Another Example of Abstraction

A class for page buffers

A problem

- Suppose we have a program that generates integers
- We want to print those integers in pages that are divided into columns
- Assume that `rows` is the number of rows and `columns` is the number of columns on each page

How might the interface look?

- If we were solving this problem in C, we would probably write something like:

```
void start();
void print(int);
void finish();
```

A note on interfaces

- Our "finish" operation presumably prints the last (partial) page. Why isn't there an operation to print the other pages?
- The other pages can be printed "automatically" when we try to put a number into a buffer that is full already

Buffer definition

```
static const int rows = 50;
static const int columns = 5;
static int buffer[rows][columns];
static int row, col;
```

Buffering conventions

- We will fill the buffer a column at a time and print it a row at a time
- Whenever we are about to put a value into the buffer, we will put it at position `(row, col)`
- After putting something into the buffer, we will increment `(row, col)` and flush the buffer if needed

Initialization

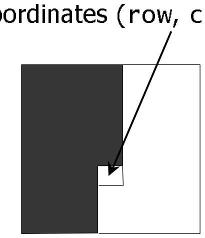
```
void start()
{
    row = 0;
    col = 0;
}
```

“Print” a number

```
void print(int n)
{
    buffer[row][col] = n;
    if (++row == rows) {
        row = 0;
        if (++col == columns) {
            flush_buffer();
            col = 0;
        }
    }
}
```

Empty the buffer

- Assume the first unused buffer element is at coordinates (`row`, `col`)



Implications

- We would like the same code to print full pages as will print the partial page at the end of the output.
- The width of a row is either `col+1` (for rows less than `row`) or `col` (otherwise)
- If the first column is not full, we will print one or more empty rows (because `col == 0` and `row < rows`)

Emptying the buffer

```
static void flush_buffer()
{
    for (int r = 0; r < rows; ++r)
        print_row
            (r, r < row? col+1: col);
    if (row != 0 || col != 0)
        new_page();
}
static void new_page()
{
    cout << “\f”;
}
```

Printing a row

```
static void print_row(int r, int w)
{
    if (w != 0) {
        for (int j = 0; j < w; ++j)
            cout << setw(8) <<
                    buffer[r][j];
        cout << endl;
    }
}
```

The last page

```
void finish()
{
    flush_buffer();
}
```

Policy decisions

- How shall we print an empty row?
- How shall we print an empty page?
- What is the format within a row?
- How large are the pages?
- All these decisions are well isolated

The object of the game

- This program has disadvantages:
  - Only one buffer
  - Tied to a single file
  - Two-dimensional array
  - Fixed size
- We would like to avoid these disadvantages

Allowing multiple buffers

- The way to allow for more than one of something is to make that something an object
- If it is an object, it must have a type, so we will need a corresponding class definition
- Each object of the class we define will represent a buffer

Allowing multiple files

- We need a way to connect a buffer to a file
- It is probably good enough to say what the file is when we create the buffer
- We can give the buffer a constructor with an `ostream*` as an argument

Memory management

- Instead of using a static two-dimensional array, we would like to use a dynamic one-dimensional array
- We will give the size when we create the object

Using a one-dimensional array

- Fiddling with separate `row` and `column` variables is a nuisance, and hard to get right
- Instead, let's try putting values sequentially into a one-dimensional array and printing them as needed

Indices

- It should not be hard to figure out how the index of an array element corresponds to where it is on the page
  - The first column has indices `[0, rows)`
  - The next column has `[rows, 2*rows)`
  - And so on until the last column (column number `columns-1`), which has indices `[(columns-1)*rows, columns*rows)`

Interface definition

We can already start coding:

```
class Buffer {
public:
    Buffer(ostream*, int, int);
    ~Buffer();
    void print(int);
};
```

Constructor: `Buffer(ostream*, int, int);`
Destructor: `~Buffer();`

How will we implement it?

- We need to store
  - (a pointer to) the buffer itself
  - the number of rows and columns
  - the total size (rows * columns)
  - how many elements are used
  - the file we are using
- We can (and should) make all these data `private`

Expanded class definition

```
class Buffer {
public:
    Buffer(ostream*, int, int);
    void print(int);
    ~Buffer();

private:
    int h, w, size, n;
    ostream* f;
    int* b;
};
```

How might we use it?

```
Buffer b1(&cout, 50, 5);
b1.print(n);

Buffer b2(&cerr, 24, 80);
b2.print(x);
```

Machine-checkable specifications

- At this point, we could compile our class definition and a program that uses it
- Of course, we could not execute the program, because we have not defined the member functions
- Still, the ability to compile code lets us find out how it feels to use the class

