An Introduction to Writing Externs in C for Max/MSP

Paul Gurnig
University of Chicago
Chicago, IL
gurnig@uchicago.edu

Abstract

The target reader of this paper is a Max/MSP[1] user with limited programming experience. This paper analyzes a single sample program adapted from Ichiro Fujinaga's tutorial on compiling externs using Xcode. Code samples are shown in Xcode, but a variety of programming environments is possible. The example discussed here introduces and explains the necessary framework to begin developing more complex externs. Knowing this framework is extremely helpful in developing externs.

[If you're only interested in the code discussion and not in getting setup, skip ahead to section 3.]

1 Overview

A language provides a vocabulary and the rules for combining words in the vocabulary.

Grady Booch

If we were to apply Booch's comment to Max, we might first observe that Max has a considerable native vocabulary (sfplay~, ezdac~, ctlin, etc.) and that the mechanism for treating the native Max vocabulary is robust, which in this case is to say that the mechanism imposes few rules. The framework is the blank canvas in Max, do what you want with it. One can simply combine the existing vocabulary in any desired way.

Even with that power, Max allows users to further syntactic extension. While creating a specialized vocabulary by writing externs is not necessary for the successful use of Max, it does allow developers the opportunity to extend the vocabulary and introduce new objects to the Max environment.

1.1 Recommended Sources

A word should be said about existing extern development documentation. There are limited, but excellent sources available. Two papers, "Writing External Objects for Max 4.0 and MSP 2.0" [2001] and "Writing External Objects for Max and MSP" [2005], are available from Cycling '74. Both of these papers are excellent, easily readible and largely cover similar material. In fact, the 2005 version seems to be an updated 2001 version.

There are also a number of sources available online at McGill University in Montreal. Ichiro Fujinaga teaches a course there titled **Advanced Multimedia Development** that focuses on writing externs for Max. As of this writing, Fujinaga has three tutorials available, each for a different version of the Mac/Programming Environment (Classic, OS X & OS X Xcode). There are limited PowerPoint slides from his course and many externs that serve as great examples.

1.2 Development Environment

The subject of development environment is a bit beyond the scope of this paper, but is critical to the success of compiling the code discussed here. In the paper "External Objects for Max and MSP", Cycling '74 explains how to write externs using CodeWarrior on the Mac and Microsoft Visual Studio on the PC. One of Fujinaga's tutorials, "Max/MSP Externals Tutorial: Version 3.1", discusses using Xcode on the Mac. There is a more recent tutorial at www.cycling74.com, entitled "Writing Externals with Xcode 2.2". Getting the development environment set up correctly can be challenging.

[1]Hereafter refered to simply as Max.

1.3 Code used in this Paper

The code shown in this paper is taken *directly* from Fujinaga's example in his paper "Max/MSP Externals Tutorial: Version 3.1". The code presented here has been varied slightly. Fujinaga's paper is largely concerned with getting the development environment setup and does provide an overview of the code. This paper is a bit more in depth (in terms of this single example) and aims to explain Fujinaga's example in light of other sources, notably Zicarelli and general C programming constructs.

1.4 What you'll need to compile and run this code

In order to compile and run the code in this paper, you'll need the following.

1. Max/MSP[2]
2. The Max/MSP SDK[3]
3. A development environment[4] that will compile C code. This paper uses Xcode[5] on the Mac.

   You'll deploy your code to Max's extern folder and access it similarly to any other Max object. Be aware that your extern may **not** appear

2 What is an Extern?

An extern, or external, is a computer program, not part of Max's standard object palette, that will be consumed by Max. The program is in the form of a shared library or dll. The most common language used for externs is C, although a bridge called FlexT is available for C++.

There are two broadly defined types of externs – a normal object and a user interface object. A normal object rendered in Max looks like many Max objects, with either a single or double line both on the top and bottom of the object, inlets and outlets, the object's name and arguments. A user interface object, beyond the scope of this document, brings its own UI to the table[6].

