Allocating Objects with a Variable-Length Field

```
class Rec {
private:
      Rec(int id, double gpa, short sal) {
            ID = id;
            GPA = gpa;
            salary = sal;
      }

      ~Rec() { }

public:
      static Rec* construct(int id, double gpa, short sal, int stuffLen)
      {
            char* space = new char[sizeof(Rec) + stuffLen];
            Rec* rec = new(space) Rec(id, gpa, sal);
            return rec;
      }

      void destroy()
      {
            this->~Rec();
            delete[] reinterpret_cast<char*>(this);
      }

      int ID;
      double GPA;
      int salary;
      char major[4];
      char stuff[];
};
```

Private constructor/destructor prevents code from allocating these objects directly (which would prevent us from controlling the size of the variable-length field).

The construct method allocates a memory buffer equal to the size of the defined fields in the class plus the amount of space for the variable-length array that the user requests. The parameterized version of new is used to initialize the object.

We cannot delete directly an object allocated in this fashion. Instead, we call the destructor explicitly and then free the original buffer used to hold the object.

The variable-length field occurs at the end of the class so that it occupies the extra memory that was allocated.

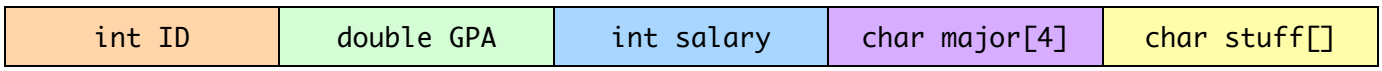
Caveat: Not *quite* standards-compliant C++.

Using the Variable-Length Object

```
const int STUFF_LENGTH = 36;

char major[4] = { 'C', 'S', 'E', 'D' };

char stuff[STUFF_LENGTH] = { 0 };
strncpy(stuff, "John Doe:1002 Anywhere Street", STUFF_LENGTH);

Rec* myrecord = Rec::construct(111223333, 2.345, 2222, STUFF_LENGTH);
memcpy(&(myrecord->major), major, 4 * sizeof(char));
memcpy(&(myrecord->stuff), stuff, STUFF_LENGTH * sizeof(char));

ofstream myFile("data4.bin", ios::out | ios::binary);
myFile.write((char*)myrecord, sizeof(Rec) + STUFF_LENGTH);

...

ifstream myRead("data4.bin",ios::in | ios::binary);
Rec* readRecord = Rec::construct(0, 0, 0, STUFF_LENGTH);
myRead.read((char*)readRecord, sizeof(Rec) + STUFF_LENGTH);

...

myrecord->destroy();
readRecord->destroy();
```

Record Layout In Memory and On Disk

```
0000:  25 22 A1 06 00 00 00 00 C3 F5 28 5C 8F C2 02 40  %".........(\...@
0010:  AE 08 00 00 43 53 45 44 4A 6F 68 6E 20 44 6F 65  ....CSEDJohn Doe
0020:  3A 31 30 30 32 20 41 6E 79 77 68 65 72 65 20 53  :1002 Anywhere S
0030:  74 72 65 65 74 00 00 00 00 00 00 00              treet.......
```



Question: Where did the four 0-bytes occupying addresses 4-7 come from?

Advantages to Using Variable-Length Objects

- Relatively little boilerplate code had to be written to manage object allocation/deallocation
- The object can still be read/written in a single operation by passing the pointer to and size of the object to the appropriate function

Disadvantages to Using Variable-Length Objects

- Still only works for the most basic of objects – none of the fields can be pointers, or objects that contain pointers
- Only one variable-length field can be used
- Classes with virtual functions are also out – the hidden virtual function table (vtable) pointer would be written to disk as part of the object; reading it back would be disastrous

More Advanced Object Serialization

```
class Serializable {
public:
      virtual MemHandle writeObject(MemManager* manager) = 0;
      virtual void readObject(MemManager* manager, MemHandle handle) = 0;
};

class Contact : public Serializable {
public:
string name;
list<int> phoneNumbers;

MemHandle writeObject(MemManager* manager)
{
      int strLength = name.length() + 1;
      int phoneCount = phoneNumbers.size();

      int objLength =
              sizeof(int)                 // length of string
            + strLength                   // chars in string + final NULL
            + sizeof(int)                 // # of phone numbers
            + phoneCount * sizeof(int);   // phone numbers in sequence

      char* objData = new char[objLength];
      char* temp = objData;

      memcpy(temp, &strLength, sizeof(int));  temp += sizeof(int);
      memcpy(temp, name.c_str(), strLength);  temp += strLength;
      memcpy(temp, &phoneCount, sizeof(int));  temp += sizeof(int);

      for(list<int>::iterator it = phoneNumbers.begin();
            it != phoneNumbers.end(); ++it)
      {
            int phoneNum = *it;
            memcpy(temp, &phoneNum, sizeof(int));  temp += sizeof(int);
      }


      MemHandle handle = manager->insert(objData, objLength);
      delete[] objData;

      return handle;
}
```

Need to precalculate the amount of space the object will require in its serialized representation before we can allocate the buffer to pass to the memory manager.

