

Generics (генетични типове)



- Целта е сходна на целите на ООП - algorithm reusing . Механизмът е въведен в CLR на .NET
- Реализациите да се отнасят за обекти от различен тип;
- Може да се създаде 'генетичен референтен тип' , 'генетичен стойностен тип', 'генетичен интерфейс' и 'генетичен делегат'. Разбира се и 'генетичен метод'.
- Нека създадем генетичен списък: **List<T>** (произнася се : List of Tee):

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{
    public List();
    public void Add(T item);
    public void Sort( IComparer<T> comparer);
    public T[] ToArray(0);
    ....
    public Int32 Count {get;}
    ...
}
```

Обикновено се именуват с T
или Тиме (напр TKey)

- Нека използваме списъка:

```
private static void SomeMethod() {
    List<DateTime> dtList = new List<DateTime>();
```

```
    dtList.Add(DateTime.Now);           //OK! Няма boxing
    dtList.Add("2/2/2011");             //грешка при компилация
```



Предимства на генетичните (пораждащи) класове:

-Разработчикът **не е нужно да притежава сорса** на генеричния алгоритъм (за разлика от C++ templates или Java generics) за да прекомпилира;

(При шаблоните, компилаторът генерира separate source-code functions (именована: specializations) при всяко отделно повикване на ф-ия шаблон или инстанция на шаблонизиран клас.)

-Type safety

- **ясен код**: рядко се налагат type casts;

-**Подобрена производителност**: преди генетиците, същото се постигаше с използване на Object типа. Това налага непрекъснато пакетиране (boxing), което изисква памет и ресурс, форсира често включване на с-мата за garbage collection. При генетичните алгоритми няма пакетиране. Това подобрява десетки пъти производителността.



CLR средата генерира 'native code' за всеки метод, първият път когато методът се повика с указан тип данни. Това разбира се, увеличава размера на кода (при генерични реализации), но не намалява производителността





Microsoft препоръчва ползване на генетични класове от Framework Class Library (FCL) вместо не-генетичните им еквиваленти.

Съществува имплементация (на Wintellect) **Power Collection library**, която прехвърля класовете от старата Standard Template Library към CLR среда. Тя е free.

За да се поддържат генетични имплементации, към .NET се добавиха:

1. Нови IL инструкции, четящи конкретния тип на аргумента;
2. Метаданното описание се обогатява с описание на типа на параметрите;
3. Променя се синтаксисът на C#, Visual Basic и т.н.
4. Променят се компилаторите;
5. Променя се JIT компилаторът, така че да генерира 'native code' за всяко повикване с конкретен тип на аргумент.

..



Open & Closed types

Тип с генетични параметри се нарича 'open type' тъй като не допуска CLR да конструира инстанции директно (както е и при интерфейсите)

Когато кодът се обърне към генетичен тип, се подават реални параметри. Тогава типът се нарича вече 'closed type' и за него се прави инстанция.

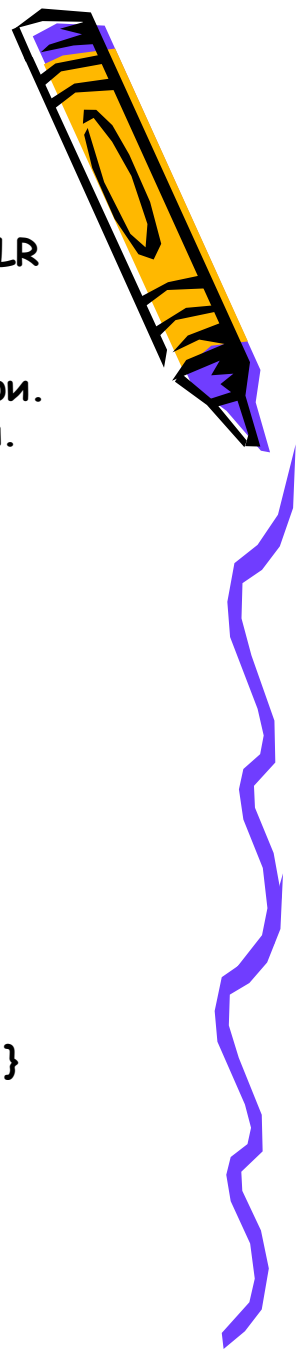
Generic types and Inheritance

Това си е нормален тип и наследяемост е напълно допустима.

