

## Standard Template Library 2

Mehrnoosh Askarpour & Danilo Ardagna

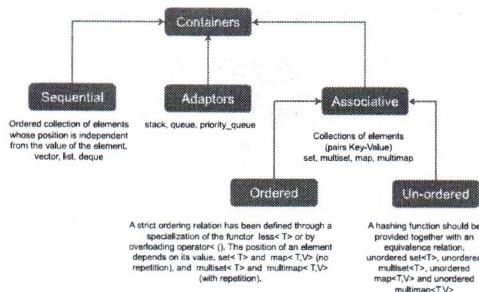
Politecnico di Milano  
[name.lastname@polimi.it](mailto:name.lastname@polimi.it)



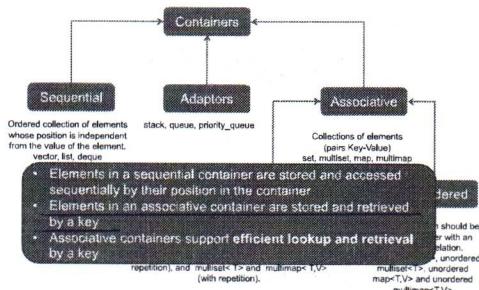
### Content

- Adaptors
- Associative Containers
  - set, map
  - multiset, multimap
  - Unordered map, set

### Sequential and Associative containers



### Sequential and Associative containers



### Adaptors

are not new containers, are usually based on sequential containers and just add some restrictions on how data are organized internally and how we can access the elements

## Container Adaptors (readings)

- Container adaptors are interfaces created on top of a (limited) set of functionalities of a pre-existing sequential container, which provide a different API
- When you declare the container adapters, you have an option of specifying which sequential container to use as underlying container

## Container Adaptors (readings)

- **stack:**
  - Container providing Last-In, First-Out (LIFO) access
  - You remove (**pop**) elements in the reverse order you insert (**push**) them. You cannot get any elements in the middle
  - Usually this goes on top of a **deque**
- **queue:**
  - Container providing First-In, First-Out (FIFO) access
  - You remove (**pop**) elements in the same order you insert (**push**) them. You cannot get any elements in the middle but only **front** and **back**
  - Usually this goes on top of a **deque**
- **priority\_queue:**
  - Container providing sorted-order access to elements
  - You can insert (**push**) elements in any order, and then retrieve (**pop**) the "highest priority" of these values at any time
  - Priority queues in C++ STL use a **heap** structure internally, which in turn is basically **array-backed**; thus, usually this goes on top of a **vector**

## Associative Containers

### Overview of the Associative Containers

Associative containers support the **general container operations**. However they **do not support the sequential-container position-specific operations**, such as **push\_front**. Because the elements are stored based on their **keys**, these operations would be meaningless for the associative containers.

Nevertheless they have:

- Type aliases
- Bidirectional iterators
- Specific operations
- Hash functions (unordered version)

### Overview of the Associative Containers

- **map:** holds key-value pairs
- **multimap:** a map in which one key can appear multiple times
- **set:** the key is the value
- **multiset:** a set in which a key can appear multiple times
- **unordered-map:** a map organized by a hash function
- **unordered-multimap:** hashed map, keys can appear multiple times
- **unordered-set:** a set organized by a hash function
- **unordered-multiset:** hashed set, keys can appear multiple times

Consider multi-version as readings

## std::map

- A map is a collection of `<key, value>` pairs with unique keys
  - It is often referred to as an **associative array**
    - An associative array is like a "normal" array except that its subscripts don't have to be integers
  - Values in a map are found by a key rather than by their position
- Example: Given a map of names to phone numbers, we'd use a person's name as a subscript to fetch that person's phone number:
- ```
E.g.: cout << phone_numbers["Mario"];
```

