Contemporary C++: *Learning Modern C++ in a Modern Way*

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 5/24

Session 5 .Introduction to contiguously sequential data structures: arrays, vectors and strings

- The C-Style array
- C++ standard array
- C++ standard library: The vector class
- Vector manipulation: writing simple programs
- Range-based for loop: Scratch the surface
- More on Autos: Automatic type deduction
- C++ standard library: The string class
- The C-Style string and String literals
- String manipulation: writing simple programs
- Q&A

150 min (incl. Q & A)

Functions

- Program organization: Data structures, Functions, User-defined types
- Function is one of fundamental tools for *abstraction* in C++.

- A function is a piece of program that has a name, and that another part of the program can *call*, or cause to run.

- function declaration:
  - function name
  - return type
  - number and type of arguments

```
double sqrt(double); // just declaration
double sqrt(double x) { // decl. and def.
  // … compute the square root of x and return it;
}
void exit(int exit_code); //
```

- function definition:
  - function declaration
  - function body

Function name

Argument type and name

Return type

```
long int sqr(int x)
{
   return x * x;
}
```

Function body

Function call

- Use **`sqr`** and **`sqrt`**.

```
double sr2 = sqrt(2); // call sqrt with argument double(2)
long s2 = sqr(2); // call sqr with argument 2
double sr3 = sqrt("three"); // error: sqrt() requires an argument of type double
array<int, 4> a = { 0, 1, 2, 3 };
long s4 = sqr(a); // error: sqr() requires an argument of type int
```

- The semantics of *argument passing* are identical to the semantics of *initialization*.

- The type of the definition and all declarations for a function must specify the same type.

Formal argument

```
long int sqr(int x)
{
    return x * x;
}

int main()
{
   long X = sqr(10); // function call

}
```

Actual argument

Functions: some examples

```
/* compute a student's overall grade from
   midterm and final exam grades and homework grade */
double grade(double midterm, double final, double homework)
{
   return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}
```

```
// return the maximum value of two integers
int Max(int x, int y)
{
  return x > y ? x : y;
}
```

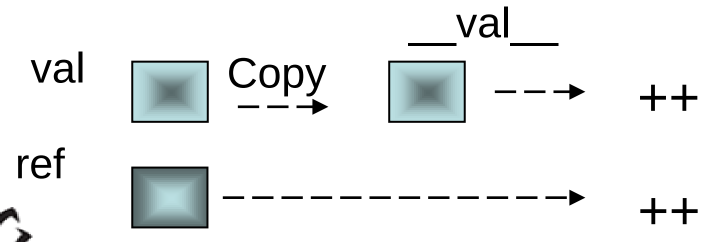
PrintSquareFunction

Prog.

Argument passing, pass by value, pass by reference

- Formal arguments vs. Actual arguments
- Initialization
- Type checking and type conversion
- Pass by value vs. Pass by reference

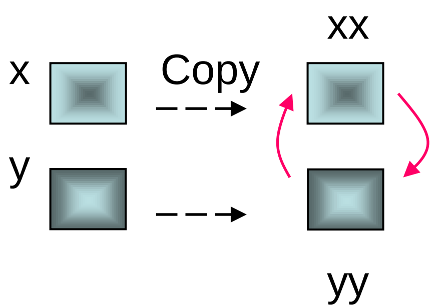
```
void f(int val, int& ref)
{
  val++;
  ref++;
}
void g()
{
  int i = 1;
  int j = 1;
  f(i,j); // i == 1, j == 2
}
```



Pass by value vs. Pass by reference: swap example

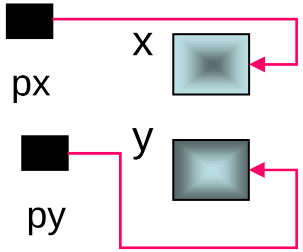
- Pass-by value

```
/* wrong */
void swap(int x, int y)
{
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
}
```



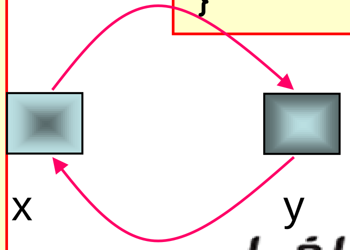
- Pass-by reference: C solution

```
/* right */
void swap(int* px, int* py)
{
   int tmp;
   tmp = *px;
   *px = *py;
   *py = tmp;
}
```



- Pass-by reference: C^{++} solution

```
/* right */
void swap(int& x, int& y)
{
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
}
```



Return type, Value return and void

- A function should return a value unless it is a void function.
- void indicates absence of information.
- There are no objects of type void.

Value return

Return type

```
int f1() { } // error: no value returned
void f2() { } // ok
int f3() { return 1; } // ok
void f4() { return 1; } // error: return value in void function
int f5() { return; } // error: return value missing
void f6() { return; } // ok
```

- void has two main applications:
  - Function: void return type
  - Generic pointer: void*
- The semantics of *function value return* are identical to the semantics of *initialization*. A return statement is considered to initialize an *unnamed variable* of the returned type.

Object vs. Variable

- Variable is an object that has a name.

Value return, some points cont.

```
int* fp() // bad, don't do this
{
  int local = 1; // automatic variable
  /* ... */
  return &local; // return pointer to local
}
```

local destroyed/deallocated here

```
int& fp() // bad, don't do this
{
  int local = 1;
  /* ... */
  return local; // return reference to local
}
```

local destroyed/deallocated here

```
int fp() // good
{
  int local = 1;
  /* ... */
  return local; // OK: return a copy of local
}
```

Inline functions

- A function can be defined to be inline.
- inline is a function specifier. other specifiers are virtual and explicit.
- The inline specifier indicates to the implementation that inline substitution of the function body at the point of call is *to be preferred* to the usual function call mechanism.

```
// return the maximum value of two integers
int Max(int x, int y)
{
  return x > y ? x : y;
}

void f()
{
  int m = Max(3, 5); // function call
}
```

Function call overhead

```
// return the maximum value of two integers
inline int Max(int x, int y)
{
  return x > y ? x : y;
}

void f()
{
  int m = Max(3, 5); // expand inline
}
```

Inline functions cont.

- The inline specifier is a *hint* to the compiler. Compilers are free to ignore inline directives and to make their own decisions about which functions to inline.

- An inline specifier does not affect the semantics of a function.

- Inline function:
  - no function call overhead
  - larger program
  - faster program

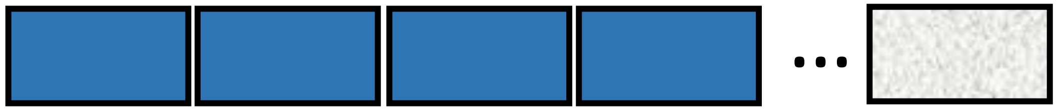
- Non-Inline function:
  - function call overhead
  - smaller program
  - slower program

Q Are inline functions guaranteed to make your performance better?

A No. Beware that overuse of inline functions can cause *code bloat*, which can in turn have a negative performance impact in paging environments.

Contiguously sequential data structures

- Data structure
- Homogenous and Contiguous data structures
- Sequential data structures
- The most fundamental collection of data is a contiguously allocated sequence of elements of the same type, called an *array*.



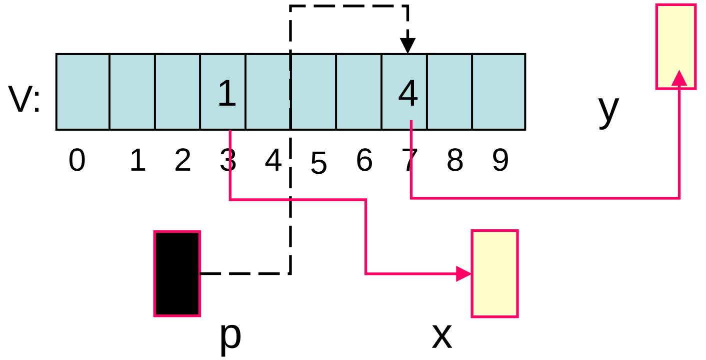
Arrays

- For a type T, T[size] is the type “array of *size* elements of Type T.” The elements are indexed from 0 to size -1.

  - [ ] is subscription operator.

  - A C++ array is simply a sequence of memory locations.

```
int v[10]; // an array of 10 ints
v[3] = 1; // assign 1 to v[3]
int x = v[3]; // read from v[3]
int* p; // p is a pointer to an int
p = &v[7]; // assign the address of v[7] to p
*p = 4; // write to v[7] through p
int y = *p; // read from v[7] through p
```



- The number of array elements called array bound. Array bound is a constant and must be known at compile-time. If you need variable bound, use vector:

```
void f(int i)
{
  int v1[i]; // error: array size not a constant expression
  vector<int> v2(i); // ok
}
```

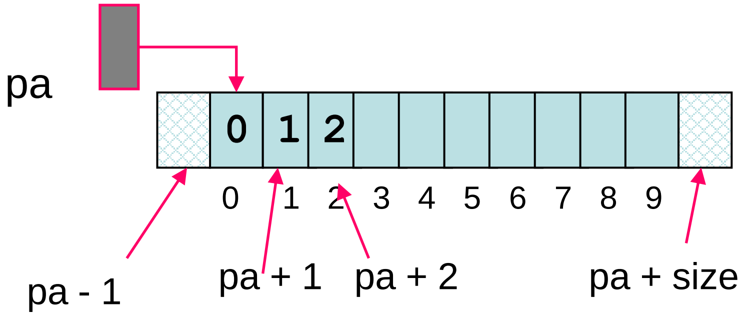
- Array initializer:

```
char vowel[] = { ‘a’, ‘e’, ‘i’, ‘o’, ‘u’ };
int a[5] = { 1, 2, 3, 4, 5 };
bool b[3] = { true, false }; // b[2] = false
```

Pointers, arrays and pointer arithmetic

- Pointers and arrays are closely related.