```
internal sealed class Node<T> {  
    public T m_data;  
    public Node<T> m_next;  
  
    public Node(T data) : this(data,nul) {}  
    public Node(T data, Node<T> next) {  
        m_data = data; m_next = next;  
    }  
    ....}
```

Използваме в произведен тип:

```
private static void SameDataLinkedList() {  
    Node<Char> head = new Node<Char>('C');  
    head = new Node<Char>('B', head);  
    head = new Node<Char>('A', head);  
}
```



Подменяне на генетични типове

С цел удобство, е честа практика:

ако имаме: `List<DateTime> dtl = new List<DateTime>();`

да предефинираме: `internal sealed class DateTimeList : List<DateTime> {}`

И тогава можем да създадем списък от генетичен тип по традиционния начин:

`DateTimeList dtl = new DateTimeList();`

Обработка на генетични типове: code explosion

- При повикване на метод от генетичен тип, JIT компилаторът прави заместването и създава 'native code' за точно този метод с точно тези подменени параметри.
- CLR генерира native code за всеки метод/тип комбинация. Това води до 'code explosion'.
- Ако впоследствие, метод се повика със същия тип аргумент, не се генерира повторен код.
- Еднократно се генерира и код в случаите, когато сргументите са от референтен тип. Напр:

`List<String>`

`List<Stream>`

макар и аргументите всъщност да сочат съвсем различни неща.



Генетични интерфейси

Без поддръжка на генетични интерфейси, всеки път когато ще създаваме value тип, наследил не-генетичен интерфейс, трябва да се случи вътрешно пакетиране (преобразуване) на аргументите (boxing). Това е загуба на ресурс и бързодействие.

Ето един стандартен в FCL интерфейс:

```
public interface IEnumerator<T> : IDisposable, IEnumerator {  
    T Current { get; }  
}
```

Ето клас, който имплементира горния интерфейс над тип Point:

```
internal sealed class Triangle : IEnumerator<Point> {  
    private Point[] m_vertices;  
    public Point Current { get { ... } }
```

Triangle обектът може да итерира през масива от точки. Пропъртието Current е от тип Point

Генетични делегати

CLR поддържа и генетични делегати за да ползва предимствата на генетичните предавания (type-safe, без непрекъснато пакетиране например към Object).

Тъй като делегат е всъщност дефиниран клас с няколко метода (ще бъде пояснено в курса ПС) когато се дефинира делегатен тип, компилаторът поражда съответните методи с реалния тип параметри

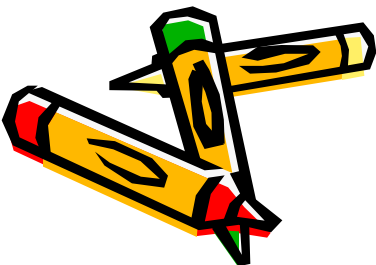




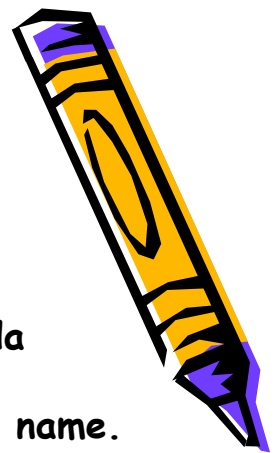
Някои особености:

- Имаме генетични **методи** (всичко в тях е нормално)
- В C# **properties, events, operator methods, constructors and finalizers** не могат да имат типови параметри. Те , обаче, могат да се дефинират вътре в генетичен тип и тогава кодът им може да ползва типовите параметри на обхващания генетичен тип директно
- Ограничители** (в генетични типове) - constraints
чрез тях може да се ограничи броя на типовете, които могат да са заместители в аргументите на генетичен тип:

```
public static T Min<T>(T o1, Y o2) where T : IComparable
{
    if(o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```



Lambda Expressions (C++)



Many programming languages support the concept of an **anonymous function**. A lambda expression is a programming technique that is related to anonymous functions.

An anonymous function is a function that has a body, but does not have a name.

A lambda expression implicitly defines **a function object class** and constructs a function object of that class type.