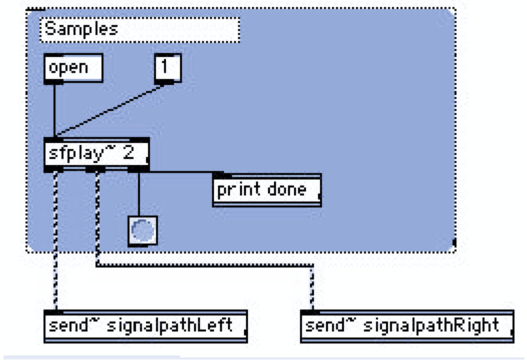


Figure 1. Some typical non-UI Max objects.

In terms of C code, at a high level, an extern essentially consists of an entry point (the main() function), a description of the object (in the form of a struct) and definitions of functionality (in the form of methods). Certain methods and elements of the struct are required by Max.

3 An Example

Learning the framework of an extern, along with compiling and deployment will go a long way to getting into some more involved projects. The essential structure shown here is relevant to externs generally.

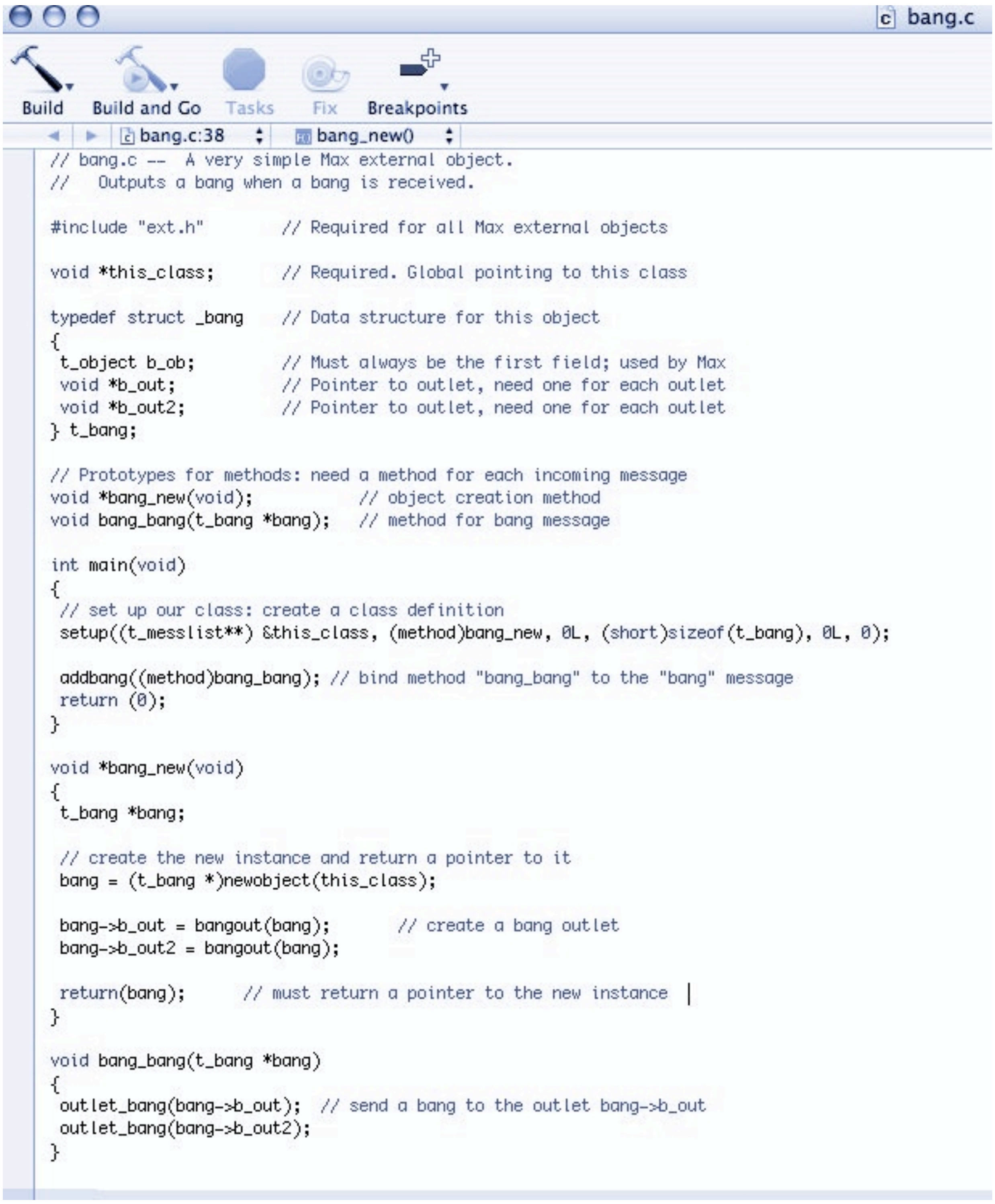
[2]A free trial version, as well as a student version, is available.

