distance from the origin;
                         // implementation is not provided

    //...                   Further methods go here
}

public class Point implements GeometricObject
{
   //...                    Implementation goes here
}
```

JAVA: ABSTRACT CLASS/METHOD

- **Abstract method**: a method that is declared (without braces and followed by a semicolon, as in a C++ declaration) but not defined
- **Abstract class**: a class that is declared `abstract`.
  It may or may not include abstract methods.
  It cannot be instantiated, but it can be subclassed
- Unlike interfaces, abstract classes can contain
  - fields that are not `static` and `final`,
  - implemented methods

C++: ABSTRACT CLASS

- **Pure virtual function**: a method that is declared `virtual`, not defined, and followed by "`=0;`"
- **Abstract class**: a class that contains at least one pure virtual function

```
class GeometricObject   // Abstract class
{
public:
   virtual double Distance()=0;

   //...                     Further methods go here
}

public class Point:GeometricObject
{
   //...                     Implementation goes here
}
```

REFERENCES (1/3)

```
Point p = new Point();
```

- **Java**
- Objects (including some data types, such as arrays) are manipulated not directly, but by **reference**, i.e., via a "handle" to the object
- References are `null` when they do not reference any object
- The use of references is so pervasive that imprecise statements are often made, e.g., "Pass an object to the method" (wrong) instead of "Pass an object reference to the method" (correct)

REFERENCES: QUIZ 1

- Java

```java
Point p = new Point(0.0, 1.0);  // First reference: p
                                // p.y is now 1.0
Point q = p;                    // Second reference

...

q.y = 2.0;
```

- What is the value of `p.y` at the end of the code fragment? Is it `1.0` or `2.0`?

REFERENCES: QUIZ 2

- Java

```java
String letter = "o";
String s = "hello";          // These String objects
String t = "hell" + letter;  // contain the same text

if(s == t)
{
    ...
}
```

- Does `(s == t)` evaluate to `true` or `false`?

Example from "Java in a Nutshell"

JAVA VS. C++ (1/3)

- **Java:**

```
Point p;
```

`p` **is a reference to a** `Point` **object**

- **C++:**

```
Point p;
```

`p` **is an object of type** `Point`**, i.e., an instance of** `Point`

JAVA VS. C++ (2/3)

- Java:

```
Point p = new Point();
```

`p` is a reference to a `Point` object

- C++:

```
Point * p = new Point();
```

`p` is a <u>pointer</u> to a `Point` object, i.e.,
it contains the memory address of a `Point` object

JAVA VS. C++ (3/3)

- Java:

```java
public foo() {
     Point p = new Point();
     ...
}
```

  When the member `foo` ends: `p` is destroyed and the `Point` object is no longer referenced, so the garbage collector destroys it as well

- C++:

```cpp
void foo() {
     Point * p = new Point();
     ...
}
```

  When the member `foo` ends: `p` is destroyed, the `Point` object is no longer referenced but nobody destroys it **(memory leak)**

REFERENCES (2/3)

- **C++** (and C)
- A reference to an entity is an **alternate name** for that entity
- When you change a reference, you change the content of the referent

```
int i;
int & ri = i; // definition of the reference
ri++;         // same as writing i++
ri = 12;      // same as writing i=12
```

POINTERS VS. REFERENCES

- **C++** (and C)
- **Pointer**
  Distinct from the object it points to
  The "*****" operator is required to dereference an address
  The value of the pointer can be changed
  Can be `NULL`

```
int i;
int * pi = &i;
*pi = 12;
pi++;
```

- **Reference**
  Different name for the object it points to
  No operator required to dereference
  Once bound to an object, it cannot be changed
  Can't be `NULL`

```
int i;
int & ri = i;
ri = 12;
```

REFERENCES (3/3)

- **C++** (and C)
- Parameters are frequently passed by reference, not by value

```
void swap(int& i, int& j)
{
    int tmp = i;
    i = j;
    j = tmp;
}
```

& VS. & (NO KIDDING)

- C++ (and C)
- The symbol "**&**" is used
  - to define a reference

```
int & ri = i;
```

  - for the address-of operator

```
int * pi = &i;
```

REFERENCES IN C++

- References are further used while redefining operators

```
enum day
{
    Sun, Mon, Tue, Wed, Thu, Fri, Sat
};

// Redefine the ++ operator for day
day &operator++(day &d)
{
    d = (day)((d+1)%7);
    return d;
}

...

day n;

...

