PROXY CLASSES

Array Definitions

- Array dimensions must be *constant expressions*
- Constant expression's value cannot change and must be evaluated at compile time
- What can be constant expression?
  - Literal is constant expression
  - `const` object or `constexpr` object initialized from constant expression

Array Definitions: What is Legal?


```
int data1[10][20]; // 2D array: 10 by 20
```

```
int const Rows{10}, Cols{20};
int data2[Rows][Cols]; // 2D array: Rows by Cols
```

```
constexpr int CRows{10}, CCols{20};
int data3[CRows][CCols]; // 2D array: CRows by CCols
```

```
enum class HLP1 : int { STUDENT = 10, TEST = 20 };
int data4[STUDENT][TEST]; // 2D array: STUDENT by TEST
```

Array Definitions: What is Illegal?

- Corresponding constructs using variables as dimension sizes are illegal!!!

```
// error!!! array dimensions must be known at compile time
int process_input(size_t dim1, size_t dim2) {
  int data[dim1][dim2];
  // other irrelevant code here ...
}
```

```
// error!!! not even legal for heap-based allocations
int process_input(size_t dim1, size_t dim2) {
  int *data = new int [dim1][dim2];
  // other irrelevant code here ...
}
```

Implementing 2D Arrays

- Define class template for 2D arrays to support objects we need but are missing from language proper:

```
template <typename T>
class Array2D {
public:
  Array2D(size_t dim1, size_t dim2);
  // ...
private:
  T *ptr;
  // ...
};
```

```
// now define the arrays we want:
Array2D<int> data(10, 20); // ok
Array2D<float> *data =
     new Array2D<float>(10, 20); // ok

void process_input(size_t dim1, size_t dim2) {
  Array2D<int> data(dim1, dim2); // ok
  ...
}
```

Implementing 2D Arrays

- Need to declare subscript operator in `Array2D` to let us do this: `std::cout << data[2][3];`
- First impulse is to declare `operator[][]` functions:

```
template <typename T>
class Array2D {
public:
  // declarations that won't compile
  T& operator[][](size_t index1, size_t index2);
  T const& operator[][](size_t index1, size_t index2) const;
  ...
};
```

Won't compile because there is such a thing as `operator[]` but no such thing as `operator[][]`

Implementing 2D Arrays

- We could use parentheses to index into arrays by overloading `operator()`:

```
template <typename T>
class Array2D {
public:
  // declarations that will compile
  T& operator()(size_t index1, size_t index2);
  T const& operator()(size_t index1, size_t index2) const;
  ...
};

// clients could use arrays this way:
Array2D<int> data;
std::cout << data(2, 3);
```

Implementing 2D Arrays


```
// clients could use arrays this way:
Array2D<int> data;
std::cout << data(2, 3);
```

Drawback is that your `Array2D` doesn't look like built-in arrays any more.
In fact, above access to element at row 2 and column 3 of `data` looks like a function call!!!

Implementing 2D Arrays

- Thinking more deeply, we analyze why the following code works:

```
int data[10][20];
...
cout << data[3][4]; // ok
```

We recall data is not really a 2D array at all, it's a 10-element one-dimensional array!!!

So expression data[3][4] really means (data[3])[4], i.e., fifth element of array that is fourth element of data

In short, value yielded by 1st application of brackets on an array data is another array, so 2nd application of brackets gets an element from that secondary array

Implementing 2D Arrays

- We should play same game in class `Array2D` by overloading `operator[]` to return object of new class `Array1D`
- Next, we overload `operator[]` again in `Array1D` to return an element in original two-dimensional array

Implementing 2D Arrays


```
template <typename T>
class Array2D {
public:
  class Array1D {
  public:
    T& operator[](size_t idx);
    T const& operator[](size_t idx) const;
    ...
  };

  Array1D operator[](size_t idx);
  const Array1D operator[](size_t idx) const;
  // other data members and functions ...
};
```

```
// this is now legal
Array2D<int> data(10, 20);
...
cout << data[3][5]; // ok
```

Proxy Objects and Proxy Classes

- Each `Array1D` object stands for a one-dimensional array that is absent from conceptual model used by clients of `Array2D`
- Objects that stand for other objects are called *proxy objects*, and classes that give rise to proxy objects are called *proxy classes*
- `Array1D` is a proxy class – its instances stand for one-dimensional arrays that, conceptually don't exist

Another Simple Solution


```
class Matrix {
  float m_matrix[4][4];
public:
// for statements like matrix[0][0] = 1;
  float* operator [] (int index) {
    return m_matrix[index];
  }

// for statements like matrix[0][0] = otherMatrix[0][0];
  float const* operator [] (int index) const {
    return m_matrix[index];
  }
};
```

Proxy Objects and Proxy Classes

- Proxy is useful in other scenarios …
- For example, `const` member functions are significantly faster than non`const` counterparts
- How to make `const` member function be invoked?

```
class Example {
public:
  // significantly faster ...
  int const & Access() const;
  int       & Access();
};
```

Proxy Objects and Proxy Classes

- One way is to provide a wrapper function …
- Disadvantage is that clients must remember to use this function rather than `Access`
- Another way is to use a proxy class …

```
class Example {
public:
  int const & Access() const;
  int       & Access();
  int const & CAccess() const { return Access(); }
  // No non-const Caccess:
  // so always calls `int const& Access() const`
};
```