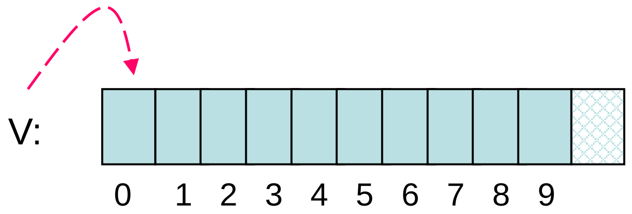
```
const int size = 10;
*pa == 0;
*(pa + 1) == 1;
*(pa + 2) == 2;
*(pa + size) == one beyond the last element
*(pa - 1) == undefined
```



- Taking a pointer to the element one beyond the end of an array is guaranteed to work.
- The name of an array is a pointer to the first element:

```
int v[10];          int *v;
```

```
int* p = &v[0];
int* q = v
```



++ --
=*p *p=
+ - += -=
== !=
< <= >
>=
-> [ ]

Pointer arithmetic operations

Arrays cont.

- Arrays don’t know their sizes and bounds.

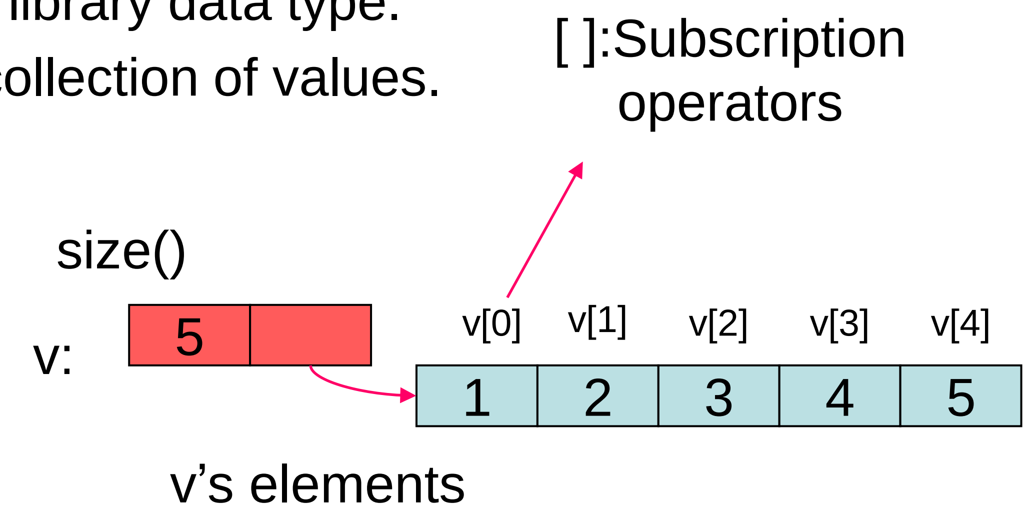
```
void fp(char v[], unsigned int size)
{
   for (int i = 0; i < size; i++)
     // use v[i]
}
```

- Write a function that finds an integer value in the array. The function should return a pointer to the found element. If find failed, it should return pointer to one beyond the last element.

```
int* find(int v[], int vsize, int val) // find val in v
{
  for (int i = 0; i < vsize; i++)
    if (v[i] == val) return &v[i]; // if val is found return pointer to element
  return &v[vsize]; // if not found return pointer to one-beyond-the-end of v
}
```

Vectors

- Vector is the most useful standard library data type.
- Vector is a container that holds a collection of values.
- All of the values in an individual vector are the same type, but different vectors can hold objects of different types.
- a vector<T> holds an sequence of values of type T.



- examples:

```
vector<int>, vector<double>, vector<string>, vector<vector<int> >, ...
```

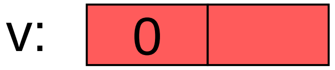
- The vector type is defined using language feature called template classes.

vector → Fundamental types + User-defined types

- vector is defined in namespace std and in <vector> header file.

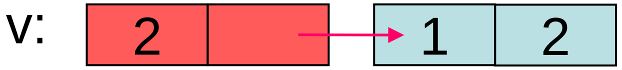
Vector cont.

- a Vector can grows as needed:
  - `vector<int> v; // v is a vector of integer. start off empty`

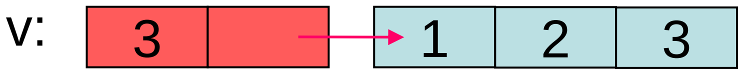


  - `v.push_back(1); // add an element with the value 1 at the end ("the back")`

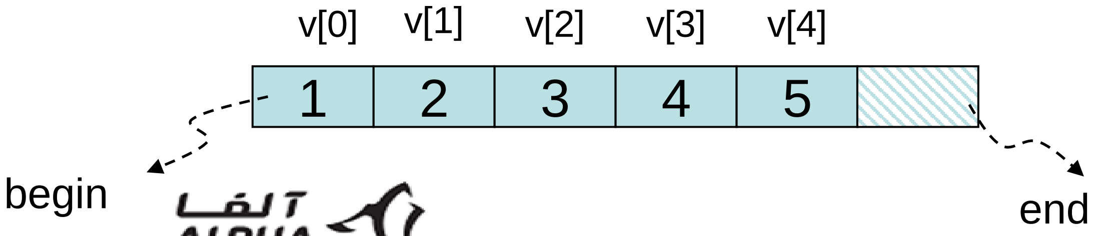
  - `v.push_back(2); // add an element with the value 2 at the end ("the back")`



  - `v.push_back(3); // add an element with the value 3 at the end ("the back")`



- **`push_back`** pushes its argument onto the back of a vector. As a side effect, it increases the size of the vector by one.



Vector cont. and algorithms

vector:

- push_back()
- size()
- subscription operator: []
- empty()
- begin()
- end()

public interface

algorithms:

- sort()
- count()
- find()

…

generic algorithms

• Read some numbers as daily temperatures and compute average and median of them.

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

using std::cin;    using std::cout; using std::endl;
using std::vector; using std::sort;

int main()
{
  vector<double> temps; // temperatures in Fahrenheit, e.g. 64.6
// to be continued on next page
```

Simple computations, simple programs

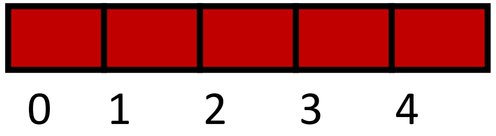
```
// continued from last page
  double temp;
  while (cin >> temp)
        temps.push_back(temp); // put into vector
  double sum = 0;
  for (int i = 0; i < temps.size(); ++i)
    sum += temps[i];
  cout << "Average temperature: " << sum / temps.size() << endl;
  sort(temps.begin(),temps.end()); // sort "from the beginning to the end"