// Increment n to the next day
++n;
```

NAME BINDING

- The act of associating identifiers (of fields, of members, ...) with the correct class/object/function/...
- **Static binding** (aka early binding)
  “Binding as you know it”: the association is performed at compile time
- **Dynamic binding** (aka late binding)
  The association is performed at run time since at compile time there is not enough information to determine which object must be called

DYNAMIC BINDING: PROS

- It increases **flexibility**:
  some decisions are not hardwired in the source code, but they are taken only at run time

- It allows for **more extensible software**:
  new classes can be added at run time without recompiling, and without even knowing their source code

DYNAMIC BINDING: CONS

- It is **slower**:
  a search into a suitable data structure must be performed at run time to determine which object/method to use

BINDING: EXAMPLES (1/3)

- Both examples are in **Java**
- Example: static binding of an object

```
Point ImaginaryUnit = new Point(0.0, 1.0);
```

- Another example: is this static or dynamic binding?

```
Point ImaginaryUnit;
ImaginaryUnit = new Point(0.0, 1.0);
```

BINDING: EXAMPLES (2/3)

- Example: dynamic binding of objects in Java

```java
class ClassA
{
    public void print() {System.out.printf("A\n");}
}

class ClassA2 extends ClassA
{
    // Override of the print() method
    public void print() {System.out.printf("A2\n");}
}

class latebinding
{
    public static void main(String[] args)
    {
        ClassA C;        // Recall that C is just a reference

        for(int i=0; i<4; i++)
        {
            // A reference to the base class can be used with
            // derived objects, but not vice versa
            if(i%2==0) C=new ClassA(); else C=new ClassA2();

            C.print();   // Which print() should be called?
        }
    }
}
```

BINDING: EXAMPLES (3/3)

- Example: dynamic binding of objects in **C++**

```cpp
#include <iostream>

class ClassA
{
    // Recall that the "virtual" keyword is necessary to indicate
    // that the method may be overridden in derived classes.
    public: virtual void print() {std::cout << "A\n";}
};

class ClassA2: public ClassA
{
    // Override of the print() method
    public: void print() {std::cout << "A2\n";}
};

int main(int argc, const char *argv[])
{
    ClassA * C;

    for(int i=0; i<4; i++)
    {
        // A pointer to the base class can be used with
        // derived objects, but not vice versa
        if(i%2==0) C=new ClassA(); else C=new ClassA2();
        C->print();

        // This is not Java! Objects must be deleted
        delete C;
    }
   return 0;
}
```

RUN TIME METHOD INVOCATION

- Is it possible to invoke a method that is dynamically chosen at run time?

- Java: yes, use the **`Method` class**

- C++: yes, use **pointers to member functions**

JAVA: THE METHOD CLASS

- Part of the `java.lang.reflect` package (more on reflection later)

- Provides access to - and information about - a single method of a class or interface.
  Both class and instance methods can be accessed

- **`Object invoke(Object obj, Object... args)`**
  Invokes the method of `obj` represented by the instance of `Method`

- **`Method getMethod(String name, Class<?>... parameterTypes)`**
  Part of class `Class`. Returns a reference to a method

METHOD CLASS: EXAMPLE

- Invoking different object methods in different situations. The method is chosen at run time

```java
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;

...

// definition of a reference to a method
Method action;

// definition of an instance of graphicObject
graphicObject Hexagon = new graphicObject();

// Decide whether action should indicate method draw()
// or method repaint() of class graphicObject
if(condition == true) action = graphicObject.class.getMethod("draw");
else action = graphicObject.class.getMethod("repaint");;

...

// Invokes either draw or repaint according to the decision
// taken before
action.invoke(Hexagon);
```

POINTERS TO FUNCTIONS

- **C++**: as in C, it is possible to define a pointer to a function

```
// Declaration of function f
double f(int a, short b);

...