[3]Available from www.cycling74.com.

[4]See Fujinaga's Tutorials or the documentation that comes with the SDK.

[5]Xcode is freely available from www.apple.com.

[6]Zicarelli covers writing User Interface Objects in « Writing External Objects for Max 4.0 and MSP 2.0 » [2001] and « Writing External Objects for Max and MSP » [2005]. Both documents are available at www.cycling74.com.



```c
// bang.c --  A very simple Max external object.
//   Outputs a bang when a bang is received.

#include "ext.h"        // Required for all Max external objects

void *this_class;       // Required. Global pointing to this class

typedef struct _bang    // Data structure for this object
{
 t_object b_ob;         // Must always be the first field; used by Max
 void *b_out;           // Pointer to outlet, need one for each outlet
 void *b_out2;          // Pointer to outlet, need one for each outlet
} t_bang;

// Prototypes for methods: need a method for each incoming message
void *bang_new(void);           // object creation method
void bang_bang(t_bang *bang);   // method for bang message

int main(void)
{
 // set up our class: create a class definition
 setup((t_messlist**) &this_class, (method)bang_new, 0L, (short)sizeof(t_bang), 0L, 0);

 addbang((method)bang_bang); // bind method "bang_bang" to the "bang" message
 return (0);
}

void *bang_new(void)
{
 t_bang *bang;

 // create the new instance and return a pointer to it
 bang = (t_bang *)newobject(this_class);

 bang->b_out = bangout(bang);       // create a bang outlet
 bang->b_out2 = bangout(bang);

 return(bang);      // must return a pointer to the new instance
}

void bang_bang(t_bang *bang)
{
 outlet_bang(bang->b_out);  // send a bang to the outlet bang->b_out
 outlet_bang(bang->b_out2);
}
```

Figure 2. bang.c Source Code in Xcode environment

Let's look at this code from the perspective of how Max runs it.

Max will load the extern's code starting with main() - main() provides *the* entry point for Max to gain access to the behavior you will provide[7] in your methods. main()'s job, in the context of this extern, is simply to “initialize its

[7]Zicarelli, *Writing External Objects for Max 4.0 and MSP 2.0*, 20

class”[8]. This means that main() is called either when Max loads (if your extern in your max-startup folder) or when you create an instance of your object at design time in the Max patcher window (this latter approach will occur if your object is **not** in your max-startup folder, but is somewhere where Max can find it).

3.1 setup()

As **Figure 2** shows, the first line of code in `main()` calls a function, `setup()`. If you spend some time coding Max objects, the details of this method call will become very familiar. This method is available by virtue of including ext.h which is the first non-commented line in the source code:

#include “ext.h”

`setup()` provides Max with the (1) information to find your object (where it lives in memory), (2) the name of your object's constructor (the housekeeping tasks performed to create your object in memory), (3) the name of its cleanup method (the housekeeping tasks performed to remove the object from memory), (4) the size of its data structure, (5) the name of the method that will define your object's UI (if you have one – we don't in this example) and (6) the type list[9]. That may seem like quite a lot, but some of the parameters are optional and can be sent zero values (0 or 0L in the case of a long datatype). In the example above, we only use three of the six available parameters. Moreover, the nasty business of doing object creation and destruction is handled for us in this simple example. That turns out to be quite nice. Here's the call to setup with the various arguments highlighted:

- the location (address) of your object:

  setup(**(t_messlist**) &this_class**, (method)bang_new, 0L, (short)sizeof(t_bang), 0L, 0);

- the name of your constructor (which is going to be in the form *programname*_new or *programname*_create):

  setup((t_messlist**) &this_class, **(method)bang_new**, 0L, (short)sizeof(t_bang), 0L, 0);