  cout<< "Median temperature: " << temps[temps.size() / 2] << endl;
  return 0;
}
```

Simple computations, simple programs

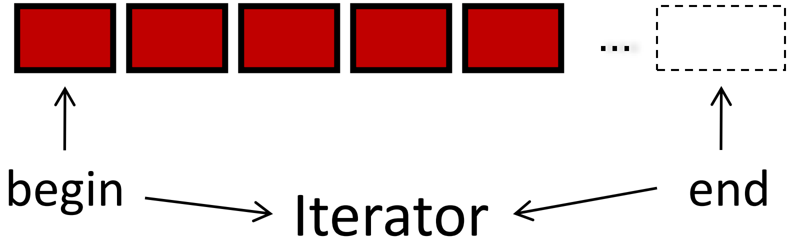
Array vs. vector

C/C++ array



- An array doesn't know its own size
- The size of array should be known at compile-time
- There is no grow or shrink for arrays.
- The name of an array converts to a pointer to its first element.

C++ vector



But

- A vector knows its own size: size()

But

- The size of vector can change at run-time

But

- Vector can grow or shrink dynamically.

But

- The name of vector doesn't decay to the first element of container.

Array vs. vector 2.

- Array uses stack or automatic storage

But

- Vector uses heap or dynamic storage

- Array uses pointer arithmetic for traversal.

But

- Vector uses random-access iterator for traversal.

ALPHA آلفا

Array container

C-Style Array

Pointer arithmetic

Size unaware

Implicit decay name to pointer

Compile-time static size

automatic storage

No dynamic resize

Array container

C-Style Array

Pointer arithmetic

Size unaware

Implicit decay name to pointer

Compile-time static size

automatic storage

No dynamic resize

Vector

Size aware

Random-access iterator

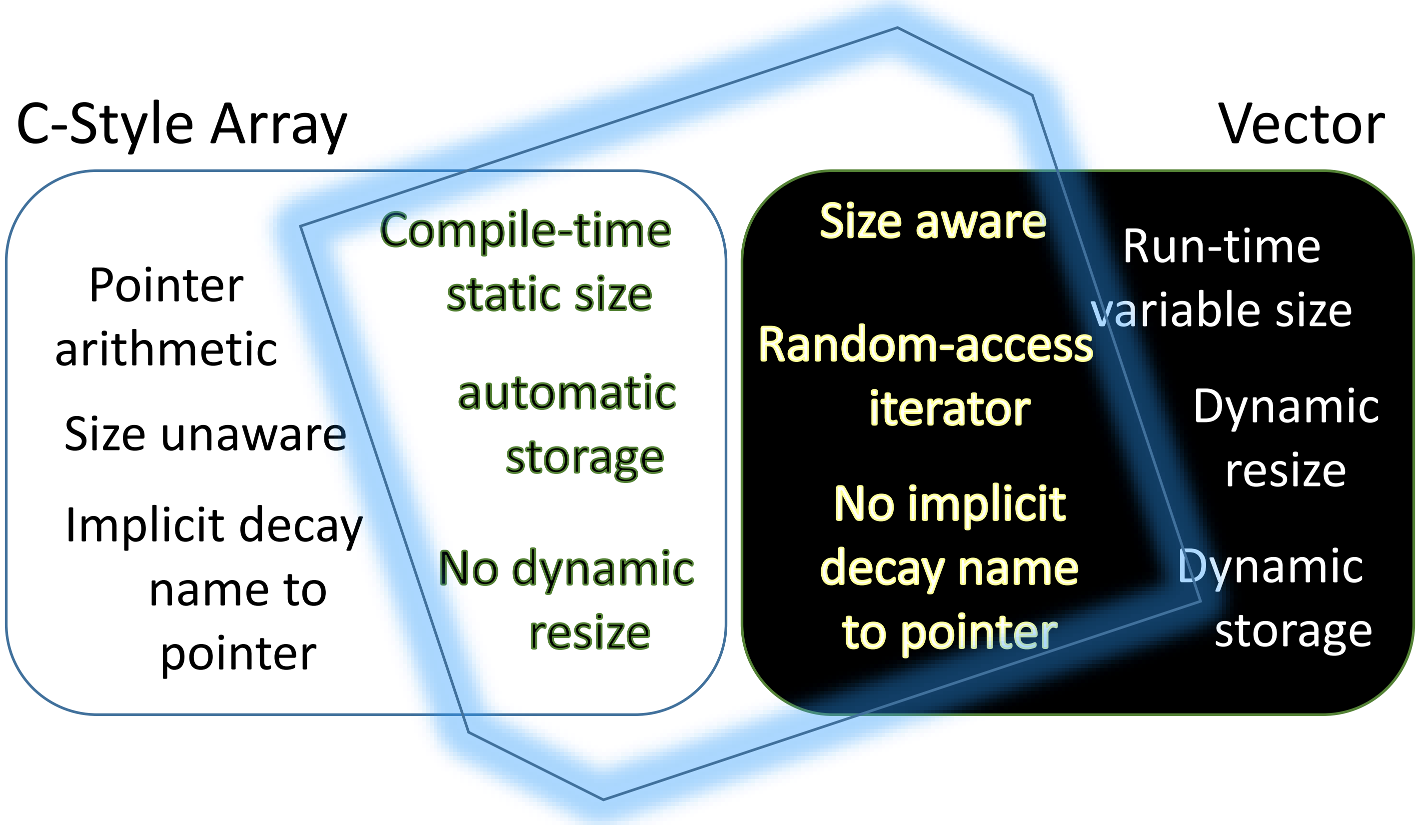
No implicit decay name to pointer

Run-time variable size

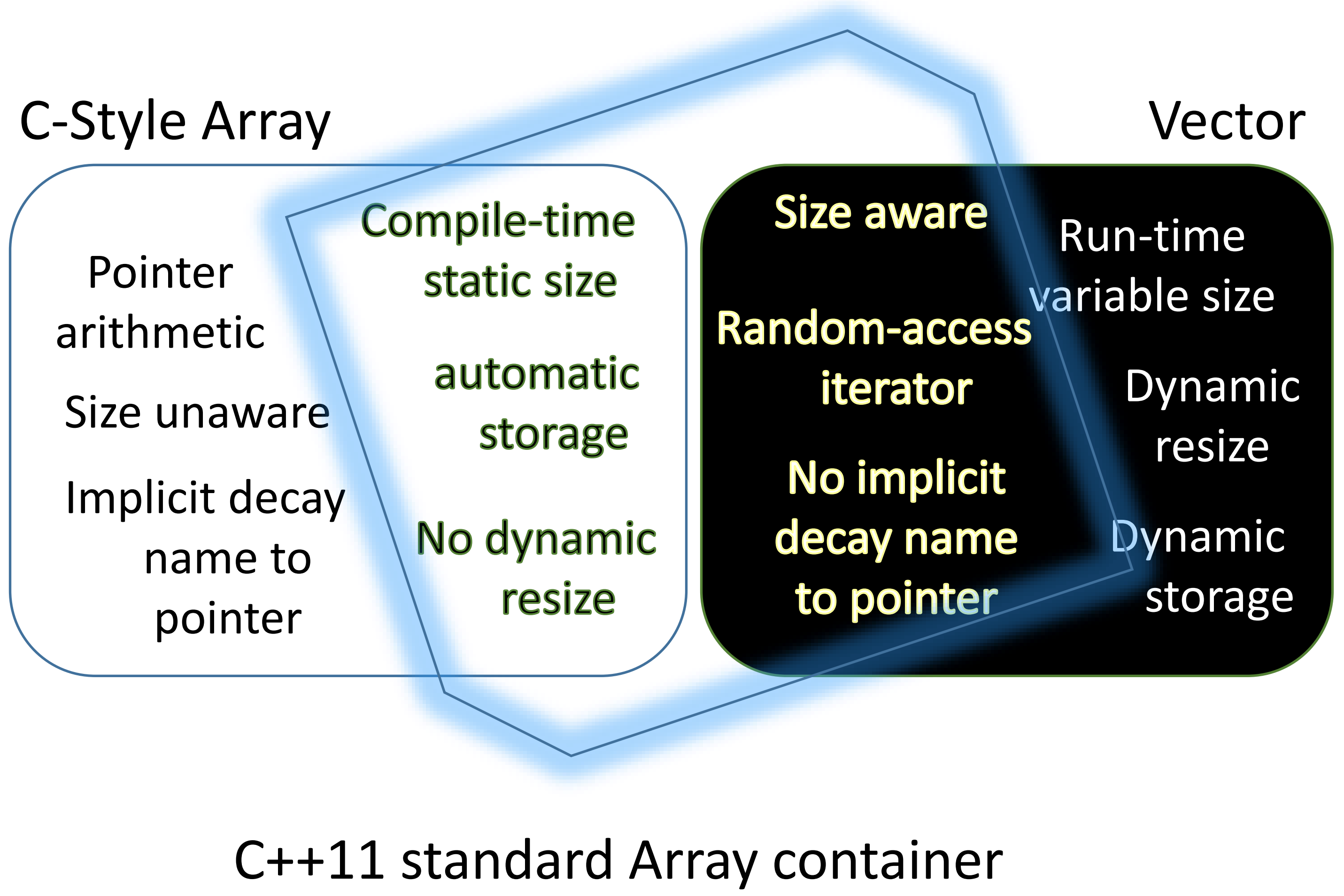
Dynamic resize

Dynamic storage

Array container



Array container

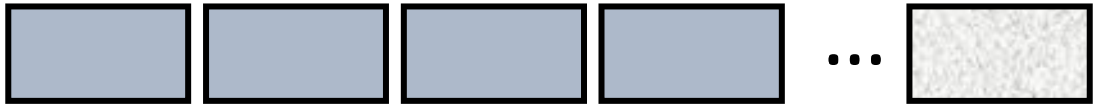


Array

- An array, is a fixed-size sequence of elements of a given type where the number of elements is specified at compile-time.

- Like a built-in array, an array is simply a sequence of elements, with no handle:

elements:



- This implies that a local array does not use any free store.

Compile-time array size

- <array>

```
template <class T, size_t N> struct array;
array<int, 3> a;
```

- The standard array is suitable for embedded systems programming and similar constrained, performance-critical, or safety critical tasks.
- (Hard) Real-time systems, Airplanes, …

```
array<int, 3> a;
array<string, 3> language = { “C”, “C++”, “Python” };
array<double> d; // error: size not specified
```

Array Accumulate

Prog.

Range-based for loop

- Conventional general *for* statement: You can do almost anything.

for (initialization-statement condition $_{opt}$; expression $_{opt}$) statement

```
int a[10];
for (int i = 0; i <=10; ++i) {
  a[i] = 0;
}
```

- off-by one error
- Common idiom in C++ programming: Iterator through a container from begin to end
- Rather verbose

1.

```
void print(list<int>& L)
{
   for (list<int>::const_iterator cit = L.begin(); cit != L.end(); ++cit)
        cout << *cit << '\n';
}
```

- Looks like difficult

2.

```
void print(list<int>& L)
{
  ostream_iterator<int> os(cout, "\n");
  copy(L.begin(), L.end(), os);
}
```

Range-based for loop

- C++11→ using auto
- cbegin() and cend()

```
void print(list<int>& L)
{
   for (auto cit = L.cbegin(); cit != L.cend(); ++cit)
         cout << *cit << '\n';
}
```

- A range for statement allows you to iterate through a "range”.
- Range: arrays, containers, initializer lists, and all data structures with the STL sense.
- C++11 →One step further: range-based for loop

1.

```
void print(list<int>& L)
{
   for(int x : L) cout << x << '\n';
}
```

2.