The `Buffer` constructor

```
Buffer::Buffer
    (ostream* f0, int h0, int w0):
        h(h0), w(w0), size(h0 * w0),
        n(0), f(f0), b(new int[size])
{ }
```

Nothing else to do here

These are *constructor initializers;* they are executed in the order of the class declaration(so they should appear in the same order in the definition)

Allocating memory with new

- The new-expression in C++ is a type-safe alternative to `malloc`
- If T is the name of a type, then
  - `new T` allocates an object of type T and returns a pointer to it
  - `new T[n]` allocates an n-element array of T and returns a pointer to its initial element
- To free the memory, use `delete` or `delete[]`, depending on whether you allocated an array

Other properties of new

- Using `new` executes constructors
- Using `delete` executes destructors
- If memory allocation fails, `new` throws an exception

"Printing" a number

```
void Buffer::print(int x)
{
    b[n] = x;
    if (++n == size) {
        flush();
        n = 0;
    }
}
```

n and `size` are known to be members of `Buffer` within the body of a member function of `Buffer`

Flushing the buffer

- We can't just make `flush` an ordinary function the way we did before
  - It wouldn't have access to the private data
  - It wouldn't know which `Buffer` to flush
- We must therefore make it a member function
- Because we don't want people to call it directly, we'll make it private

Revising the class definition

```
class Buffer {
public:
    // as before
private:
    int h, w, size, n;
    ostream* f;
    int* b;
    void flush();        // new
    void new_page();     // new
    void print_row(int); // new
};
```

Flushing the buffer

```
void Buffer::flush()
{
    for (int r = 0; r < h; ++r)
        print_row(r);
    if (n != 0)
        new_page();
}
void Buffer::new_page()
{
    *f << "\f";
}
```

Printing a row

```
void Buffer::print_row(int r)
{
    while (r < n) {
        *f << setw(8) << b[r];
        r += h;
    }
    *f << endl;
}
```

The destructor

```
Buffer::~Buffer()
{
    flush();
    delete[] b;
}
```

What have we gained?

- Each `Buffer` object holds all the information about a particular buffer
- Users can't get their hands on the implementation data
- The interface is explicit—it's just the public section(s) of the class definition
- The auxiliary functions have disappeared from view

Not quite the whole truth...

- In C, copying a structure copies its elements
- C++ therefore behaves similarly unless you ask otherwise...
- ...and asking otherwise requires a bit of explanation of language features
- For now, just don't copy a `Buffer`
- Meanwhile, let's get started with the explanations...

References

- A reference is a way of giving a(nother) name to an object

```
int x = 3;
int& y = x;   // y is now another name for x
y = 42;       // x is now 42
```

- Reference types look (syntactically) like pointer types, except that they use & instead of *

Reference examples

```
int x[3] = { 8, 1, 6 };
int i = 2;
int& y = x[i];
i = 1;
y = 7;        // x is now { 8, 1, 7 }
```

Why bother with references?

- Attach a temporary name to an object inside a complicated data structure
- Implement call by reference
  - Allow a function to modify its arguments
  - Pass an argument that does not have copying defined
  - Avoid copying for efficiency reasons
- Copy constructors

A function that modifies its argument

```
void clobber(int& x)
{
    x = 0;
}

int main()
{
    int i = 3;
    clobber(i);       // i is now 0
}
```

Passing uncopyable arguments

- We have a `Buffer` class whose objects cannot legitimately be copied
- How can we pass a `Buffer` called, say, b as an argument?

```
void f(Buffer*); f(&b);
void g(Buffer&); g(b);
```

Reference to `const`

- As we can have a pointer to a constant, we can have a reference to a constant
  - The usual purpose is to avoid copying where possible
  - Therefore, the compiler will create a copy for us automatically if it cannot be avoided (instead of complaining)
- Typical syntax: `const T&`

Examples of references to `const`

```
int i = 3;
const int& j = i;
i = 4;          // j is now 4
j = 3;          // error: j is const
int& x = 10;  // error: 10 is not an object
const int& y = 10; // y now names
                   // a copy of 10
```

Copy constructors

- A constructor is called a *copy constructor* if its (only) argument is a reference to an object (usually `const`) of its class

  `Buffer::Buffer(const Buffer&);`
- The copy constructor controls how every object of its class it copied

Additional pieces

- Controlling copies of class objects is not quite enough: It is also necessary to control assignment, which is different

  ```
  T y = x; // creates y as a copy of x
  y = x;   // copies x on top of y
  ```
- Assignment is controlled through `operator=` (the assignment operator)
- More details next week