- a method that will do cleanup; in this case this is pointing to nothing

  setup((t_messlist**) &this_class, (method)bang_new, **0L**, (short)sizeof(t_bang), 0L, 0);

- the size of your internal data structure (required):

  setup((t_messlist**) &this_class, (method)bang_new, 0L, **(short)sizeof(t_bang)**, 0L, 0);

- the name of the method that will define your user interface[10]:

  setup((t_messlist**) &this_class, (method)bang_new, 0L, (short)sizeof(t_bang), **0L**, 0);

- an argument list you may pass to your class (here again, we are pointing to nothing, or null):

  setup((t_messlist**) &this_class, (method)bang_new, 0L, (short)sizeof(t_bang), 0L, **0**);

In our example, we're providing Max with three pieces of information: (1) the object address, (2) the constructor, (3) the size of the data structure. The other bits are not needed, but we still must provide values as this method is not overloaded[11].

3.2 addbang()

We have one more line of code in our main() method[12] and that line has a simple explanation. It's important, for the purposes of running our extern, to distinguish between where a certain method is located in memory (accomplished by binding our function) and what that method actually does (accomplished by defining our function). The `addbang((method)bang_bang)` method tells Max which of our functions to run when our extern encounters a bang. In short, it says, “when my object receives a bang, the name of the method that is to be

[8]Zicarelli, *Writing External Objects for Max 4.0 and MSP 2.0*, 5

[9]Zicarelli, *Writing External Objects for Max 4.0 and MSP 2.0*, 21

[10]Which is beyond the scope of this introductory paper.

[11]Overloading here refers to the concept of having multiple function signatures.

[12]return(0) is simply indicating that we're exiting normally.

invoked is bang_bang". `addbang()` is likely a method you'll encounter often when writing externs. `addbang()` has a number of add*something*() siblings: `addfloat()`, `addftx()`, `addint()`, `addinx()`, `addmess()`.

If our method name were *N_bang*, then the syntax for "binding" our method to our implementation is:

addbang((method)*N_bang*);

The method name for *N* is normally the name of our program followed by an underscore followed by the word bang. So, if we were writing an object called attenuator for Max and our object needed to accept a bang as input, our method name for handling such input might be *attenuator*_bang. Similarly, if we wanted methods to handle ints, floats, etc. in our attenuator object, we would have methods such as *attenuator*_int(), *attenuator*_float, etc. Typically, we bind these methods to inlets in our `main()` method.

3.3 Recap

So, in short, the `main()` routine of an extern is used to bind our methods.

3.4 Implementation

Telling Max how to build our object.

In Max, most objects have inlets and outlets. We send messages in, we get something out. Take the ubiquitous **notein** object, for example:

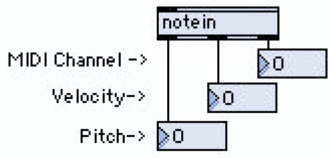


Figure 3. notein pictured in the patch from Max's help on notein.

notein gets its information coming in via the MIDI stream, so there is no visible connector going to an inlet[13]. The implementation of this object, behind the scenes, takes that stream of data, parses it, and outputs specific pieces of data (pictured above). In Max then, we can do something with this data – we can route pitch, operate on the values, etc.

In code, we describe the outlets in our data structure. In our example, that is this:

```
typedef struct _bang    // Data structure for this object
{
 t_object b_ob;         // Must always be the first field; used by Max
 void *b_out;           // Pointer to outlet, need one for each outlet
 void *b_out2;          // Pointer to outlet, need one for each outlet
} t_bang;
```

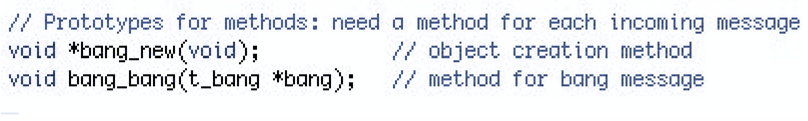
You may be asking, "where's the inlet defined?". One inlet is automatically created for an object so you don't have to create one manually in code. This is a nice default since typically we'll want at least one inlet. It is possible to instruct Max not to create an inlet in your constructor[14]. You do need to create any outlets in the `struct`, however, and you'll need to declare a void pointer for each outlet. We've created two outlets in our example here, simply for the sake of having more than one.