// Definition of a pointer to a function that (like f) receives
// an int and a short as parameters, and returns a double.
// The pointer is called pf, but - of course - any name is
// as good as pf
double (*pf)(int, short);

// Now pf points to f
pf = &f;

...

// Calling f directly
d = f(1,2);

// Calling f via the pointer
d = (*pf)(1,2);
```

POINTERS TO MEMBER FUNCTIONS

- **C++**: it is possible to define a pointer to a member function, i.e., a pointer to a method

```
// Definition of a pointer to a member function of class
// graphicObject that (like draw and repaint) receives
// nothing and returns nothing.
void (graphicObject::*action)();

// definition of an instance of graphicObject
graphicObject * Hexagon = new graphicObject();

...

// Decide whether action should indicate method draw()
// or method repaint() of class graphicObject
if(condition == true) action = &graphicObject::draw;
else action = &graphicObject::repaint;

...

// Invokes either draw or repaint according to the decision
// taken before
(Hexagon->*action)();
```

REFLECTION

- **Reflection**: the process by which a computer program can observe and modify its own structure and behavior at run time
- Data and code structures can be manipulated as well
- For OO languages: classes and objects can be observed and modified as well

TYPE INTROSPECTION

- **Type introspection**: the process by which an OO program can determine the type of an object at run time
- Supported by Java and C++
- Key functionalities: determining whether an object...
  - ...is an **instance of** a given class
  - ...**inherits from** the specified class

INTROSPECTION: JAVA (1/2)

- Introspection is natively supported in Java;
  some support is also provided by `java.lang.Object`
- **`getClass()`** method
  Inherited from `java.lang.Object`.
  Returns a **type token** [`Class<T>`], i.e., an instance of the
  class `Class` that represents the class of the calling object.
  Allows to check whether an object is an instance of a given class
- **`instanceof`** operator
  Returns `true` if the expression on its left can be cast
  to the type on its right.
  Allows to check whether an object is an instance of
  (or inherits from) a specified class

INTROSPECTION: JAVA (2/2)

- Example: invoking `instanceof` and `getClass()`

```
public void someMethod(Pixel pix)
{
    Point point;

    // Dynamically check whether pix is derived from Point
    if(pix instanceof Point)
    {
        // Dinamically cast to Point
        point = pix;

        // I can now manipulate the object as if it were a Point

        // However, I can always check whether the object is
        // indeed an instance of Pixel
        if(point.getClass().getName().equals("Pixel")) // Returns true!
        {
            System.out.printf("Pixel\n");
        }
    }
    else  // Check failed
    {
        ...
    }
}
```

INTROSPECTION: C++ (1/2)

- Introspection is natively supported in C++
- **`typeid(obj)`** operator
  Returns a reference to an object of type `type_info` that describes the type of object `obj`.
  Allows to check whether `obj` is an instance of a given class
- **`dynamic_cast<target-type>(pr)`** operator
  Succeeds if `pr` is a pointer (or reference) to either an object of type `target-type` or an object derived from it.
  If it succeeds, a valid pointer/reference is returned.
  Allows to check whether `pr` is derived from a given class

INTROSPECTION: C++ (2/2)

- Example: using `typeid` and `dynamic_cast`

```
void aClass::someMethod(Pixel * p_pix)
{
    Point * p_point;

    // Dynamically cast to Point*
    if(p_point = dynamic_cast<Point *>(p_pix))
    {
        // I can now manipulate the object as if it were a Point

        // However, I can always check whether the object is
        // indeed an instance of Pixel
        if(typeid(*p_point) == typeid(Pixel)) // Returns true!
        {
            cout << "Pixel\n";
        }
    }
    else  // Casting failed
    {
        ...
    }
}
```

PARAMETERIZED TYPES (1/2)

- **Define a class without knowing what datatype(s) will be handled by the operations of the class**
- The code must operate with any datatype(s) specified at instantiation time ("**generic programming**")
- Less source code duplication, same object code
- Example: a single, parametrized quicksort routine can sort data of any type (provided data can be compared)

PARAMETERIZED TYPES (2/2)

- Java: **generic types** (aka "generics")
- C++: **template classes**

WHY NOT OBJECT?

- A "very base" class (e.g., `Object` in Java) can be used instead, with the real object type inspected at runtime
- Coherency inside the class (all methods passing the same object type) manually handled
- No error detection at compile time