```
void print(list<int>& L)
{
   for(auto x : L) cout << x << '\n';
}
```

Expands to

Iterate through the sequence using *begin* and *end* to find the sequence's bounds

Compiler converts

Compiler converts to

```
for(auto p = L.begin(); p!= L.end(); ++p)
cout << *p << '\n';
```

Range-based for loop: more examples

- Arrays
  - No off-by-one error

```
int a[10];
int i = 0;
for (auto x : a) x = i++;
for (const auto x : a) cout << x << '\n';
```

- maps
  - C++98

```
// initialize map using initializers list: we'll discuss it too soon.
map<string, int> inventory = { {"Nail", 3000}, { "Hammer", 10}, {"Saw", 5} };
for (auto p = inventory.begin(); p != inventory.end(); ++p) p->second++;
for (auto p = inventory.begin(); p != inventory.end(); ++p)
  cout << p->first << '\t' << p->second << '\n';
```

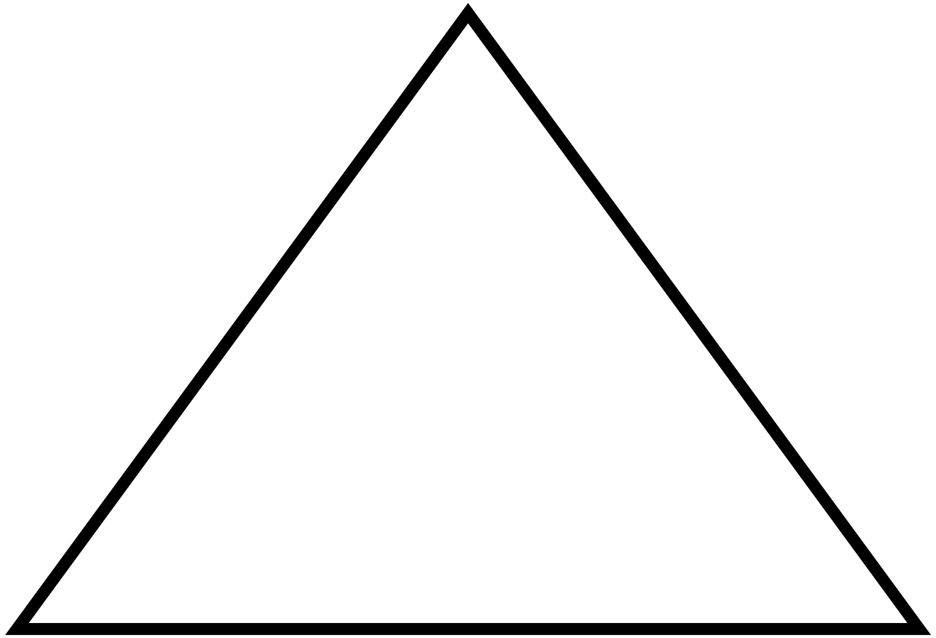
  - C++11

