Declaration of Statically-Allocated Arrays

In C++, an array is simply a fixed-sized aggregation of a list of cells, each of which can hold a single value (objects).

The number of cells in an array is called its dimension.

The number of values that are actually stored in an array is called its usage.

```cpp
const int BUFFERSIZE = 256;
const int DICESUMS   = 11;
char Buffer[BUFFERSIZE];     // constant integer dimension
int DiceFreq[DICESUMS + 1]; // constant integer expression, used as dimension
int numItems = 10000;       // integer variable
string Inventory[numItems];  // NOT valid - numItems is not a constant
```

The dimension MUST be a constant value (known at compile-time).

The dimension and usage are separate values, with no association as far as the language is concerned with the array itself.

Limitations

There is no way to alter the dimension of an array once it is declared.

Access to individual cells uses the same syntax as Java; however, there is no run-time check to be sure that the specified index is actually valid.

There is no automatic aggregate operations for arrays in C++.

- `operator=( )` does not copy the contents one array into another
- arrays cannot be passed by value to a function
- `operator==( )` is not supported for arrays

When an array is passed to a function, its dimension and/or usage must generally be passed as well.
Out-of-Bounds Array Indices

What happens when a statement uses an array index that is out of bounds?

First, there is no automatic checking of array index values at run-time (some languages do provide for this). Consider the C++ code:

```
int A[7];
```


```
```

Memory

```
???????? 42
```

Clearly this is undesirable. What actually happens as a result depends upon what this location is being used for…

Memory Access Errors

Consider the possibilities. The memory location \( A[7] \) may:

- store a variable declared in your program
- store an instruction that is part of your program (unlikely on modern machines)
- not be allocated for the use of your program

In the first case, the error shown on the previous slide would cause the value of that variable to be altered. Since there is no statement that directly assigns a value to that variable, this effect seems very mysterious when debugging.

In the second case, if the altered instruction is ever executed it will have been replaced by a nonsense instruction code. This will (if you are lucky) result in the system killing your program for attempting to execute an illegal instruction.

In the third case, the result depends on the operating system you are using. Some operating systems, such as Windows 95/98/Me do not carefully monitor memory accesses and so your program may corrupt a value that actually belongs to another program (or even the operating system itself). Other operating systems, such as Windows NT/2000/XP or UNIX, will detect that a memory access violation has been attempted and suspend or kill your program.
Moral

Intelligent, disciplined programmers make efficient, effective use of arrays in C++.

Unintelligent and careless programmers find C++ arrays to be an effective means of producing unstable programs.

C++ vector Type

The C++ vector mimics the behavior of a dynamically allocated array and also supports automatic resizing at runtime.

vector declarations:

```cpp
vector<int> iVector;
vector<int> jVector(100);
cin >> Size;
vector<int> kVector(Size);
```

vector element access:

```cpp
jVector[23] = 71;
int temp = jVector[41];
cout << jVector.at(23) << endl;
int jFront = jVector.front();
int jBack = jVector.back();
```

vector reporters:

```cpp
cout << jVector.size();
cout << jVector.capacity();
cout << jVector.max_capacity();
if ( jVector.empty() ) // . . .
```
**vector Constructors**

The `vector` template provides several constructors:

```cpp
class vector {
public:
  // constructor for empty vector
  vector();

  // constructor for n copies of value
  vector(size_type n, const T& value = T());

  // constructor for n copies of default for T
  vector(size_type n, const Allocator& = Allocator());

private:
  // implementation details
  size_type _size;
};
```

The `vector` template also provides a suitable deep copy constructor and assignment overload.

---

**vector Example**

```cpp
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

int main() {
  int MaxCount = 100;
  vector<int> iVector(MaxCount);
  for (int Count = 0; Count < MaxCount; Count++) {
    iVector[Count] = Count;
  }
}
```

Specify initial vector size.

Can access like an array...

Warning: the capacity of this `vector` will NOT automatically increase as needed if access is performed using the `[]` operator. See the discussion of member functions `insert()` and `push_back()`.
vector Indexing

In the simplest case, a vector object may be used as a simple dynamically allocated array:

```cpp
int MaxCount = 100;
vector<int> iVector(MaxCount);
for (int Count = 0; Count < 2*MaxCount; Count++) {
    cout << iVector[Count];
}
```

However, the usage above provides neither runtime checking of the vector index bounds, or dynamic growth. If the loop counter exceeded the capacity of the vector object, an access violation would occur.

```cpp
int MaxCount = 100;
vector<int> iVector(MaxCount);
for (int Count = 0; Count < 2*MaxCount; Count++) {
    cout << iVector.at(Count);
}
```

Use of the at() member function causes an out_of_bounds exception in the same situation.

---

STL Iterators

iterator an object that keeps track of a location within an associated STL container object, providing support for traversal (increment/decrement), dereferencing, and container bounds detection. (See Stroustrup 3.8.1 – 3.8.4)

An iterator is declared with an association to a particular container type and its implementation is both dependent upon that type and of no particular importance to the user.

Iterators are fundamental to many of the STL algorithms and are a necessary tool for making good use of the STL container library.

Each STL container type includes member functions begin() and end() which effectively specify iterator values for the first element and the "first-past-last" element.
## Vector Iterator Example

```cpp
string DigitString = "45658228458720501289";
vector<int> BigInt;
for (int i = 0; i < DigitString.length(); i++) {
    BigInt.push_back(DigitString.at(i) - '0');
}

vector<int> Copy;
for (vector<int>::iterator It = BigInt.begin();
    It != BigInt.end(); It++) {
    Copy.push_back(*It);
}
```

Observe that with the `push_back()` member, `BigInt` will grow to hold as many digits as necessary.

This could also be written using a `for` loop, or by using the assignment operator.

## Iterator Operations

Each STL iterator provides certain facilities via a standard interface:

```cpp
string DigitString = "45658228458720501289";
vector<int> BigInt;
for (int i = 0; i < DigitString.length(); i++) {
    BigInt.push_back(DigitString.at(i) - '0');
}

vector<int>::iterator It;
It = BigInt.begin();
int FirstElement = *It;
It++;
It = BigInt.end();
It--;
int LastElement = *It;
```

Create an iterator for `vector<int>` objects.

Target the first element of `BigInt` and copy it.

Step to the second element of `BigInt`.

Now it targets a non-element of `BigInt`. Dereference will yield a garbage value.

Back it up to the last element of `BigInt`. 