## std::map

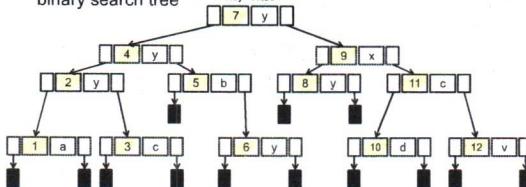
```
// count the number of times each word occurs
// in the input
map<string, size_t> word_count;
string word;
while (cin >> word) // fetch and increment the
    // counter for word
    ++word_count[word];

for (const auto &w : word_count)
    // for each element in the map print the results
    cout << w.first << " occurs " <<
        w.second <<
        ((w.second > 1) ?
            " times" : " time") << endl;
```

if the word is not included in the map we introduce it as a new one, if it's already included we just increment the counter

## Implementation of a map

A map is implemented by red-black trees, self-balancing binary search tree



Insert and delete  $O(\log n)$  at worst case. The rules keep the tree balanced

very fast data structure

## std::set

- A set is simply a collection of objects. A set is most useful when we simply want to know whether a value is present

- Example: a business might define a set named `bad_checks` to hold the names of individuals who have written bad checks. Before accepting a check, that business would query `bad_checks` to see whether the customer's name was present

```
E.g.:
if (bad_checks.find("Mario") == bad_checks.end())
    cout << "Mario is ok!";
else
    cout << "Mario is bad guy!";
```

## std::set

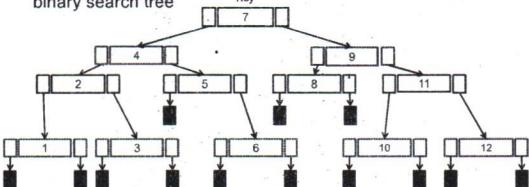
- Example: excluding certain words from the map of word occurrences

```
// count the number of times each word occurs
// in the input
map<string, size_t> word_count;
set<string> exclude = {"The", "But", "And", "Or",
    "An", "A", "the", "but",
    "and", "or", "an", "a"};
string word;
while (cin >> word)
    if (exclude.find(word) == exclude.end())
        ++word_count[word];
```

we don't want word to be one of the exclude's

## Implementation of a set

A set is implemented by red-black trees, self-balancing binary search tree



Insert and delete  $O(\log n)$  at worst case. The rules keep the tree balanced

## Set and Multiset - Example

```
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i); // duplicate copies of each
    ivec.push_back(i); // number
}
set<int> iset(ivec.cbegin(), ivec.cend());
multiset<int> miset(ivec.cbegin(), ivec.cend());

cout << ivec.size() << endl; // prints 20
cout << iset.size() << endl; // prints 10
cout << miset.size() << endl; // prints 20
```

we create a set of integers from the vector  
note: we cannot write something like:  
X ~~set<int> iset(ivec);~~

the set ignores replicas

## Associative vs. Sequential containers

- The associative containers do not support
  - position-specific operations, e.g., `push_front` or `back`
  - constructors or insert operations that take an element value and a count
- The associative container iterators are always **bidirectional**

## Requirements on Key Type

For the ordered containers the key type must define a way to compare the elements:

- By default, the library uses the `<` to compare the keys
- We can also supply our own `<` operation to use a **strict weak ordering** over the key type
  - Two keys cannot both be "less than" each other
  - If  $k_1$  is "less than"  $k_2$  and  $k_2$  is "less than"  $k_3$ , then  $k_1$  must be "less than"  $k_3$
  - If there are two keys, and neither key is "less than" the other, then we'll say that those keys are "equivalent." If  $k_1$  is "equivalent" to  $k_2$  and  $k_2$  is "equivalent" to  $k_3$ , then  $k_1$  must be "equivalent" to  $k_3$

DEMO

## The pair type

A pair is a library type, defined in the utility header, which holds two data members.

```
// holds two strings
pair<string, string> anon;

// holds a string and an size_t
pair<string, size_t> word_count;

// holds string and vector<int>
pair<string, vector<int>> line;
pair<string, string> author{"James", "Joyce"};
```