```
for (auto& item : inventory) item.second++;
for (const auto& item : inventory)
cout << item.first << '\t' << item.second << '\n';
```

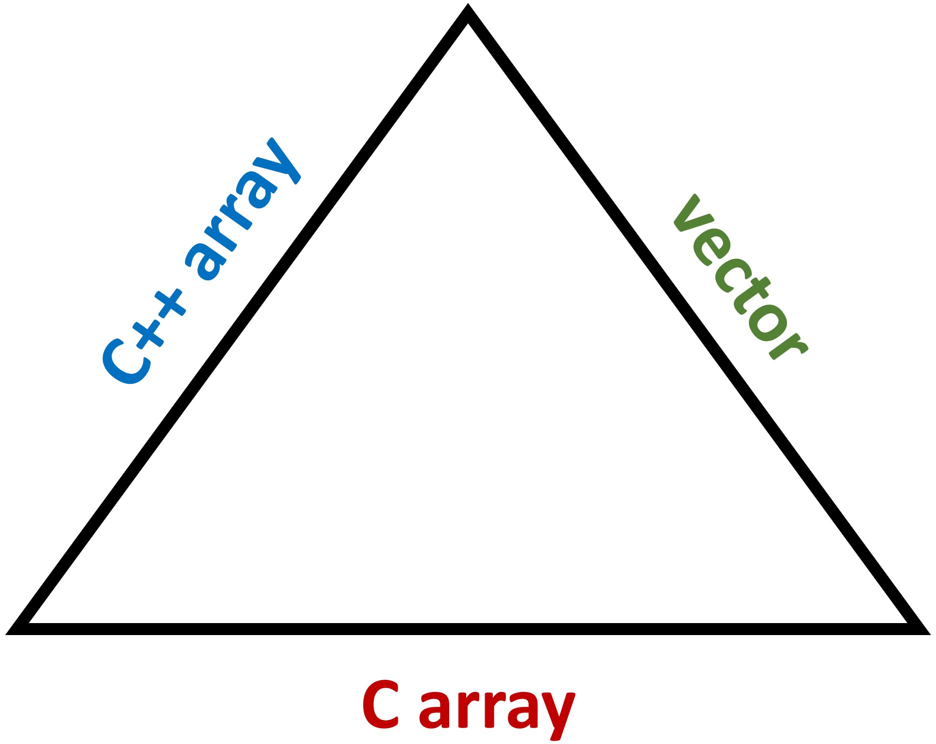
- C++14

Built-in array vs. Standard array vs. Vector

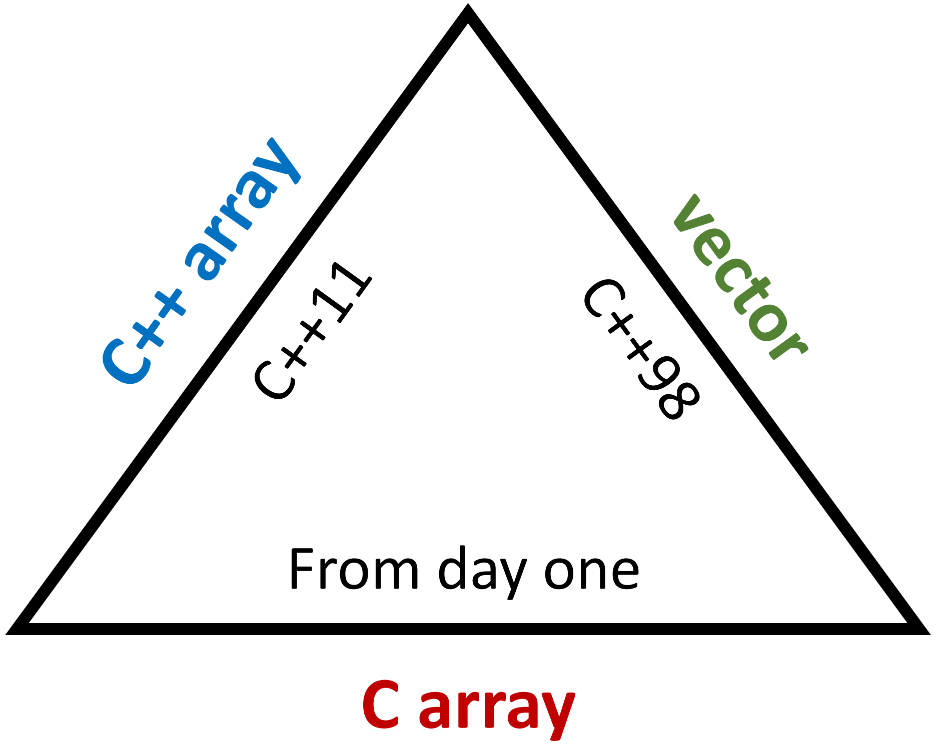
Built-in array vs. Standard array vs. Vector



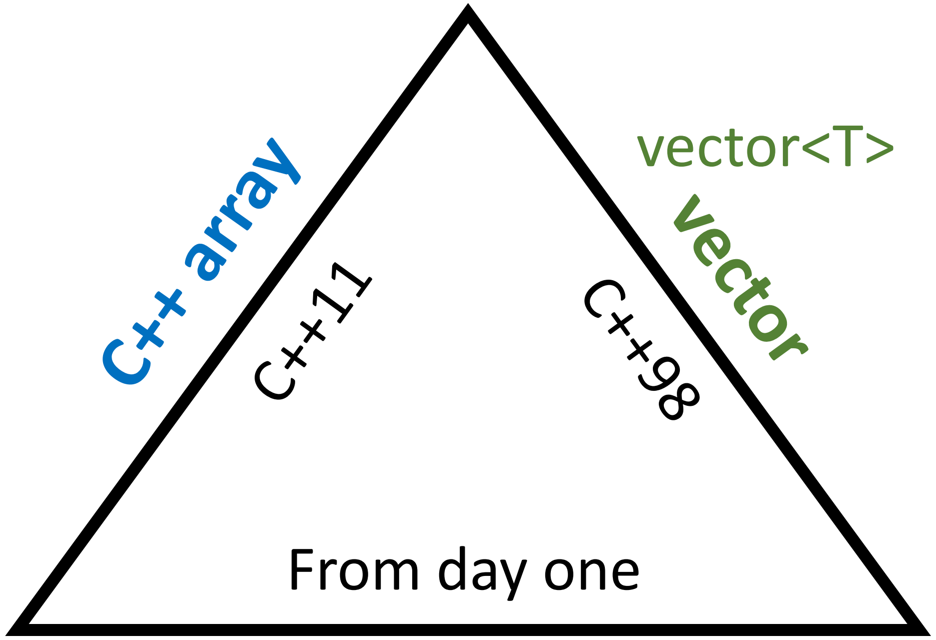
Built-in array vs. Standard array vs. Vector



Built-in array vs. Standard array vs. Vector

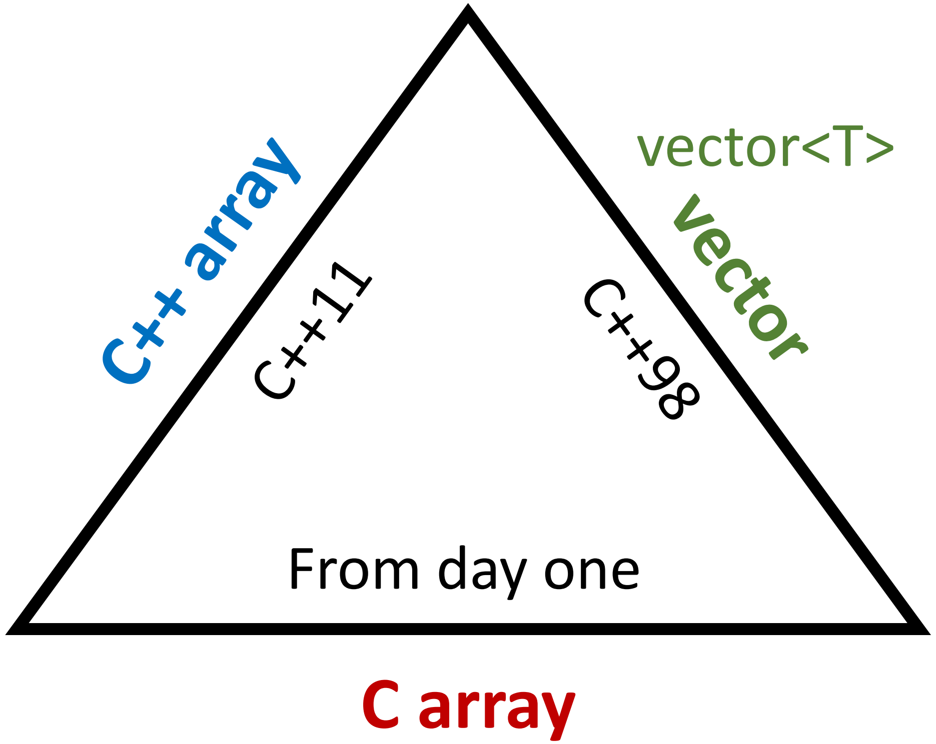


Built-in array vs. Standard array vs. Vector



