Type Aliases

A name may be defined as a synonym for an existing type name. Traditionally, `typedef` is used for this purpose. In the new standard, an alias declaration can also be used `C++11`. The two forms are equivalent.

```
using newType = existingType; // C++11
typedef existingType newType; // equivalent, still works
```

Examples:

```
using counter_type = unsigned long;
using table_type = std::vector<int>;
```

The auto Type Specifier C++11

Sometimes type names are long and tedious to write, sometimes it can be difficult for the programmer to remember the exact type of an expression. The compiler, however, has no problems to deduce a type. The auto keyword tells the compiler to deduce the type from the initializer. Examples:

```
vector<int> v; // a vector is like a Java ArrayList
...
auto it = v.begin(); // begin() returns vector<int>::iterator

auto func = [](int x) { return x * x; }; // a lambda function
```

Do *not* use auto when the type is obvious, for example with literals.

```
auto sum = 0; // silly, sum is int
```

Auto Removes & and const

When auto is used to declare variables, "top-level" & and const are removed. Examples:

```
int i = 0;
int& r = i;
auto a = r;  // a is int, the value is 0
auto& b = r; // b is ref to int

const int ci = 1;
auto c = ci;       // c is int, the value is 1
const auto d = ci; // d is const int, the value is 1
```

The decltype Type Specifier C++11

Sometimes we want to define a variable with a type that the compiler deduces from an expression, but do not want to use that expression to initialize the variable. decltype returns the type of its operand:

```
sometype f() { ... }

decltype(f()) sum = 0;

vector<Point> v;
...
for (decltype(v.size()) i = 0; i != v.size(); ++i) { ... }
```

- f() and v.size() are not evaluated.

Strings

As in Java, the string type is a standard class. Unlike Java, strings are modifiable (similar to Java's string builders/string buffers).

```
#include <iostream>
#include <string>

int main() {
    std::cout << "Type your name ";
    std::string name;
    std::cin >> name;
    std::cout << "Hello, " + name << std::endl;
}
```

- Overloaded operators: <<, >>, +, < , <=, ...
- Character access: `s[index]`, no runtime check on the index.
- Operations: `s.size()`, `s.substr(start, nbr)`, many more.

Looping Through a string

```
string s = ...;

for (string::size_type i = 0; i != s.size(); ++i) {
    cout << s[i] << endl;
    s[i] = ' ';
}
```

The new standard introduced a *range-based* `for` statement C++11:

```
for (auto& c : s) {
    cout << c << endl;
    c = ' ';
}
```

Note the reference: it is necessary since the characters are modified.

A Note About != and ++

For statements, Java and C++ style:

```
for (int i = 0; i < 10; i++) { ... }  // Java

for (int i = 0; i != 10; ++i) { ... } // C++
```

In C++, testing for end of range with `!=` is preferred instead of `<`. And pre-increment (`++i`) is preferred instead of post-increment (`i++`). When the loop variable is of a built-in type, like here, both alternatives are equivalent. But the loop variable can be an object of a more complex type, like an iterator, and not all iterators support `<`. And pre-increment is more efficient than post-increment.

Vectors

The class `vector` is a template class, similar to Java's `ArrayList`. A vector can hold elements of arbitrary type. An example (read words and print them backwards):

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;
int main() {
    vector<string> v;
    string word;
    while (cin >> word) {
        v.push_back(word);
    }
    for (int i = v.size() - 1; i >= 0; --i) {
        cout << v[i] << endl;
    }
}
```

Vector Notes

- A vector is initially empty, elements are added at the end with `push_back`.
- Storage for new elements is allocated automatically.
- Element access with `v[index]`, no runtime check on the index.
- Elements cannot be added with subscripting: `vector<int> v;` `v[0] = 1` is wrong.
- The subscript type is `vector<T>::size_type`; usually this is `unsigned long`.
- Vectors may be copied: `v1 = v2`; and compared: `v1 == v2`.

Introducing Iterators

- `vector` is one of the container classes in the standard library. Usually, *iterators* are used the access the elements of a container (not all containers support subscripting, but all have iterators).
- An iterator "points" to one of the elements in a collection (or to a location immediately after the last element). It may be *dereferenced* with `*` to access the element to which it points and incremented with `++` to point to the next element.
- Containers have `begin()` and `end()` operations which return an iterator to the first element and to one past the end of the container.

Traversing a vector

```
vector<int> v;
...
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    *it = 0;
}
```

Better with auto C++11:

```
for (auto it = v.begin(); it != v.end(); ++it) {
    *it = 0;
}
```

Even better with a range-based for C++11:

```
for (int& e : v) {
    e = 0;
}
```

Iterator Arithmetic

Vector iterators support some arithmetic operations. Find the first negative number in the second half of the vector `v`:

```
auto it = v.begin() + v.size() / 2; // midpoint
while (it != v.end() && *it >= 0) {
    ++it;
}
if (it != v.end()) {
    cout << "Found at index " << it - v.begin() << endl;
} else {
    cout << "Not found" << endl;
}
```

const Iterators

A "normal" iterator can be used to read and write elements. A `const_iterator` may only be used to read elements. `cbegin()` and `cend()`, new in C++11, return const iterators.