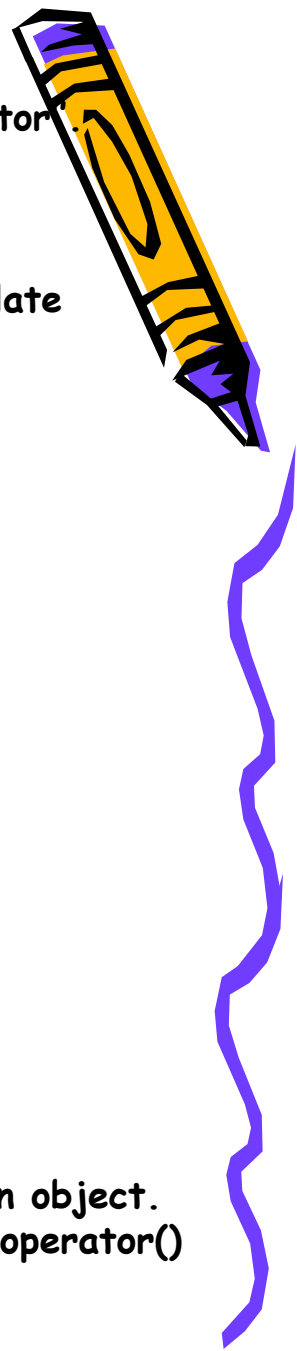
You can think of a lambda expression as an anonymous function **that maintains state** and that can access the variables that are available to the enclosing scope.

function pointers and function objects have advantages and disadvantages:

- **function pointers** involve minimal syntactic overhead, but they do not retain state within a scope;
- **function objects** can maintain state, but they require the syntactic overhead of a class definition.

Lambda expressions are a programming technique that combines the benefits of function pointers and function objects and that avoids their disadvantages. Lambda expressions are flexible and can maintain state, just like function objects, and their compact syntax removes the need for a class definition, which function objects require.





A **function object**, or **functor**, is any type that implements **operator()** – "call operator".

Function objects provide two main advantages over a straight function call.

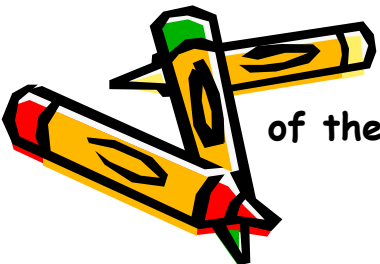
- The first is that a **function object can contain state**.
- The second is that a **function object is a type** and therefore can be used as a template parameter.

To create a function object, create a type and implement **operator()**, such as:

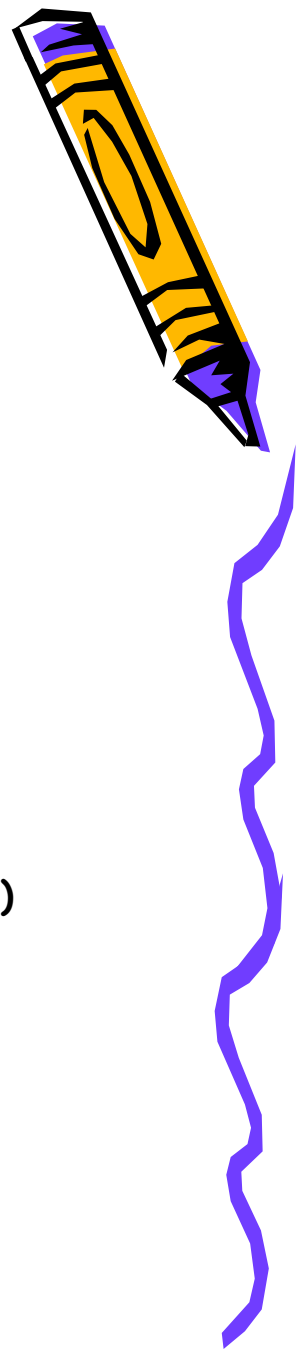
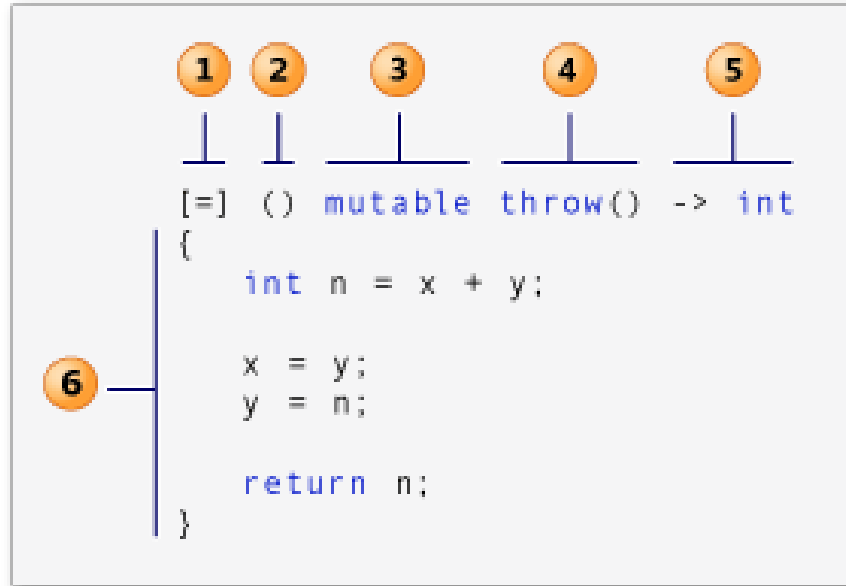
```
class Functor
{
public:
    int operator()(int a, int b)
    {
        return a < b;
    }
};
```

```
int main()
{
    Functor f;
    int a = 5;
    int b = 7;
    int ans = f(a, b);
}
```