```
public class Box
{
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

JAVA: GENERIC TYPE

- **Generic class or interface parameterized over types**
- Names of **type parameters** delimited by angle brackets; names purely conventional
- Names can be freely used inside the class/interface

```
class ClassName<T1, T2, ..., Tn> { /* ... */ }
```

NAMING CONVENTIONS

- Type parameter names are single, uppercase letters
  - **E** - Element
  - **K** - Key
  - **N** - Number
  - **T** - Type
  - **V** - Value

GENERIC TYPE: EXAMPLE

- Definition

```
public class Box<T>
{
    // T stands for "Type"
    private T t;

    public void set(T newt) { t = newt; }
    public T get() { return t; }
}
```

- Instantiation: replace the generic type with some concrete value

```
Box<Integer> integerBox = new Box<Integer>();
```

C++: TEMPLATE

- **Template class**: definition

```
template <class T1, class T2, …> class ClassName { /* ... */ };
```

- **Template function**: definition

```
template <typename T1, typename T2, …> FuncName(…) { /* ... */ }
```

- Template variable (C++14): not talking about it

TEMPLATE CLASS: EXAMPLE

- Definition

```
template<class T> class Box
{
    // T stands for "Type"
    private T * t;

    public void set(T * newt) { t = newt; }
    public T * get() { return t; }
};
```

- Instantiation: replace the generic type with some concrete value

```
Box<int> * integerBox = new Box<int>();
```

TEMPLATE FUNCTION: EXAMPLE

- A parametrized quicksort

```
template<typename T> inline void swap(T& v1,T& v2)
{ T temp=v2; v2=v1; v1=temp; }

template<class T> void quicksort(T *array,int hi,int lo=0)
{
    while(hi>lo)
    {
        int i=lo; int j=hi;
        do
        {
            while(array[i]<array[lo]&&i<j) i++;
            while(array[--j]>array[lo]);
            if(i<j) swap(array[i],array[j]);
        }
        while(i<j);

        swap(array[lo],array[j]);

        if(j-lo>hi-(j+1)) {quicksort(array,j-1,lo); lo=j+1;}
        else {quicksort(array,hi,j+1); hi=j-1;}
    }
}
```

Code: java2s.com

CONCURRENCY: JAVA

- Concept of thread,
  associated with an instance of the class `Thread`.
  Every program has at least one thread
  and can create more

- Support for synchronization via the `wait()`, `notify()` (`Object` class) and `join()` methods (`Thread` class)

- Support for mutually exclusive access to resources with the `synchronized` keyword

THREAD CLASS

- Implements the interface `Runnable` with the single method `run()`, which contains the code to be run. In the standard implementation, `run()` does nothing

- Two strategies for creating a new thread

  1. Instantiate a class derived from `Thread`

  2. Create an instance of `Thread`, and pass to the constructor an object implementing `Runnable`

CREATING A THREAD (1/2)

First strategy

- Subclass `Thread` and override the `run()` method, then create an instance of the subclass

```
public class HelloThread extends Thread
{
    public void run()
    {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[])
    {
        (new HelloThread()).start();
    }
}
```

Code: docs.oracle.com

CREATING A THREAD (2/2)