## The pair type

- The data members of pair are public
- These members are named `first` and `second`
- Elements in a map are pairs
- Only a limited number of operations defined in the library

|                                                     |                                                                                                                                                                                                |
|-----------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>pair&lt;T1, T2&gt; p;</code>                  | Pair definition with or without initialization                                                                                                                                                 |
| <code>pair&lt;T1, T2&gt; p(v1, v2);</code>          |                                                                                                                                                                                                |
| <code>pair&lt;T1, T2&gt; p=(v1, v2);</code>         |                                                                                                                                                                                                |
| <code>make_pair(v1, v2)</code>                      | Pair definition. Type of pair is inferred from v1 and v2 type.                                                                                                                                 |
| <code>p.first / p.second</code>                     | Returns first or second member of p                                                                                                                                                            |
| <code>p1(&lt;, &gt;, &lt;=, &gt;=, ==, !=)p2</code> | Relational operators and equality.<br>For example <code>p1 &lt; p2</code> if <code>p1.first &lt; p2.first</code> or <code>!(p2.first &lt; p1.first) &amp;&amp; p1.second &lt; p2.second</code> |

"pair" are structures with two members which are named "first" and "second"

## Associative Container Type Aliases

- `key_type`: type of the key of the container
- `mapped_type`: type associated with each `map` key (= just the value)
- `value_type`: it is the same as `key_type` for sets and `pair<const key_type, mapped_type>` for maps.

## Associative Container Type Aliases

- `key_type`: type of the key of the container
- `mapped_type`: type associated with each `map` key
- `value_type`: it is the same as `key_type` for sets and `pair<const key_type, mapped_type>` for maps.

```
set<string>::value_type v1; // v1 is a string
set<string>::key_type v2; // v2 is a string
map<string, int>::value_type v3; // v3 is a
                                // pair<const string, int>
map<string, int>::key_type v4; // v4 is a string
map<string, int>::mapped_type v5; // v5 is an int
```

in sets keys and values are basically the same

## key is const!

Remember that the `value_type` of a map is a pair and that we can change the value but not the key member of that pair

```
auto map_it = word_count.begin();
cout << map_it->first << " " << map_it->second;

// error: key is const
map_it->first = "new key";
```

## key is const!

The keys in a `set` are also `const`. We can use a `set` iterator to read, but not write, an element's value

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    // error: keys in a set are read-only
    *set_it = 42;
    // ok: can read the key
    cout << *set_it << endl;
}
```

## Iterating across an associative container

When we use an iterator to traverse a map, multimap, set, or multiset, the iterators yield elements in **ascending key order**

```
auto map_it = word_count.cbegin();
while (map_it != word_count.cend()) {
    cout << map_it->first << " occurs "
    << map_it->second << " times" << endl;
    ++map_it;
}
```

} the ordering with maps comes for free

## Adding Elements

Because (unordered) map and (unordered) set contain unique keys, inserting elements that are already present, has no effect

|                 |                                                                                                                                                                                                                                                                                                                                                          |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| c.insert(v)     | v value_type object. args are used to construct an element. Insert in map or set only if an element with the given key already is not in c. <b>Return a pair of an iterator referring to the element with the given key and a bool indicating whether the element was inserted.</b> For multimap and multiset it returns an iterator to the new element. |
| c.insert(b, e)  | b and e are iterators denoting a range of c::value_type elements                                                                                                                                                                                                                                                                                         |
| c.insert(il)    | il is a braced list of values                                                                                                                                                                                                                                                                                                                            |
| c.insert(p, v)  | Like the first two, but uses p as a hint for where to begin the search for where the new element should be stored. Returns an iterator to the element with the given key                                                                                                                                                                                 |
| c.emplace(args) |                                                                                                                                                                                                                                                                                                                                                          |

} optimization that we can do (however is not that important)

## Adding Elements

Because (unordered) map and (unordered) set contain unique keys, inserting elements that are already present, has no effect

```
vector<int> ivec = {2,4,6,8,2,4,6,8}; // ivec has
   // eight elements
set<int> set2; //empty set
set2.insert(ivec.begin(), ivec.end()); // set2 has
   // four elements
set2.insert({1,3,5,7,1,3,5,7}); // set2 now has
   // eight elements
```

} because ivec has copies

} we have 4 new elements (because of the copies) and added to the previous 4 we now have 8 elements

## Adding Elements

```
multimap<string, string> authors;
// adds the first element with the
// key Barth, John
authors.insert({"Barth, John", "Sot-Weed
Factor"});

// ok: adds the second element with the key
// Barth, John
authors.insert({"Barth, John", "Lost in the
Funhouse"});