The last line of the main function shows how you call the function object. This call looks like a call to a function, but it is actually calling **operator()** of the **Functor** type.



Lambda Expression Syntax



- 1. **lambda-introducer** (referred to as **capture clause** later in this topic)
- 2. **lambda-parameter-declaration-list** (referred to as parameter list later in this topic)
- 3. **mutable-specification** (referred to as mutable specification later in this topic)
- 4. **exception-specification** (referred to as exception specification later in this topic)
- 5. **lambda-return-type-clause** (referred to as return type later in this topic)
- 6. **compound-statement** (referred to as lambda body later in this topic)





Capture Clause

A lambda expression can access any variable that has automatic storage duration and that can be accessed in the enclosing scope - by value or by reference: variables that have the ampersand (&) prefix are accessed by reference and variables that do not have the & prefix are accessed by value. The empty capture clause, [], indicates that the body of the lambda expression accesses no variables in the enclosing scope.

You can specify the default capture mode in the syntax by specifying & or = as the first element of the capture clause - & specifies that all captured variables by reference; the = specifies that the body of the lambda expression accesses all captured variables by value.

For example, if the body of a lambda expression accesses the external variable `total` by reference and the external variable `factor` by value, then the following capture clauses are equivalent:

```
[&total, factor]
```

```
[&, factor]
```

```
[=, &total]
```

Parameter List

The parameter list for a lambda expression resembles the parameter list for a function, with the following exceptions:

- The parameter list cannot have default arguments.
- The parameter list cannot have a variable-length argument list.
- The parameter list cannot have unnamed parameters.

The parameter list for a lambda expression is optional

Example:

```
int z = [=] { return x + y; }();
```



Mutable Specification

The mutable specification part enables the body of a lambda expression to modify variables that are captured by value

You can use the **throw()** exception specification to indicate that the lambda expression does not throw any exceptions

The **return type** part of a lambda expression resembles the return type part of an ordinary method or function. However, the return type follows the parameter list and you must include **->** before the return type.

```
int main()
{
    int m = 0, n = 0;
    [&, n] (int a) mutable { m = ++n + a; }(4);
    cout << m << endl << n << endl;
}
```

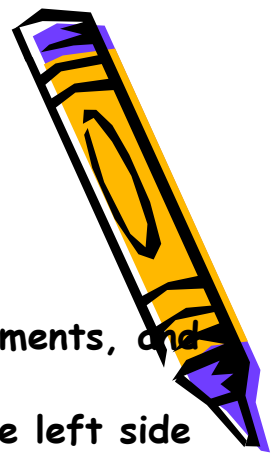
This example prints the following to the console:

```
5
0
```

Because the variable **n** is captured by value, its value remains **0** after the call to the lambda expression



Lambda Expressions (C# Programming Guide)



A lambda expression is an anonymous function that can contain expressions and statements, and **can be used to create delegates**.