Telling Max about our methods

We're actually fairly far along in our analysis. Following our typedef, we have prototype declarations for the methods we'll implement after our `main()` function.

[13]The inlet you see pictured in **notein** is for enabling/disabling the object or setting the port.

[14]In your constructor, you would call `class_noinlet(`*thisclass*`)` and pass in *thisclass* where *thisclass* is the name of your class.



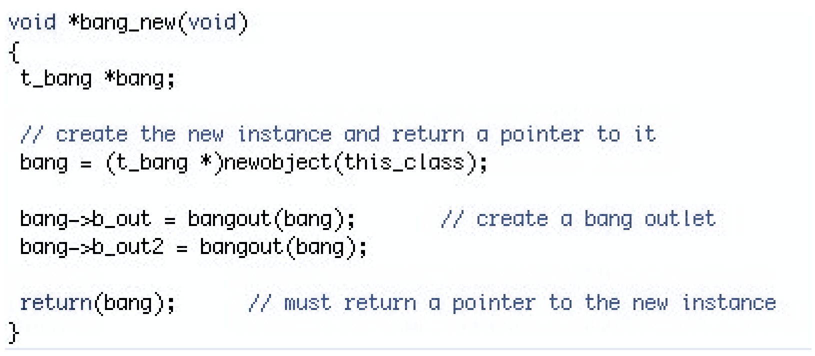
```
// Prototypes for methods: need a method for each incoming message
void *bang_new(void);           // object creation method
void bang_bang(t_bang *bang);   // method for bang message
```

Figure 4. Prototypes

Declaring the prototypes is our way of telling the compiler about the methods and appropriate arguments it will encounter before the method's actual definition. We could put our implementation above the `main()` function, but we'll follow the convention here of having the prototypes, followed by `main()`, following by the method implementation. So in `main()`, we reference both `bang_new()` and `bang_bang()`; however, these methods are **not** been declared in ext.h and are not native to C, so when we reference them in `main()`, they don't technically exist. (Your compiler may not care if you skip the prototype, but it's strongly recommended that you include prototypes.) The prototypes provide a nice way for us to see what our object has implemented, or at least what we want to implement and what our method signatures will be.

The prototypes outline (but do not define) the methods that tell Max *what to do*. This implementation is in contrast to our method references in `main()` that inform Max *where* to find our methods.

So our constructor, *bang_new(void)*, is a void pointer that takes a void as an argument. That method is defined as this:



```
void *bang_new(void)
{
 t_bang *bang;

 // create the new instance and return a pointer to it
 bang = (t_bang *)newobject(this_class);

 bang->b_out = bangout(bang);       // create a bang outlet
 bang->b_out2 = bangout(bang);

 return(bang);      // must return a pointer to the new instance
}
```

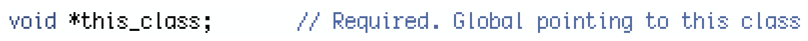
Figure 5. bang_new

Our constructor creates a pointer pointing to our typedef, calls `newobject`, sets up our outlets (`bangout`[15]) and returns.We'll let Zicarelli himself explain `newobject`:

> *You call* `newobject` *when creating an instance of your class in your creation function.* `newobject` *allocates the proper amount of memory for an object of your class and installs a pointer to your class in the object, so that it can respond with your class's methods if it receives a message.*

[Zicarelli, *Writing External Objects for Max 4.0 and MSP 2.0*, 35]

Our call to newobject refers to `this_class`. Earlier, we defined `this_class` as this:



```
void *this_class;       // Required. Global pointing to this class
```

Figure 6. this_class

In our constructor, `bang_new`, we're doing two things: we're creating an instance of our class, declared as `t_bang` (and sending it to `newobject()`), and creating pointers to our outlets. Our outlets send out bangs. They send bangs by calling `bangout()`.