Copy any data about the object into the buffer, in any format that we see fit.

Send the buffer with the serialized object to the memory manager, free the buffer, and return the handle to the object on disk.

More Advanced Object Serialization

```
void readObject(MemManager* manager, MemHandle handle)
{
	int objLength = manager->get(0, handle);

	char* objData = new char[objLength];
	char* temp = objData;

	manager->get(objData, handle);

	int strLength, phoneCount;

	memcpy(&strLength, temp, sizeof(int));  temp += sizeof(int);

	// This is kind of a trick -- since temp has been advanced to the first
	// character of the null-terminated name in the buffer, we can just
	// pass the char* to the string constructor and it will pull in the
	// entire string, stopping when it reaches the null char.
	name = string(temp);  temp += strLength;

	memcpy(&phoneCount, temp, sizeof(int));  temp += sizeof(int);

	phoneNumbers.clear();
	for(int i = 0; i < phoneCount; i++)
	{
		int phoneNum;
		memcpy(&phoneNum, temp, sizeof(int));  temp += sizeof(int);
		phoneNumbers.push_back(phoneNum);
	}

	delete[] objData;
}
```

Modified `get()` method just returns size of block if passed a null pointer.

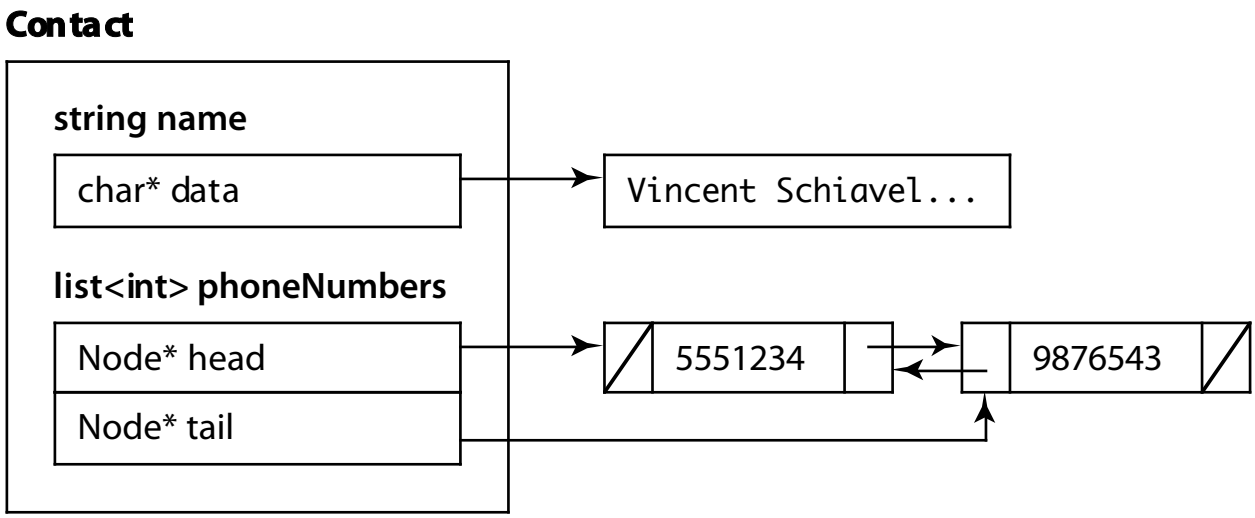
Read the data back out of the buffer and into the fields of the object, assuming the format used by the `writeObject()` method.

Don't forget to delete the temporary buffer after the object is loaded.

Using the Serialization Methods

```
MemManager* manager = /* pointer to a memory manager */;

Contact* contact1 = new Contact();
contact1->name = "Vincent Schiavelli";
contact1->phoneNumbers.push_back(5551234);
contact1->phoneNumbers.push_back(9876543);

MemHandle contact1Handle = contact1->writeObject(manager);

// Later in the program...

Contact* contact1Read = new Contact();
contact1Read->readObject(&manager, contact1Handle);
```

Record Layout In Memory



Record Layout On Disk

```
0000:  13 00 00 00 56 69 6E 63 65 6E 74 20 53 63 68 69  ....Vincent Schi
0010:  61 76 65 6C 6C 69 00 02 00 00 00 82 B4 54 00 3F  avelli.......T.?
0020:  B4 96 00                                         ...
```