All lambda expressions use the lambda operator =>, which is read as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression or statement block. The lambda expression `x => x * x` is read "x goes to x times x." This expression can be assigned to a delegate type as follows:

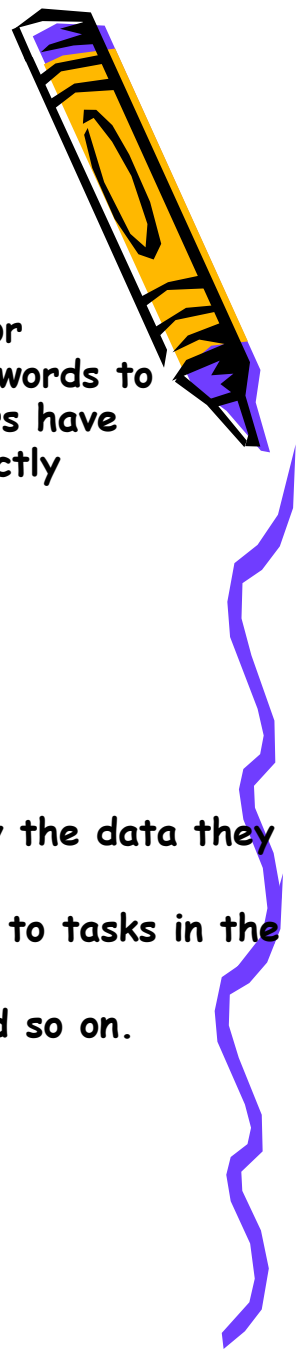
```
delegate int del(int i);  
static void Main(string[] args)  
{ del myDelegate = x => x * x;  
  int j = myDelegate(5); //j = 25 }
```

Example of another usage of lambda expression:

```
// Use a lambda expression to define an event handler:  
this.Click += (s, e) => { MessageBox.Show(((MouseEventArgs)e).Location.ToString());};
```



New Standard Concurrency Features in Visual C++ 11



The C++ iteration, known as C++11 and approved by the International Organization for Standardization (ISO) in 2011, formalizes a new set of libraries and a few reserved words to deal with concurrency- ones of the main improvements in the release. Many developers have used concurrency in C++ before, but always through a third-party library—often directly exposing OS APIs

we'll learn about the following:

Asynchronous tasks: those smaller portions of the original algorithm only linked by the data they produce or consume.

Threads: units of execution administrated by the runtime environment. They relate to tasks in the sense that tasks are run on threads.

Thread internals: thread-bound variables, exceptions propagated from threads and so on.



1. Asynchronous Tasks

Some Sequential Case Code:

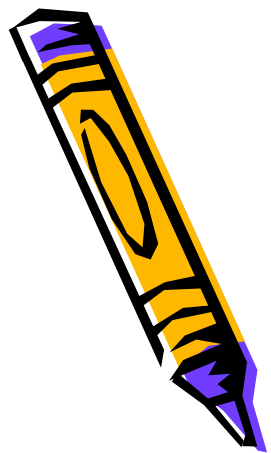
```
int a, b, c;
int calculateA()      { return a+a*b; }
int calculateB()      { return a*(a+a*(a+1)); }
int calculateC()      { return b*(b+1)-b; }
int main(int argc, char *argv[])
{  getUserData(); // initializes a and b
  c = calculateA() * (calculateB() + calculateC());
  showResult();
}
```

The main function asks the user for some data and then submits that data to three functions: calculateA, calculateB and calculateC. The results are later combined to produce some output information for the user.

Imagine, the calculating functions in the companion material are coded in a way such that a random delay between one and three seconds is introduced in each. Considering that these steps are executed sequentially, this leads to an overall execution time—once the input data is entered—of nine seconds.

As these functions are independent, we can execute them in parallel by using the async function:

```
int main(int argc, char *argv[])
{
    getUserData();
    future<int> f1 = async(calculateB), f2 = async(calculateC);
    c = (calculateA() + f1.get()) * f2.get();
    showResult();
}
```





we've introduced two concepts here: **async and future**, both defined in the `<future>` header and the `std` namespace.

The first one receives a function, a lambda or a function object (functor) and returns a future. You can understand the concept of a future as the placeholder for an eventual result.

Which result? The one returned by the function called asynchronously.

At some point, we'll need the results of these parallel-running functions. Calling **the get** method on each future blocks the execution until the value is available.