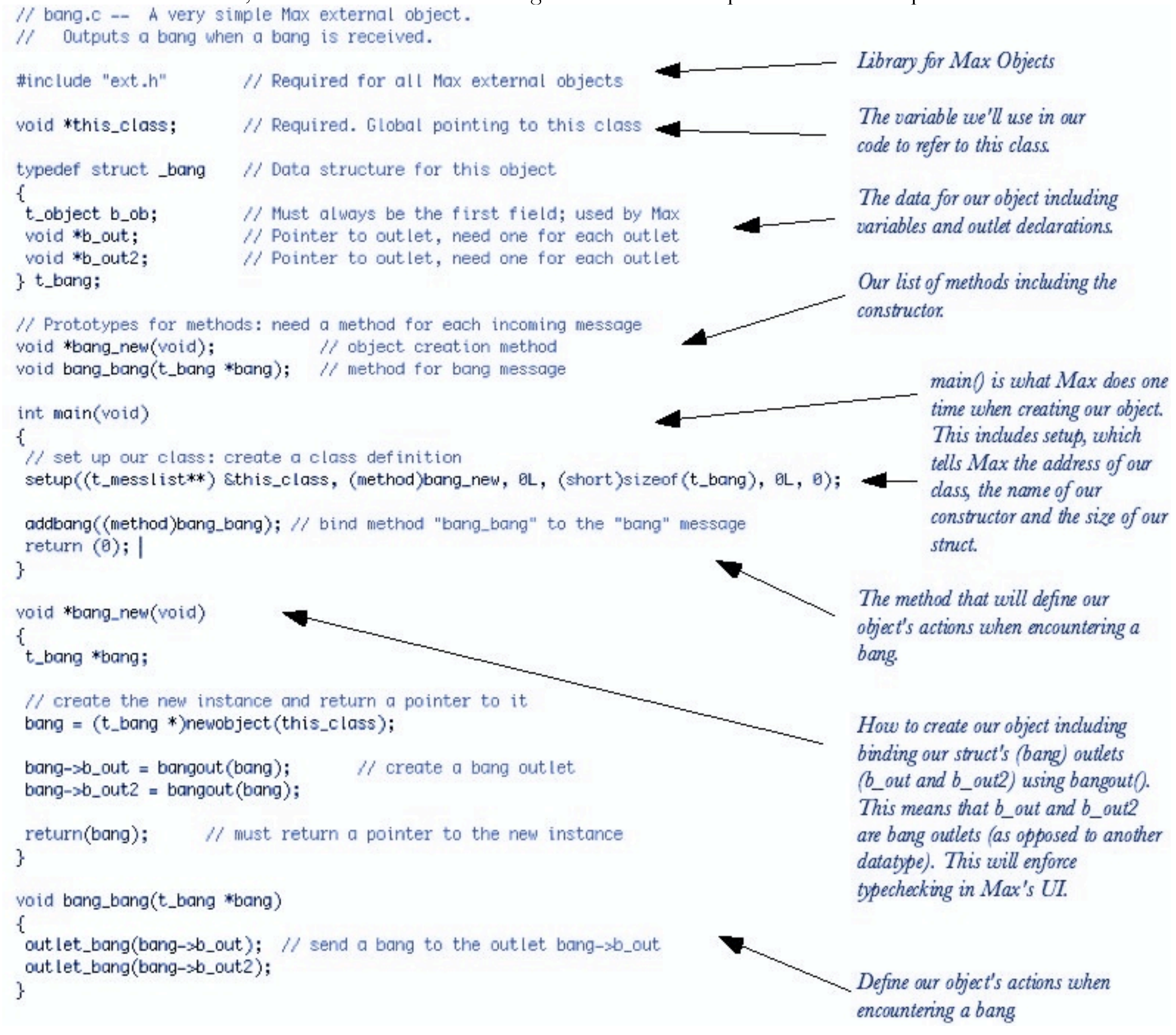
Our constructor is done. Lastly, `bang_bang` will take our struct as an argument and bind its outlets to `outlet_b`. `outlet_bang`, as the name implies, simply sends a bang out the bound outlets.

[15]There's also `floatout`, `intout` and `listout`.

4 Recap

That's a lot of stuff, especially if you're new to writing externs.. This does cover the framework for creating our extern. If your extern is going to do some sort of machinations to output some value based on a bang, the structure of your extern is not going to differ much from what you see here.