Here insert returns always only an iterator to
the inserted element
```

## Erasing Elements

We can erase one element or a range of elements by passing `erase` an iterator or an iterator pair.

|              |                                                                                                      |
|--------------|------------------------------------------------------------------------------------------------------|
| c.erase(k)   | Removes every element with key k from c. Returns size_type indicating the number of removed elements |
| c.erase(p)   | Removes the element denoted by the iterator p. Returns an iterator to the element after p            |
| c.erase(b,e) | Removes elements in the range from b to e, and returns e                                             |

## Subscripting a map

- The `map` and `unordered_map` containers provide the **subscript operator** and a corresponding `at` function. The **set types do not support subscripting** because there is no "value" associated with a key in a set.
- We cannot subscript a `multimap` or an `unordered_multimap`** because there may be more than one value associated with a given key.

|                      |                                                                                                                                                            |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>c[k]</code>    | Returns the element with key <code>k</code> ; if <code>k</code> is not in <code>c</code> , adds a new, value initialized element with key <code>k</code> . |
| <code>c.at[k]</code> | Checked access to the element with key <code>k</code> ; <code>out_of_range</code> error if <code>k</code> is not in <code>c</code> .                       |

## Subscripting a map

```
map <string, size_t> word_count;
// insert a value-initialized element with
// key Anna; then assign 1 to its value
word_count["Anna"] = 1;
```

— here the map is empty  
— we're explicitly adding "Anna" once

! The point is: we first insert a new element of the map with key = "Anna" and value STANDARD INITIALIZED, only then WE PERFORM THE ASSIGNMENT



## Subscripting a map

```
map <string, size_t> word_count;
// insert a value-initialized element with
// key Anna; then assign 1 to its value
word_count["Anna"] = 1;
```

- `word_count` is searched for the element whose key is `Anna`.
- The element is not found.
- A new key-value pair is inserted into `word_count`.
  - The key is a const string holding `Anna`.
  - The value is value initialized, i.e., it takes 0.
  - The newly inserted element is fetched set to 1.

## Subscripting a map

```
map <string, size_t> word_count;
// insert a value-initialized element with
// key Anna; then assign 1 to its value
word_count["Anna"] = 1;
```

• Another way:  
`word_count.insert(make_pair("Anna", 1));`

alternative:  
`word_count.insert(make_pair("Anna", 1));`

## Subscripting a map

```
map <string, size_t> word_count;
// insert a value-initialized element with
// key Anna; then assign 1 to its value
word_count["Anna"] = 1;
// fetch the element indexed by Anna
cout << word_count["Anna"];
// fetch the element and add 1 to it
++word_count["Anna"];
// fetch the element and print it
cout << word_count["Anna"];
```

as result: 2

DEMO

## Accessing Elements

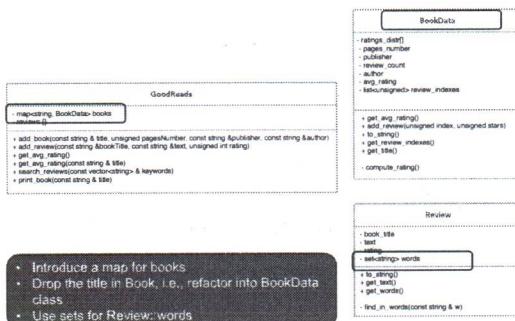
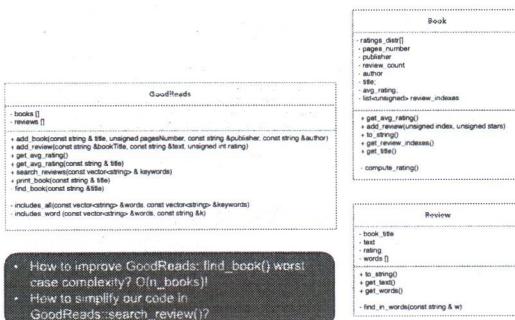
|                               |                                                                                                                        |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------|
| <code>c.find(k)</code>        | Returns an iterator to the <b>first element with key k</b> , or the off-the-end iterator if k is not in the container. |
| <code>c.count(k)</code>       | Returns the number of elements with key k. For the containers with unique keys, the result is always zero or one.      |
| <code>c.lower_bound(k)</code> | Return an iterator to the <b>first element with key not less than k</b> .                                              |
| <code>c.upper_bound(k)</code> | Return an iterator to the <b>first element with key greater than k</b> .                                               |

} bely on the order: in the unordered version we cannot use them

Lower and upper bound are not valid for unordered containers, in that case you can rely on `equal_range`

## Accessing Elements

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
auto it = iset.find(1); // returns an iterator that
// refers to the element with key 1
it = iset.find(11); // returns the
// iterator == iset.end()
iset.count(1); // returns 1
iset.count(11); // returns 0
```