The worst-case delay of this modification is about three seconds versus nine seconds for the sequential version.

2. Threads

The asynchronous task model presented in the previous section might suffice in some given scenarios, but if you need a deeper handling and control of the execution of threads, C++11 comes with the **thread class**, declared in the `<thread>` header and located in the `std` namespace.

Despite being a more complex programming model, threads offer better methods for synchronization and coordination, allowing them to yield execution to another thread and wait for a determined amount of time or until another thread is finished before continuing.

In the following example, we have a lambda function, which, given an integer argument, prints its multiples of less than 100,000 to the console:





```
auto multiple_finder = [](int n) {  
    for (int i = 0; i < 100000; i++)  
        if (i%n==0)  
            cout << i << " is a multiple of " << n << endl; };
```

```
int main(int argc, char *argv[])  
{  
    thread th(multiple_finder, 23456);  
    multiple_finder(34567);  
    th.join();  
}
```

As you'll see in later examples, the fact that we passed a lambda to the thread is circumstantial; a function or functor would've sufficed as well.

In the main function we run this function in two threads with different parameters. Take a look at the result (which could vary between different runs due to timings):

```
0 is a multiple of 23456  
0 is a multiple of 34567  
23456 is a multiple of 23456  
34567 is a multiple of 34567  
46912 is a multiple of 23456  
69134 is a multiple of 34567  
70368 is a multiple of 23456  
93824 is a multiple of 23456
```



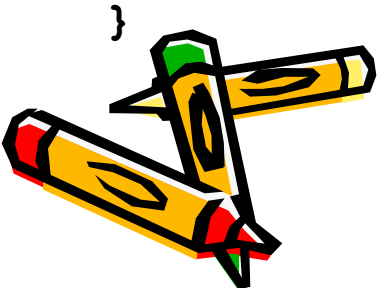
we might implement the example about asynchronous tasks in the previous section with threads. For this, we need to introduce **the concept of a promise**. A promise can be understood as a sink through which a result will be dropped when available. Where will that result come out once dropped? Each promise has an associated future.

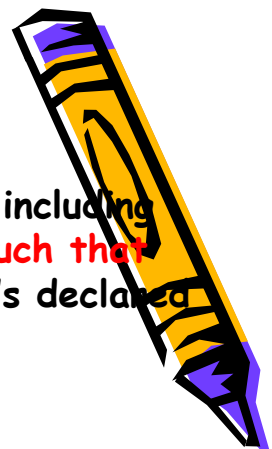
The code shown, associates three threads (instead of tasks) with promises and makes each thread call a calculate function. Compare these details with the lighter AsyncTasks version.

```
typedef int (*calculate)(void);

void func2promise(calculate f, promise<int> &p)
{ p.set_value(f()); }

int main(int argc, char *argv[])
{
    getUserData();
    promise<int> p1, p2;
    future<int> f1 = p1.get_future(), f2 = p2.get_future();
    thread t1(&func2promise, calculateB, std::ref(p1)),
            t2(&func2promise, calculateC, std::ref(p2));
    c = (calculateA() + f1.get()) * f2.get();
    t1.join(); t2.join();
    showResult();
}
```





3. Thread-Bound Variables and Exceptions

In C++ you can define **global variables** whose scope is bound to the entire application, including threads. But relative to threads, now there's a way **to define these global variables such that every thread keeps its own copy**. This concept is known as thread local storage and it's declared as follows:

```
thread_local int subtotal = 0;
```

If the declaration is done in the scope of a function, the visibility of the variable will be narrowed to that function but each thread will keep maintaining its own static copy.

*Although **thread_local** isn't available in Visual C++ 11, it can be simulated with a non-standard Microsoft extension:*

```
#define thread_local __declspec(thread)
```

Syncing up Concurrent Execution

It would be desirable if all applications could be split into a 100 percent-independent set of asynchronous tasks. In practice this is almost never possible, as there are at least dependencies on the data that all parties concurrently handle. This section introduces new C++11 technologies **to avoid race conditions**.



- Atomic types:** similar to primitive data types, but enabling thread-modification.
- Mutexes and locks:** elements that enable us to define thread-safe critical regions.
- Condition variables:** a way to freeze threads from execution until some criteria is satisfied.