For the sake of review, look over the code below again with a brief explanation of each part.



```
// bang.c --  A very simple Max external object.
//   Outputs a bang when a bang is received.

#include "ext.h"        // Required for all Max external objects

void *this_class;       // Required. Global pointing to this class

typedef struct _bang    // Data structure for this object
{
 t_object b_ob;         // Must always be the first field; used by Max
 void *b_out;           // Pointer to outlet, need one for each outlet
 void *b_out2;          // Pointer to outlet, need one for each outlet
} t_bang;

// Prototypes for methods: need a method for each incoming message
void *bang_new(void);            // object creation method
void bang_bang(t_bang *bang);    // method for bang message

int main(void)
{
 // set up our class: create a class definition
 setup((t_messlist**) &this_class, (method)bang_new, 0L, (short)sizeof(t_bang), 0L, 0);

 addbang((method)bang_bang); // bind method "bang_bang" to the "bang" message
 return (0); |
}

void *bang_new(void)
{
 t_bang *bang;

 // create the new instance and return a pointer to it
 bang = (t_bang *)newobject(this_class);

 bang->b_out = bangout(bang);       // create a bang outlet
 bang->b_out2 = bangout(bang);

 return(bang);      // must return a pointer to the new instance
}

void bang_bang(t_bang *bang)
{
 outlet_bang(bang->b_out);  // send a bang to the outlet bang->b_out
 outlet_bang(bang->b_out2);
}
```

Library for Max Objects

The variable we'll use in our code to refer to this class.

The data for our object including variables and outlet declarations.

Our list of methods including the constructor.

main() is what Max does one time when creating our object. This includes setup, which tells Max the address of our class, the name of our constructor and the size of our struct.

The method that will define our object's actions when encountering a bang.

How to create our object including binding our struct's (bang) outlets (b_out and b_out2) using bangout(). This means that b_out and b_out2 are bang outlets (as opposed to another datatype). This will enforce typechecking in Max's UI.

Define our object's actions when encountering a bang

5 Going Forward

The functionality we've provided with our extern is certainly nothing you need an extern to do. We've received a bang and output a bang. If you can put this code in your favorite editor, compile it and deploy it, you're well on your way to writing more complex externs. The goal here is to show the structure of an extern and to better understand what's happening. There are tradeoffs in choosing any language – in terms of creating externs for Max, we have many choices and C is one of them. There are a variety of externs available at the Cycling '74 website. A quick survey of those externs will show the breadth of what is possible and may give you ideas for some great extern of your own.

Acknowledgements

Howard Sandroff, professor – Music, University of Chicago

Mark Shacklette, adjunct professor – CSSP, University of Chicago

References

Dobrian, Chris. Max: Tutorials and Topics. San Francisco: Cycling '74, 2000. Cycling '74. 31 Dec. 2005 <http://www.cycling74.com/twiki/bin/view/ProductDocumentation>.

Fujinaga, Ichiro. "Advanced Multimedia Development." Www.music.mcgill.ca. McGill University. 31 Dec. 2005 <http://www.music.mcgill.ca/~ich/classes/mumt402_05/mumt402outline.html>.

Fujinaga, Ichiro. "Advanced Multimedia Development." www.music.mcgill.ca. McGill University. 31 Dec. 2005 <http://www.music.mcgill.ca/~ich/classes/mumt402_05/mumt402Week0.ppt_files/mumt402Week0.ppt.ppt>.

Fujinaga, Ichiro. "Advanced Multimedia Development." www.music.mcgill.ca. McGill University. 31 Dec. 2005 <http://www.music.mcgill.ca/~ich/classes/mumt402_05/mumt402Week0.ppt_files/mumt402Week1.ppt.ppt>.

Zicarelli, David. MAX: Writing External Objects for Max and MSP. San Francisco: Cycling '74, 2005. Max/MSP Software Development Kit (SDK) for Macintosh. 31 Dec. 2005 <http://www.cycling74.com/twiki/bin/view/ProductDocumentation>.

Zicarelli, David. Writing External Objects for Max 4.0 and MSP 2.0. San Francisco: Cycling '74, 2001. Max/MSP Software Development Kit (SDK) for Macintosh. 31 Dec. 2005 <http://www.cycling74.com/twiki/bin/view/ProductDocumentation>.