## A running example - GoodReads

```
class GoodReads {
    map<string, BookData> books; // <title, BookData>
    vector<Review> reviews;
public:
    void add_book(const string & title, unsigned pagesNumber,
                  const string & publisher,
                  const string & author);
    void add_review(const string &bookTitle,
                   const string &text, unsigned int rating);
    float get_avg_rating() const;
    float get_avg_rating(const string & title) const;
    void search_reviews(const vector<string> & keywords) const;
    void print_book(const string & title) const;
};
```

## A running example - GoodReads

```
class BookData {
    vector<unsigned> ratings_distr;
    unsigned pages_number;
    string publisher;
    unsigned review_count;
    string author;
    float avg_rating;
    list<unsigned> review_indexes;
public:
    BookData(unsigned int pagesNumber, const string &publisher, const string &author);
    float get_avg_rating() const;
    void add_review(unsigned index, unsigned stars);
    string to_string() const;
    list<unsigned> get_review_indexes() const;
private:
    float compute_rating();
};
```

## A running example - GoodReads

```
class Review {
    string book_title;
    string text;
    unsigned rating;
    set<string> words;
public:
    Review(const string &bookTitle, const
           string &text, unsigned int rating);
    string to_string() const;
    string get_text() const;
    set<string> get_words() const;
};
```

## A running example - GoodReads

- find book is not needed anymore, book search performed as:
- const auto it = books.find(title);
- and if it == books.cend() the book is not in the collection otherwise it can be accessed through it

Worst case complexity now becomes  $O(\log(n\_books))$   
(instead of  $O(n\_books)$ )

## A running example - GoodReads

- How to simplify keywords search in `search_reviews()`
- Given two sets a and b you can test if a  $\supseteq$  b by:
- `std::includes(a.cbegin(), a.cend(),
 b.cbegin(), b.cend())`
- where `std::includes()` is defined in algorithm and returns a bool



## A running example - GoodReads

```
void GoodReads::search_reviews(const vector<string> & keywords)
{
    const set<string> keywords_set(keywords.cbegin(), keywords.cend());

    for (auto it = reviews.cbegin(); it != reviews.cend(); ++it){
        const set<string> & words = it->get_words();

        if(std::includes(words.cbegin(), words.cend(),
                        keywords_set.cbegin(), keywords_set.cend()))
            cout << it->to_string() << endl;
    }
}
```

## GoodReads method complexity

### Vector based

- Book::add\_review()
  - push\_back in a vector
  - Worst case complexity  $O(n\_reviews)$
- GoodReads::find\_book()
  - Sequential search in a vector
  - Worst case complexity  $O(n\_books)$
- GoodReads::add\_book()
  - find\_book and push\_back in a vector
  - Worst case complexity  $O(n\_books)$
- GoodReads::get\_avg\_rating()
  - Sequential access in a vector
  - Worst case complexity  $O(n\_books)$
- GoodReads::get\_avg\_rating(
  - const string & title
  - find\_book
- Worst case complexity  $O(n\_books)$

### Optimized version

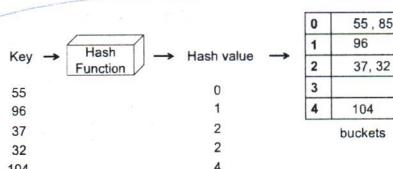
- Book::add\_review()
  - push\_back in a list
  - Worst case complexity  $O(1)$
- GoodReads::find\_book()
  - find in a map
  - Worst case complexity  $O(\log(n\_books))$
- GoodReads::add\_book()
  - find and insert in a map
  - Worst case complexity  $O(\log(n\_books))$
- GoodReads::get\_avg\_rating()
  - Sequential access in a map
  - Worst case complexity  $O(n\_books)$
- GoodReads::get\_avg\_rating(
  - const string & title
  - find in a map
- Worst case complexity  $O(\log(n\_books))$