Atomic Types



The `<atomic>` header introduces a series of primitive **types: `atomic_char`, `atomic_int`** and so on—implemented in terms of interlocking operations. Thus, these types are equivalent to their homonyms without the `atomic_` prefix but with the difference that all their assignment operators (`=`, `++`, `--`, `+=`, `*=` and so on) are protected from race conditions.

In the following example there are two parallel threads (one being the main) looking for different elements within the same vector:

```
atomic_uint total_iterations;
vector<unsigned> v;
int main(int argc, char *argv[])
{ total_iterations = 0; scramble_vector(1000);
  thread th(find_element, 0);
  find_element(100);
  th.join();
  cout << total_iterations << " total iterations." << endl;
}
```

```
void find_element(unsigned element)
{ unsigned iterations = 0;
  find_if(begin(v), end(v), [=, &iterations](const unsigned i) -> bool {
    ++iterations;
    return (i==element);
  });
  total_iterations+= iterations;
  cout << "Thread #" << this_thread::get_id() << ": found after " <<
    iterations << " iterations." << endl;
}
```



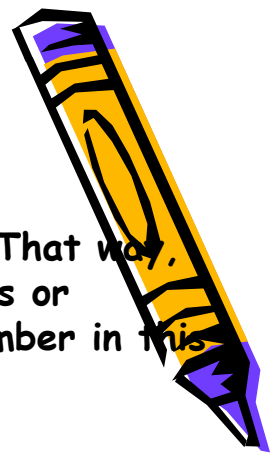
Mut(ual) Ex(clusion) and Locks

The `<mutex>` header defines a series of lockable classes to define critical regions. That way, you can define a mutex to establish a critical region throughout a series of functions or methods, in the sense that only one thread at a time will be able to access any member in this series by successfully locking its mutex.

A thread attempting to lock **a mutex** can either stay blocked until the mutex is available or just fail in the attempt. In the middle of these two extremes, the alternative **timed_mutex** class can stay blocked for a small interval of time before failing. Allowing lock attempts to desist helps prevent deadlocks.

A locked mutex must be explicitly unlocked for others to lock it. Failing to do so could lead to an undetermined application behavior—which could be error-prone, similar to forgetting to release dynamic memory. Forgetting to release a lock is actually much worse, because it might mean that the application can't function properly anymore if other code keeps waiting on that lock. Fortunately, C++11 also comes with locking classes. A lock acts on a mutex, but its destructor makes sure to release it if locked.

The following code defines a critical region around a mutex mx:





```
mutex mx;
```

```
void funcA();
```

```
void funcB();
```

```
int main()
```

```
{  thread th(funcA) funcB();  
    th.join();
```

```
}
```

This mutex is used to guarantee that two functions, funcA and funcB, can run in parallel without coming together in the critical region.

The function funcA will wait, if necessary, in order to come to the critical region. In order to make it do so, you just need the simplest locking mechanism—**lock_guard**:

```
void funcA()
```

```
{  for (int i = 0; i<3; ++i)
```

```
{    this_thread::sleep_for(chrono::seconds(1));
```

```
    cout << this_thread::get_id() << ": locking with wait... " << endl;
```

```
    lock_guard<mutex> lg(mx);
```

```
    ... // Do something in the critical region.
```


```
    cout << this_thread::get_id() << ": releasing lock." << endl;
```

```
}
```

```
}
```



The way it's defined, funcA should access the critical region three times. The function funcB, instead, will attempt to lock, but if the mutex is by then already locked, funcB will just wait for a second before again attempting to get access to the critical region. The mechanism it uses is **unique_lock** with the policy **try_to_lock_t**, as shown:



void funcB()

```
{ int successful_attempts = 0;
  for (int i = 0; i<5; ++i)
  { unique_lock<mutex> ul(mx, try_to_lock_t());
    if (ul)
    { ++successful_attempts;
      cout << this_thread::get_id() << ": lock attempt successful." << endl;
      // Do something in the critical region
      cout << this_thread::get_id() << ": releasing lock." << endl;
    } else {
      cout << this_thread::get_id() <<
        ": lock attempt unsuccessful. Hibernating..." << endl;
      this_thread::sleep_for(chrono::seconds(1));
    }
  }
  cout << this_thread::get_id() << ": " << successful_attempts
    << " successful attempts." << endl;
}
```

The way it's defined, funcB will try up to five times to enter the critical region. Following are the results of the execution.