Built-in array vs. Standard array vs. Vector

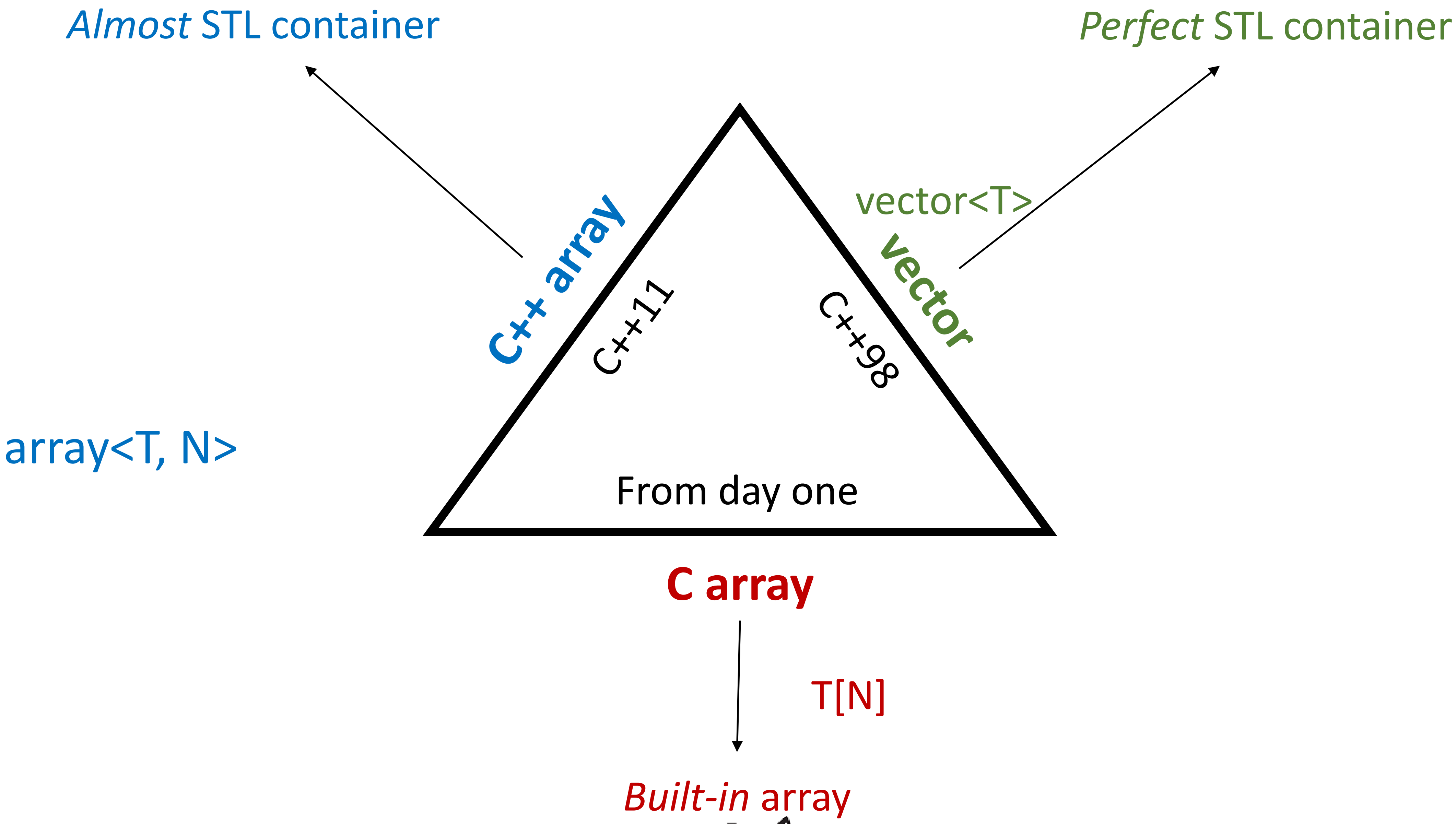
vector<T>



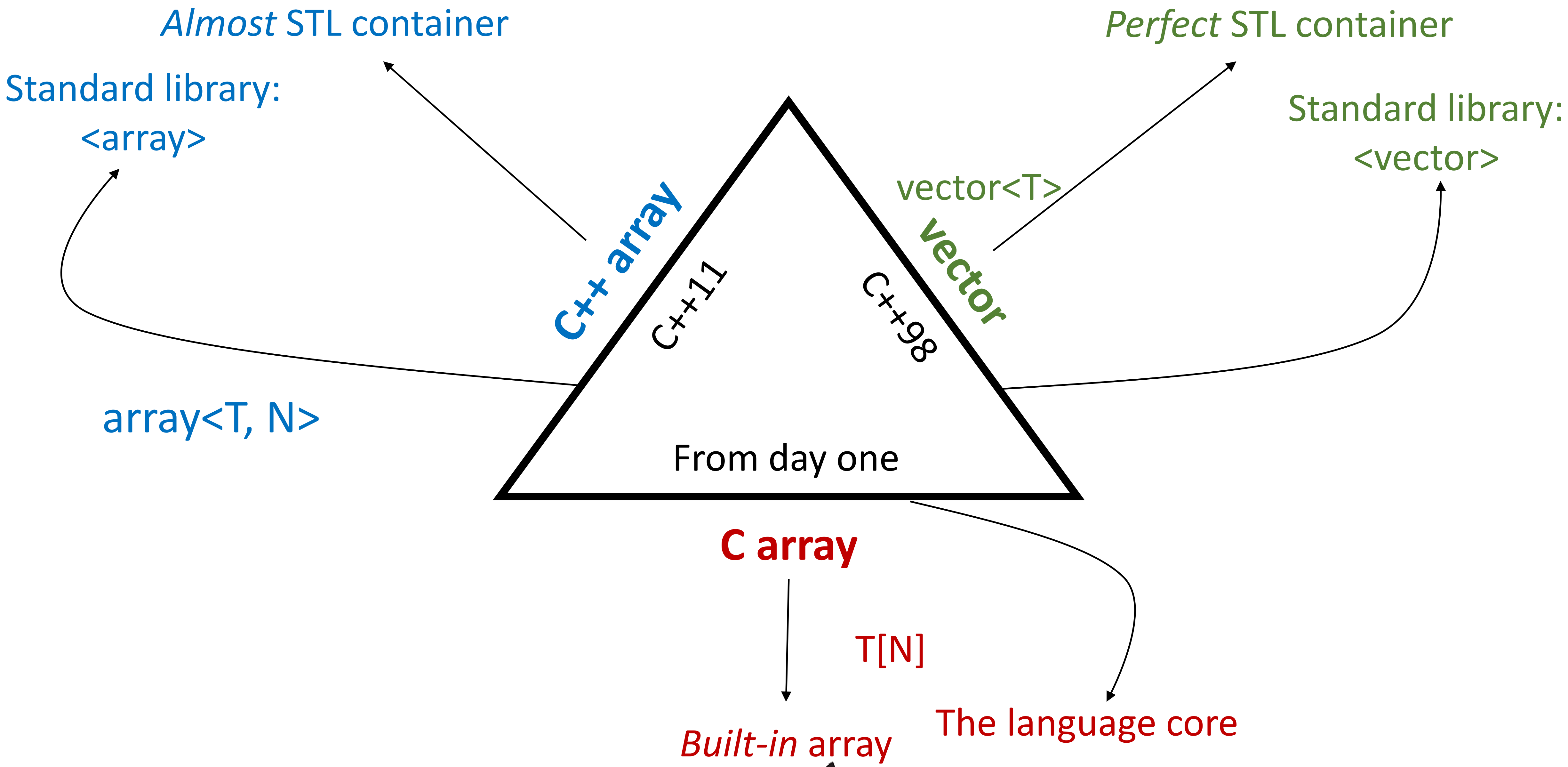
array<T, N>

T[N]

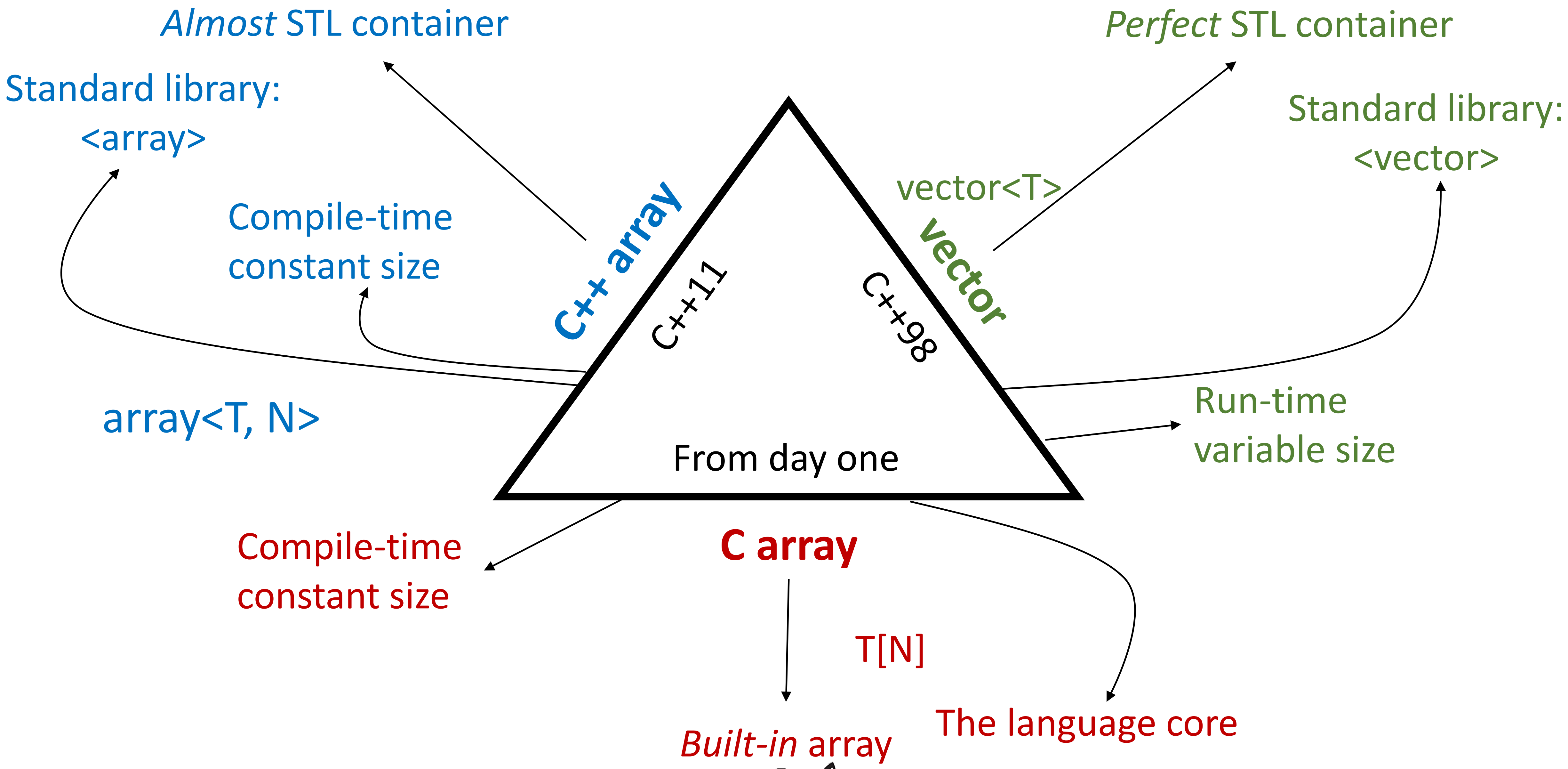
Built-in array vs. Standard array vs. Vector



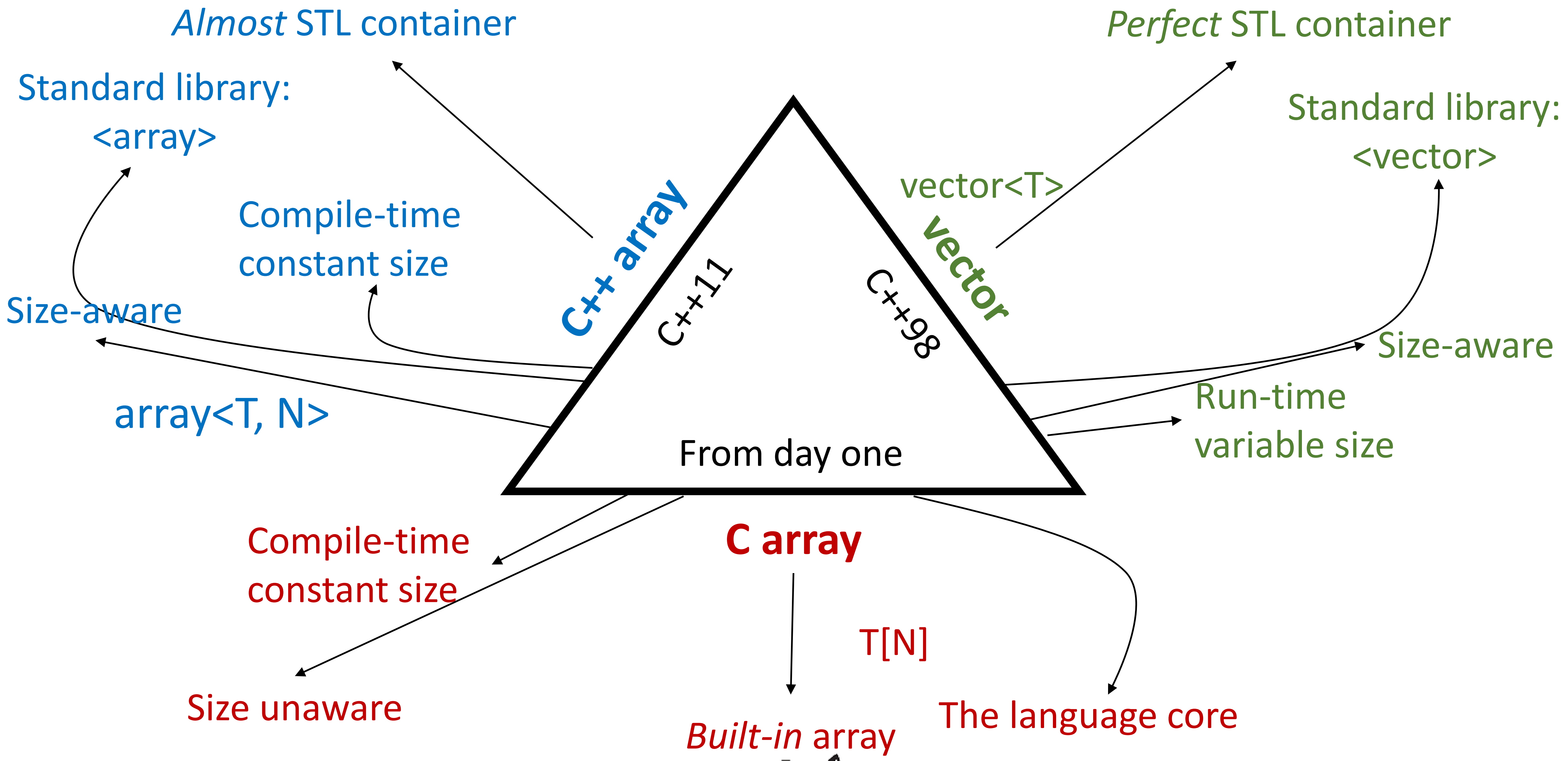
Built-in array vs. Standard array vs. Vector



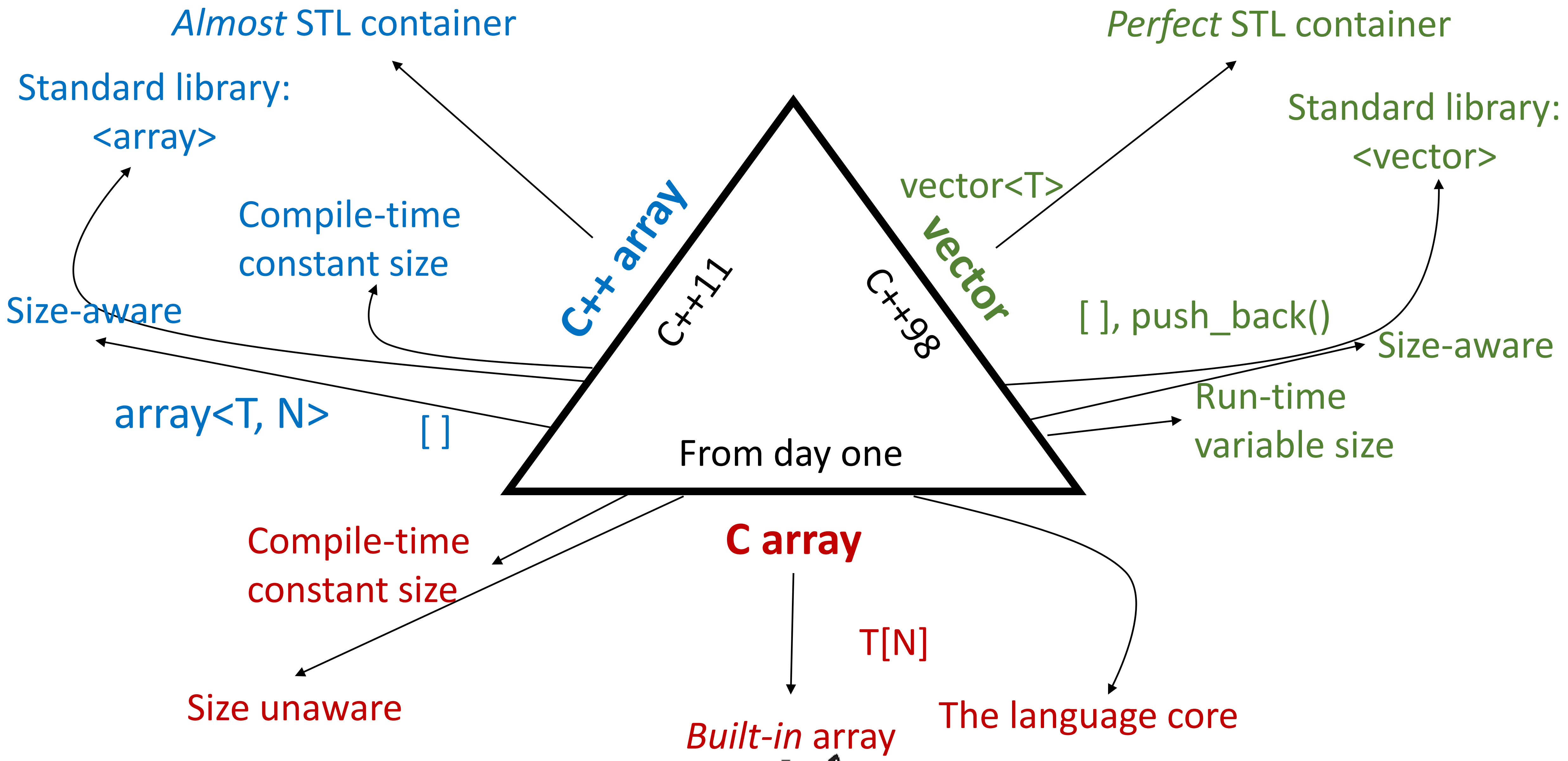
Built-in array vs. Standard array vs. Vector



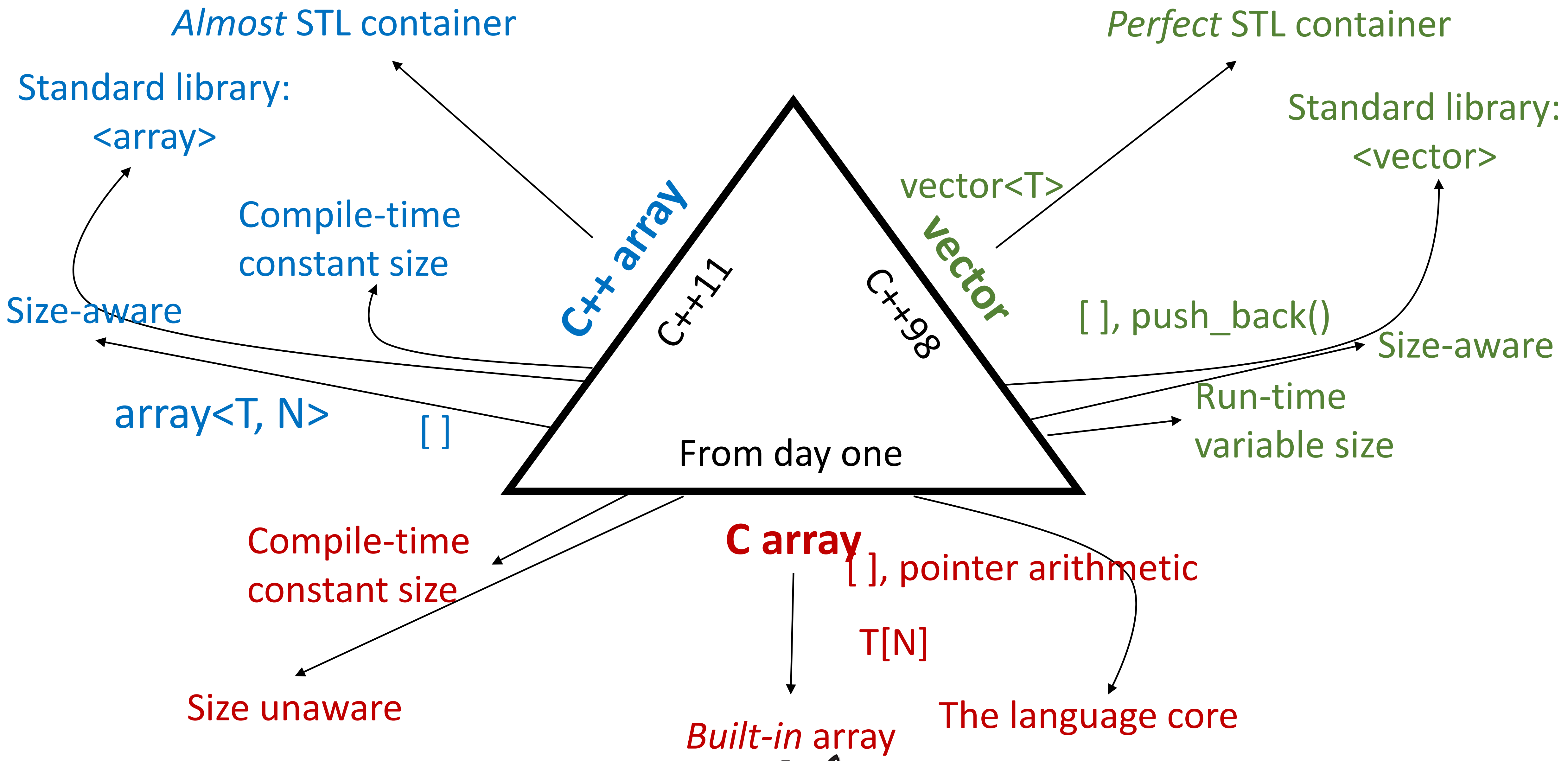
Built-in array vs. Standard array vs. Vector