Second strategy

- **Create an instance of `Thread`, pass a `Runnable` object to the constructor**

```
public class HelloRunnable implements Runnable
{
    public void run()
    {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[])
    {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Code: docs.oracle.com

THREAD CLASS: SOME METHODS

- `void start()`
  Causes the thread to begin execution
- `void setPriority(int newPriority)`
  Changes the priority of the thread
- `static void sleep(long millis, int nanos)`
  Causes the thread to pause execution for the specified number of milliseconds plus the specified number of nanoseconds
- `public final void wait(long timeout)` (inherited from `Object`)
  Causes the thread to wait until either another thread invokes the notify() method or a specified amount of time has elapsed
- `public final void notify()` (inherited from `Object`)
  Wakes up the thread
- `void join()`
  Causes the current thread to pause execution until the thread upon which `join()` has been invoked terminates. Overloads of `join()` allow to specify a waiting period

SYNCHRONIZED METHODS

- No two concurrent executions of `synchronized` methods on the same object are possible
- Mutual exclusion: invocations are serialized.
  The object behaves like it has a global lock which all its synchronized methods must acquire
  (indeed, it is exactly so)
- Constructors cannot be synchronized
  (does not make sense anyway)

EXAMPLE

- If an object is visible to more than one thread, all reads or writes to that object's variables can be done through synchronized methods to avoid some concurrency issues

```
public class SynchronizedCounter
{
    private int c = 0;

    // Mutual exclusion: no race conditions
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }

    // Mutual exclusion: no memory consistency errors
    public synchronized int value() { return c; }
}
```

Code: docs.oracle.com

SYNCHRONIZED STATEMENTS

- Any statement, or group of statements, can be declared as `synchronized` by specifying the object that provides the lock
- All accesses to the statement(s) are serialized
- Improves concurrency: only a portion of a method is serialized

EXAMPLE

- In the following code, there is no reason to prevent interleaved updates of `c1` and `c2`

```
public class MsLunch
{
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1()
    {
        synchronized(lock1) { c1++; }
    }

    public void inc2()
    {
        synchronized(lock2) { c2++; }
    }
}
```

Code: docs.oracle.com

JAVA: MORE ON CONCURRENCY

Look at the packages

- `java.util.concurrent`
- `java.util.concurrent.atomic`
- `java.util.concurrent.locks`

CONCURRENCY: C++

- Concept of thread,
associated with an instance of the class `Thread`.
Every program has at least one thread
and can create more

- Support for synchronization via the `join()` and `detach()` methods of the `Thread` class

- Support for mutually exclusive access to resources with **atomic types** and **mutex classes**

THREAD CLASS

- A thread starts immediately when an object is instantiated from the class `Thread`
- The code to be run is passed inside a function as a parameter to the constructor of `Thread`
- Further arguments for the constructor are passed as parameters to the function

EXAMPLE

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

void f(int i)
{
    std::cout << "Hello, here is an int: "
              << i << std::endl;
}

int main()
{
    std::thread t1(f, 27);

    // If you omit this call, the result is undefined
    t1.join();

    return 0;
}
```

THREAD CLASS: SOME METHODS

- **`bool joinable()`**
  Returns `true` if the thread object is *joinable*, i.e., it actually represents a thread of execution, and `false` otherwise
- **`id get_id()`**
  If the thread object is joinable, returns a value that uniquely identifies the thread
- **`void join()`**
  Causes the <u>current thread</u> to pause execution until the thread upon which `join()` has been invoked terminates
- **`void detach()`**
  Causes the current thread to be detached from the thread upon which `detach()` has been invoked

ATOMIC TYPES

- **Atomic types** are types that are guaranteed to be accessible without causing race conditions
- Some examples:

Atomic type	Contains
`atomic_bool`	`bool`
`atomic_char`	`char`
`atomic_int`	`int`
`atomic_uint`	`unsigned int`

MUTEXES

Allow mutually-exclusive access to critical sections of the source code

- **`mutex`** class
  Implements a binary semaphore. Does not support recursion (i.e., a thread shall not invoke the `lock()` or `try_lock()` methods on a mutex it already owns): use the `recursive_mutex` class for that

- **`timed_mutex`** class
  A mutex that additionally supports timed "try to acquire lock" requests (`try_lock_for(…)` method)

- **`void lock(…)`** function
  Locks all the objects passed as arguments, blocking the calling thread until all locks have been acquired

- **`bool try_lock(…)`** function
  Nonblocking variant of `lock(…)`. Returns `true` if the locks have been successfully acquired, `false` otherwise

LAST MODIFIED: APRIL 29, 2016