## unordered\_map & unordered\_set

## Unordered Associative Containers

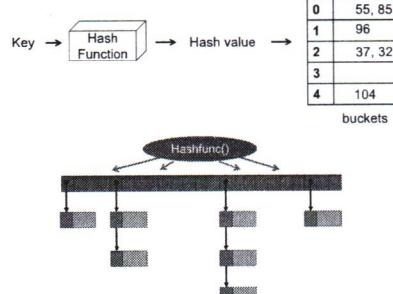
- Collection of buckets
  - Each bucket contains a variable number of items
- Use of a **hash function** to map elements to buckets
  - Given the item key, identify the proper bucket to store such item
  - All of the elements with a given hash value are stored in the same bucket
  - All the elements with the same key (in the multi- version) will be in the same bucket

## Unordered Associative Containers



Different keys with the same hash value are stored in the same bucket and originate a collision!

## Unordered Associative Containers



How do we organize a structure like this?

## Unordered Associative Containers

If we have many collision →

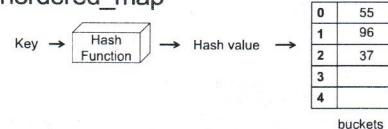


## Unordered Associative Containers

- The performance of an unordered container depends on the quality of its hash function and on the number and size of its buckets:
  - Average complexity  $O(1)$
  - Worst case complexity  $O(N)$  (if all the elements have collision (all stored in one element))
- Rather than using a comparison operation to organize their elements, these containers use a hash function and the key type's == operator
- Use an unordered container if the key type is inherently unordered or if performance testing reveals problems that hashing might solve

DEMO

### A Simple Hash Function to Manage an unordered\_map



```
unsigned my_hash_func(unsigned x, unsigned size)
{
    return x % size;
}
```

In APSC you will see how to define your hash functions. In APC we will rely on the ones defined for the built-in and STL types

} we will use only the already defined hash function (true for the base types: integers, characters, string, ...). However, we'll never use an unordered map for a type that we create (e.g. no unordered\_map for Sales date class))

## Using an Unordered Container

```
unordered_map<string, size_t> word_count;
string word;
while (cin >> word)
    ++word_count[word];
for (const auto &w : word_count)
    cout << w.first << " occurs " << w.second << endl;
// count occurrences, but the words won't be
// in alphabetical order
```

The code is the same as for maps! if we can use maps we can also use unordered\_maps.

The difference is that if we go sequentially to the container we don't get the order (because the storing is based on the hash function, which do not preserve any order).

## Complexities of operations

You can choose the most suitable container in terms of complexity, depending on what operations you need to apply on them

| Container          | Operation        |                  |                  |
|--------------------|------------------|------------------|------------------|
|                    | Insert           | Find             | Delete           |
| list/ forward_list | $O(1)$           | $O(n)$           | $O(1)$           |
| set/map            | $O(\log(n))$     | $O(\log(n))$     | $O(\log(n))$     |
| unordered set/map  | $O(1)$ or $O(n)$ | $O(1)$ or $O(n)$ | $O(1)$ or $O(n)$ |

(Exam)

The first thing that we look for is: WHAT DO WE NEED TO OPTIMIZE?

### map vs. unordered\_map

- If the most frequent operations of your application are find, insert or delete of single elements and you access through a key:
  - Use a map if you want to optimize the worst case complexity
    - $O(\log(N))$  vs.  $O(N)$  in an unordered\_map
  - Use an unordered\_map if you want to optimize the average case complexity
    - $O(1)$  vs.  $O(\log(N))$  in a map

## References

- Lippman Chapters 10,11
- <http://www.cplusplus.com/reference/stl/>
- <http://www.learncpp.com/cpp-tutorial>

## Credits

- Bjarne Stroustrup. [www.stroustrup.com/Programming](http://www.stroustrup.com/Programming)