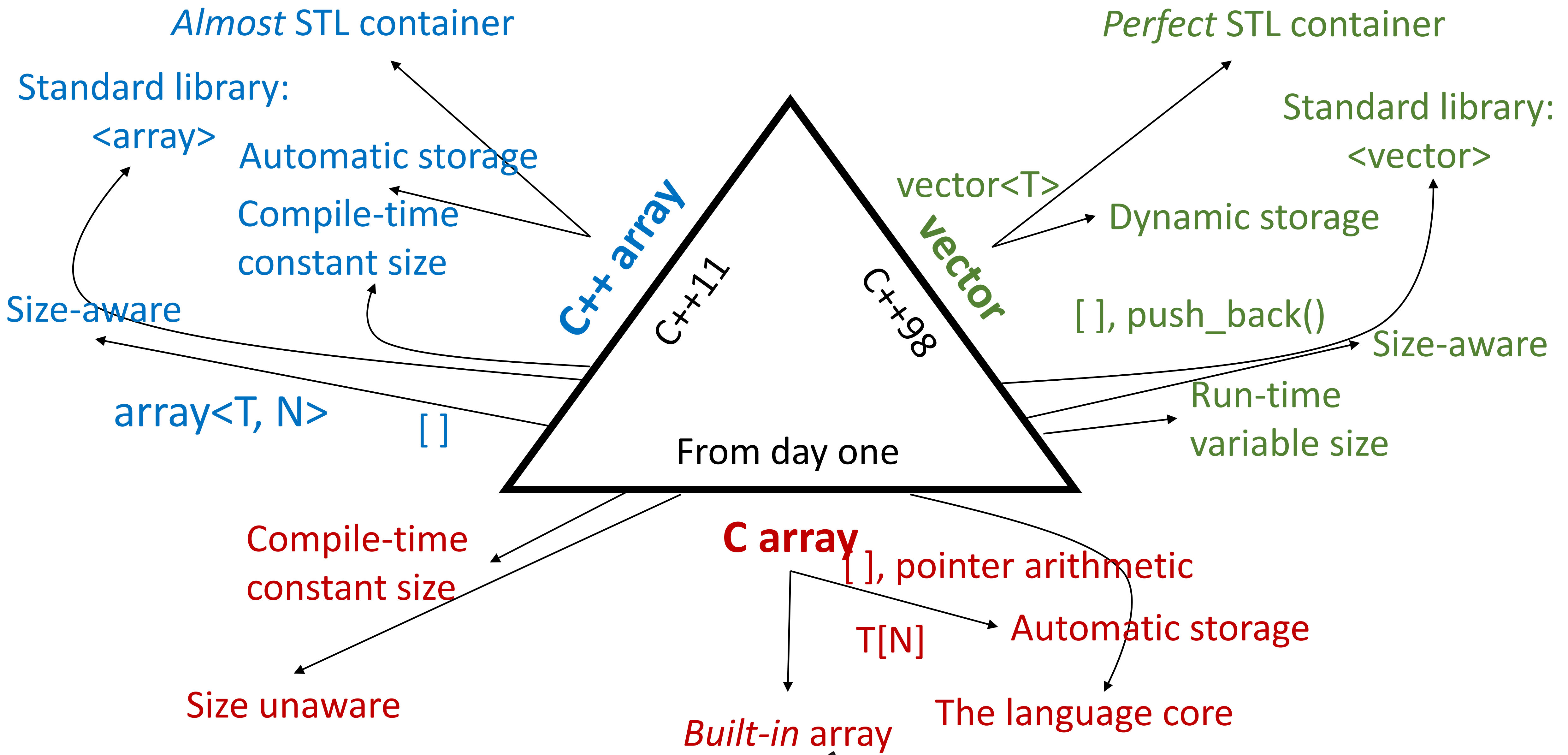
Built-in array vs. Standard array vs. Vector



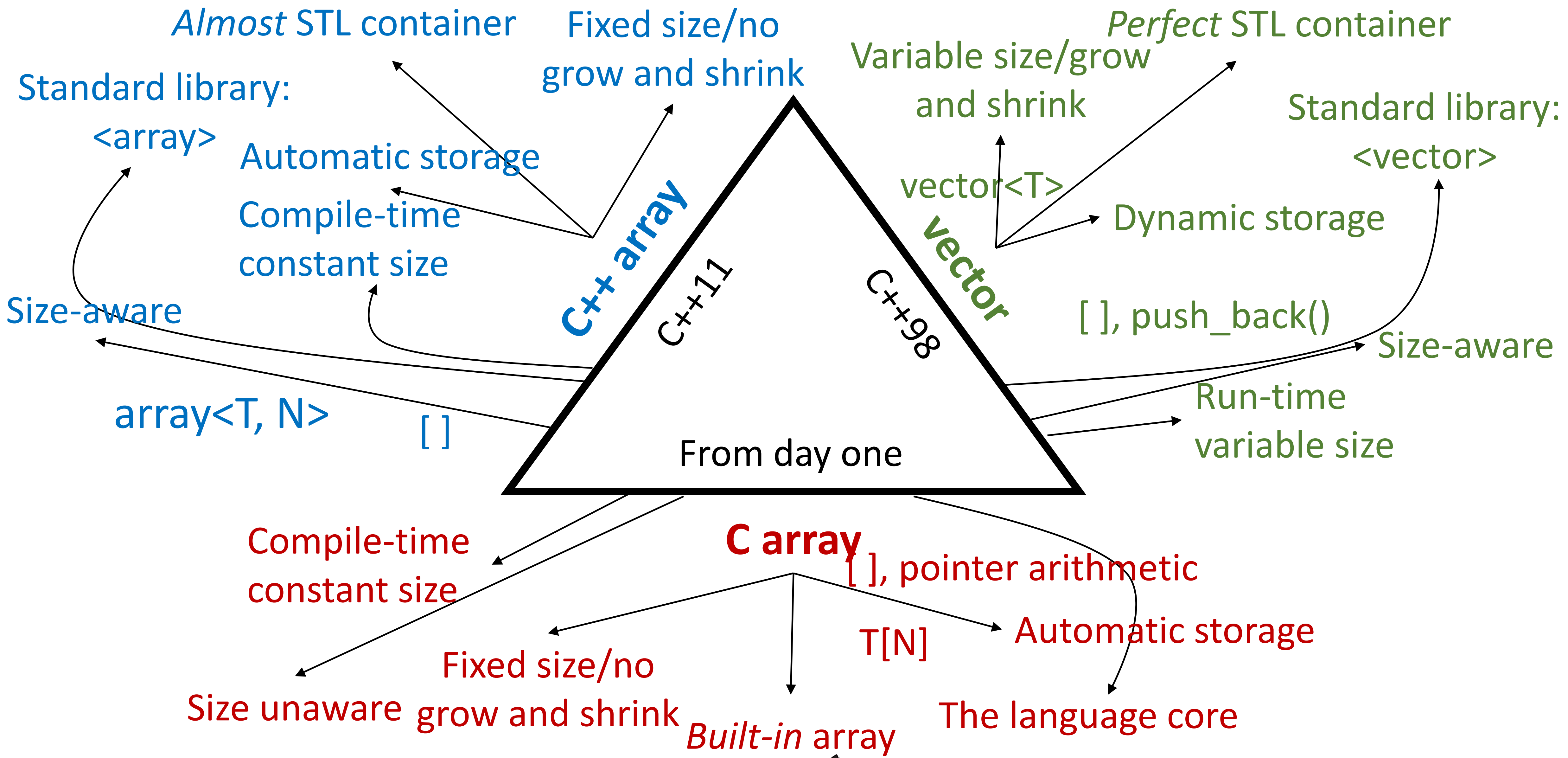
Built-in array vs. Standard array vs. Vector



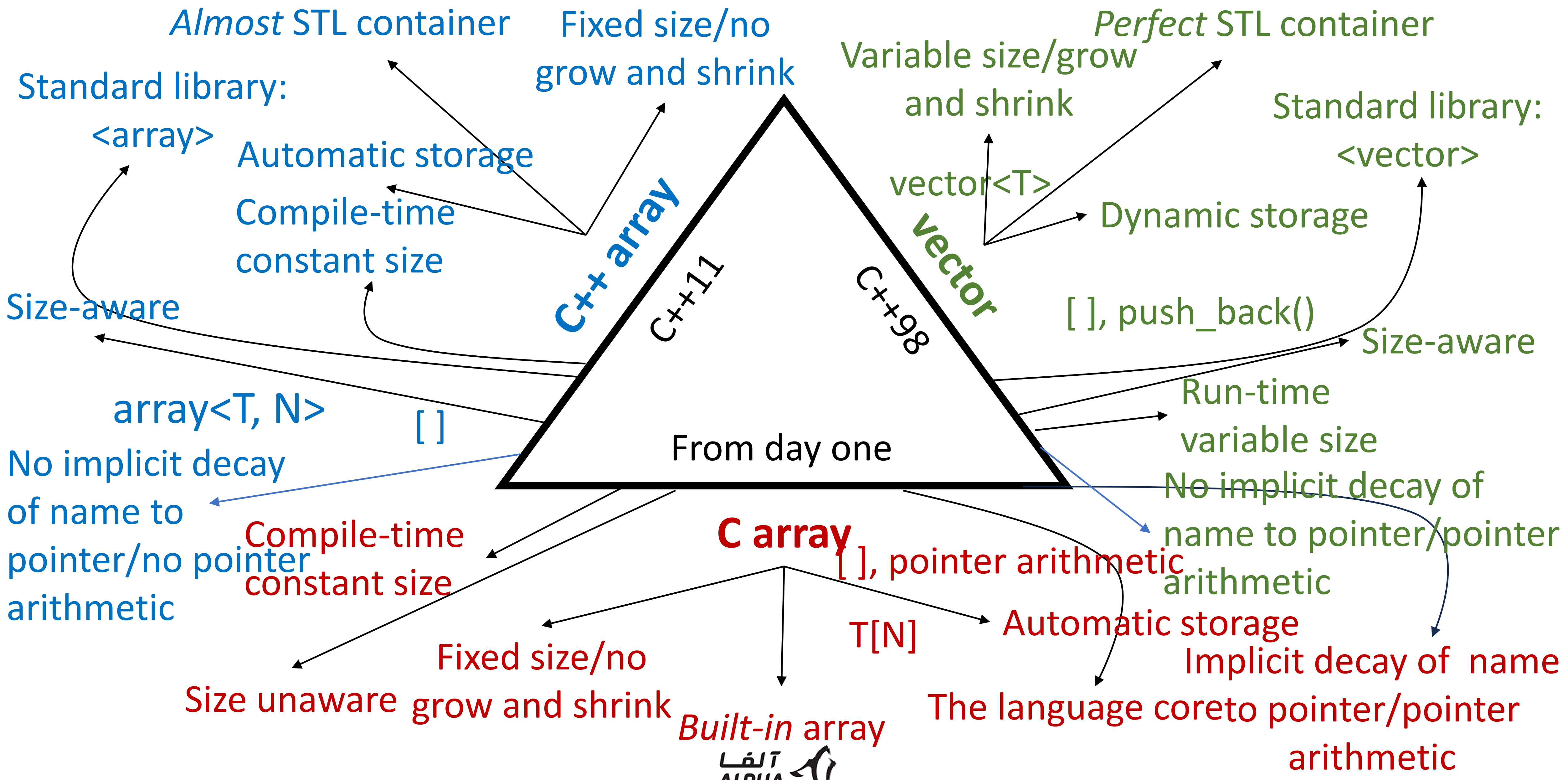
Built-in array vs. Standard array vs. Vector



Built-in array vs. Standard array vs. Vector



Built-in array vs. Standard array vs. Vector



String literals

- A *string literal* is a character sequence enclosed within double quotes:

"t h i s i s a s t r i n g"

t	h	i	s		i	s		a		s	t	r	i	n	g	\0