Out of the five attempts, funcB could only come to the critical region twice.

```
funcB: lock attempt successful.
funcA: locking with wait ...
funcB: releasing lock.
funcA: lock secured ...
funcB: lock attempt unsuccessful.
Hibernating ...
funcA: releasing lock.
funcB: lock attempt successful.
funcA: locking with wait ...
funcB: releasing lock.
funcA: lock secured ...
funcB: lock attempt unsuccessful.
Hibernating ...
funcB: lock attempt unsuccessful.
Hibernating ...
funcA: releasing lock.
funcB: 2 successful attempts.
funcA: locking with wait ...
funcA: lock secured ...
funcA: releasing lock.
```



Condition Variables

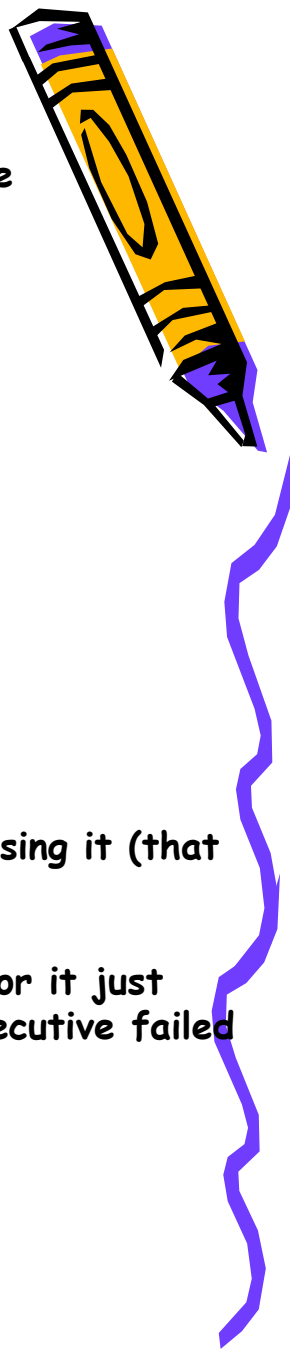
The header `<condition_variable>` comes with the last facility, fundamental for those cases when coordination between threads is tied to events.

a producer function pushes elements in a queue:

```
mutex mq;  
condition_variable cv;  
queue<int> q;  
void producer()  
{ for (int i = 0; i < 3; ++i) {    ... // Produce element  
    cout << "Producer: element " << i << " queued." << endl;  
    mq.lock(); q.push(i); mq.unlock();  
    cv.notify_all();  
}  
}
```

The standard queue isn't thread-safe, so you must make sure that nobody else is using it (that is, the consumer isn't popping any element) when queuing.

The consumer function attempts to fetch elements from the queue when available, or it just waits for a while on the condition variable before attempting again; after two consecutive failed attempts, the consumer ends:




```
void consumer()
{ unique_lock<mutex> l(m);
  int failed_attempts = 0;
  while (true) {
```

```
    mq.lock();
    if (q.size())
    { int elem = q.front();
      q.pop();
      mq.unlock();
      failed_attempts = 0;
      cout << "Consumer: fetching " << elem << " from queue." << endl;
      ... // Consume elem
    } else {
```

```
        mq.unlock();
        if (++failed_attempts>1)
        {
            cout << "Consumer: too many failed attempts -> Exiting." << endl;
            break;
        } else {
            cout << "Consumer: queue not ready -> going to sleep." << endl;
            cv.wait_for(l, chrono::seconds(5));
        }
    }
}
```

```
Consumer: queue not ready -> going to sleep.
Producer: element 0 queued.
Consumer: fetching 0 from queue.
Consumer: queue not ready -> going to sleep.
Producer: element 1 queued.
Consumer: fetching 1 from queue.
Consumer: queue not ready -> going to sleep.
Producer: element 2 queued.
Producer: element 3 queued.
Consumer: fetching 2 from queue.
Producer: element 4 queued.
Consumer: fetching 3 from queue.
Consumer: fetching 4 from queue.
Consumer: queue not ready -> going to sleep.
Consumer: two consecutive failed attempts -> Exiting
```



The consumer is to be awoken via **notify_all** by the producer every time a new element is available. That way, the producer avoids having the consumer sleep for the entire interval if elements are ready.