Length of name	Characters of name, including final null	Number of elements in phoneNumbers	Elements of phoneNumbers

Simplifying the Copying of Data into the Buffer

- Previous approach forced us to keep track of a pointer that "walks across" the buffer so that we can copy each item of data one after the other, or to read items from the buffer
- This, along with the `memcpy()` calls, is very error-prone; user could forget to advance the pointer, pass in the wrong data size, etc.
- Pre-calculating the size that the object will require on disk can be non-trivial if the structure of the object is complex

Creating a "Binary Stream" Class

- Consider the `stringstream` classes – users can "push" and "pull" data to/from them in a sequence
- The output stream maintains a pointer to its current position and automatically grows the buffer as needed to fit more data
- The input stream takes an existing string and reads data from it, also maintaining a pointer to the current position in the buffer
- We can write some basic stream classes that mimic this functionality, but with binary data instead of strings

Binary Output Stream Class

```
class obinarystream {
private:
      char* _data;
      char* _currentPtr;
      int _size;
      int _capacity;

      void growBuffer();

public:
      obinarystream() {
            _capacity = 16;
            _size = 0;
            _data = new char[_capacity];
            _currentPtr = _data;
      }

      ~obinarystream() { delete[] _data; }

      char* data() const { return _data; }

      int size() const { return _size; }

      template <typename Elem>
      friend obinarystream& operator<<(obinarystream& stream, const Elem& e);
};

template <typename Elem>
obinarystream& operator<<(obinarystream& stream, const Elem& e)
{
      while(stream._size + sizeof(Elem) > stream._capacity)
            stream.growBuffer();

      stream._size += sizeof(Elem);
      memcpy(stream._currentPtr, &e, sizeof(Elem));
      stream._currentPtr += sizeof(Elem);

      return stream;
}
```

As before, we keep track of the current position in the buffer. Instead of pre-calculating the size, we increase it as data is pushed in.

We need to be able to get a pointer to the internal buffer and its size to pass to the memory manager.

A templated `operator<<` lets us copy just about anything into the memory buffer. The implementation simply does a direct `memcpy()` of the data in the object passed in.

Binary Input Stream Class

```
class ibinarystream
{
private:
      char* _data;
      char* _currentPtr;
      int _size;

public:
      ibinarystream(int sz) {
            _size = sz;
            _data = new char[_size];
            _currentPtr = _data;
      }

      ~ibinarystream() { delete[] _data; }

      char* data() const { return _data; }

      int size() const { return _size; }

      template <typename Elem>
      friend ibinarystream& operator>>(ibinarystream& stream, Elem& e);
};

template <typename Elem>
ibinarystream& operator>>(ibinarystream& stream, Elem& e)
{
      memcpy(&e, stream._currentPtr, sizeof(Elem));
      stream._currentPtr += sizeof(Elem);

      return stream;
}
```

We let the `ibinarystream` allocate and manage the buffer for us. We can use the `data()` function to get a pointer to the internal buffer that we can pass to the memory manager's `get()` function, which will fill the buffer with data.

The implementation of `operator>>` is even simpler because there is no need to resize the buffer. We merely copy `sizeof(Elem)` bytes into the element reference passed in as an argument, then advance the current position that far.

Using the Binary Stream Classes

```
MemHandle writeObject(MemManager* manager) {
      obinarystream os;
      os << name.length() + 1;

      for(int i = 0; i < name.length(); i++)
            os << name[i];
      os << '\0';

      os << phoneNumbers.size();
      for(list<int>::iterator it = phoneNumbers.begin();
            it != phoneNumbers.end(); ++it) {
            int phoneNum = *it;
            os << phoneNum;
      }

      MemHandle handle = manager->insert(os.data(), os.size());
      return handle;
}

void readObject(MemManager* manager, MemHandle handle) {
      int objLength = manager->get(0, handle);
      ibinarystream is(objLength);
      manager->get(is.data(), handle);

      int nameLength, phoneCount;
      char ch;

      is >> nameLength;

      name.clear();
      for(int i = 0; i < nameLength - 1; i++) {
            is >> ch;
            name += ch;
      }
      is >> ch;

      is >> phoneCount;
      phoneNumbers.clear();
      for(int i = 0; i < phoneCount; i++) {
            int phoneNum;
            is >> phoneNum;
            phoneNumbers.push_back(phoneNum);
      }
}
```

We write the string character-by-character, because passing a string object to `obinarystream` would write the pointers inside the string object rather than the character data.

Likewise, we have to read the string back a character at a time, remembering to also pull the final null character out of the stream before moving on.