```
char v[17];        int *v = “this is a string”;
```

- The type of a string literal is “array of the appropriate number of const characters.”

- A string literal can be assigned to a char*.

```
char* p = “Hello”;
“” // empty string: const char[1]
```

```
wchar_t* p = L“Hello”;
L“” // empty string: const wchar_t[1]
```

- Escape characters:

Name	ASCII name	C^{++} name
newline	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
alert	BEL	\a
question mark	?	\?
single quote	‘	\’
double quote	“	\”

Writing simple functions

```
/* atoi: convert s to integer */
int atoi(const char s[])
{
  int i, n;
  n = 0;
  for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
    n = 10 * n + (s[i] - '0');

  return n;
}
```

```
/* strlen: return length of string s */
int strlen(const char *s) // buggy and naïve
{
  int n;
  for (n = 0; *s != '\0', s++)
    n++;

  return n;
}
```

Writing simple functions cont.

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, const char *t)
{
  int i;
  i = 0;
  while ((s[i] = t[i]) != '\0')
  i++;
}
```

```
/* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t) // average
{
  int i;
  i = 0;
  while ((*s = *t) != '\0') {
    s++;
    t++;
  }
}
```

آلفا
ALPHA

C-style array of chars vs. standard library string : comparison

- char string is defined in core language, but string is defined in standard library.
- The size of char string can be determined using null character/strlen() function, but the size of string can be determined using size() member function.
- char string is not generic, but standard string are generic.
- In array of char, we may be need to pointer arithmetic but in string we don’t.
- In array of char, we may be need to memory management but in string we don’t.

☞

Use string rather than zero-terminated arrays of char.

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant

Q & A