```
vector<int> v;
...
for (auto it = v.cbegin(); it != v.cend(); ++it) {
    cout << *it << endl;
}
```

When the container is a constant object, `begin()` and `end()` return constant iterators. Example (more about parameters later):

```
void f(const vector<int>& v) {
    for (auto it = v.begin(); it != v.end(); ++it) {
        *it = 0; // Wrong -- 'it' is a constant iterator
    }
}
```

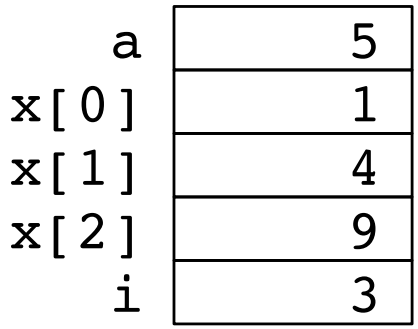
Arrays

- Bjarne Stroustrup: "The C array concept is broken and beyond repair."
- Modern C++ programs normally use vectors instead of built-in arrays.
- There are no checks on array subscripts during runtime (nothing like Java's `ArrayIndexOutOfBoundsException`). With a wrong index you access or destroy an element outside the array.
- There are two ways to allocate arrays: on the stack (next slides) and on the heap (after pointers).

Stack-Allocated Arrays

```
void f() {
    int a = 5;
    int x[3];  // size must be a compile-time constant
    for (size_t i = 0; i != 3; ++i) {
        x[i] = (i + 1) * (i + 1);
    }
    ...
}
```

Activation record:



More on Arrays

- Array elements can be explicitly initialized:

```
int x[] = {1, 4, 9};
```

- String literals are character arrays with a null terminator:

```
char ca[] = "abc"; // ca[0] = 'a', ..., ca[3] = '\0'
```

- Arrays cannot be copied with the assignment operator or compared with the comparison operators.
- In most cases when an array name is used, the compiler substitutes a pointer to the first element of the array:

```
int* px1 = x;
int* px2 = &x[0]; // equivalent
```

Pointers and Arrays

You may use a pointer to access elements of an array, and use pointer arithmetic to add to/subtract from a pointer. Pointers are iterators for arrays.

```
int x[] = {0, 2, 4, 6, 8};
for (int* px = x; px != x + 5; ++px) {
    cout << *px << endl;
}
```

When a pointer is incremented, the increments are in the size of the addressed data type, so `px + 1` means `px + sizeof(T)` for an array of type `T`. You may also subtract two pointers to get the number of array elements between the pointers.

Accessing array elements with subscripting or through pointers is equivalent. Actually, subscripting is defined as follows:

```
x[index] <=> *(x + index)
```

Library `begin` and `end` Functions C++11

In the example on the previous slide we obtained a pointer to the first element with the array name, a pointer past the last element with the array name plus the number of elements. For a `vector`, we would have used the `begin` and `end` members.

Since an array is a built-in type it doesn't have any member functions. Instead, there are library functions for this purpose:

```
int x[] = {0, 2, 4, 6, 8};
for (int* px = begin(x); px != end(x); ++px) {
    cout << *px << endl;
}
```

The global `begin` and `end` functions can be used also for other containers.

C-Style Strings

A C-style string is a null-terminated array of characters. In C++, C-style strings are almost only used as literals: `"asdfg"`. The C library `<cstring>` contains functions to manipulate C-style strings, for example `strlen` (string length), `strcpy` (string copy), `strcmp` (string compare). These functions are difficult to use correctly, since the caller is responsible for allocating storage for the strings.

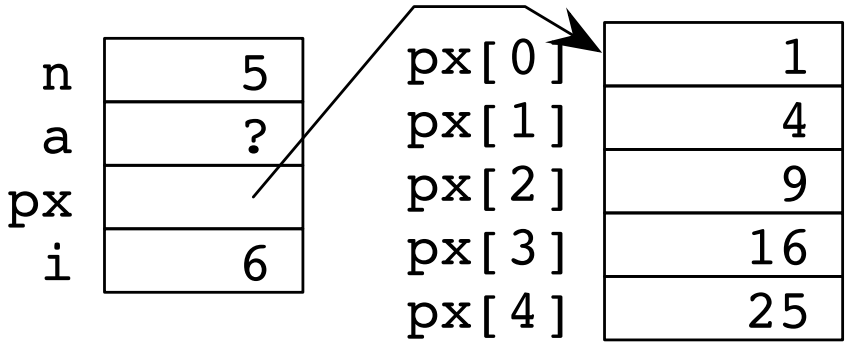
C++ strings and C-style strings can co-exist. In class `string`, most functions and operators are overloaded to accept C-style strings as parameters. Examples:

```
const char* cs = "asdfg";
string s = cs;
s += ".";
string::size_type pos = s.find("df");
```

Heap-Allocated Arrays

Arrays on the heap are similar to Java arrays, but you must delete the array after use. Example:

```
void g(size_t n) {
    int a;
    int* px = new int[n]; // size may be dynamic, >= 0
    for (size_t i = 0; i != n; ++i) {
        px[i] = (i + 1) * (i + 1);
    }
    ...
    delete[] px; // note []
}
```



Array Notes

- A heap-allocated array must be accessed via a pointer.
- A heap-allocated array does not contain information about its length.
- The global `begin()` and `end()` iterator functions cannot be used for heap-allocated arrays.
- `delete[]` is necessary to delete the array (otherwise the objects in the array will not be destroyed).