# Isopod Structured Objects

# (ISO)

## Overview

ISO is a fundamental component of the isopod engine. It defines a common way to load, store, inspect, and use data.

Natively the ISO system defines three file formats – a textual (and editable) data description (with a filename extension of **ix**), a binary format (with extension **ib**), and a zlib-compressed binary format (with extension **ibz**). The binary formats can be loaded as a single block, or embedded in executables, and patched in-place for immediate access. They may also contain a raw binary section for uploading to specific hardware-accessible locations.

In addition, handling of other file types is managed by subclassing **iso::FileHandler**. Handlers are chosen based on the extension of the file to be loaded. The handlers read the file and return the data in an ISO pointer. The same classes are usually able to reverse the process – taking ISO data and writing it to the format they support.

These are some of the features of the system:

#### Reference counting

All referenced data is referenced-counted, and will be automatically deleted when the last strong reference is removed. In addition, weak references are supported.

#### Data inspection

The data can be inspected and its type extracted during runtime.

#### Direct Data access

The data layout is completely compatible with C types, and can be directly accessed through C pointers. Data can be packed into binary files, and after loading, can be accessed in the same way.

#### Named References

All referenced data (effectively, pointers in C) can contain an id, stored as either a crc32 or a string. Data can be browsed based on these ids.

#### Type Definition

The type system supports:

Arbitrary sized integer and floating point values

* Strings
* Sized and unsized arrays
* Arbitrarily complex composite structures
* Named types
* Virtual types that are implemented with callbacks

#### Conversion

Data can be automatically converted from one form to another. This can be used at runtime to retain compatibility with older binary files, or during export/conditioning to manipulate the data or create platform-specific variants.

#### Platform dependent block

Binary files can contain an unstructured data block for platform-specific raw assets.

#### File Dependencies

Dependencies between files can be output to help automate the build process.

#### Duplicate Removal

Any duplicated data appearing in a single file can be automatically detected and combined. In addition, cross-platform data duplication can be extracted into a common file.

#### Cross-file References

Data can be referenced from one file to another, and patched automatically at load-time.

#### Patching

A similar mechanism allows small delta-files to be generated that are applied to existing data. This allows for small update files for title updates.

## Types

A type in ISO is represented by a **ISO\_type** structure, which specifies the type class, along with any additional information a particular type class requires.

There are ten type classes:

|  |  |
| --- | --- |
| **ISO\_UNKNOWN** | **A null type to indicate unknown data.** |
| **ISO\_INT** | **An integer of specified bit-depth, or a fixed point value with a bit-depth and fixed-point position.** |
| **ISO\_FLOAT** | **An IEEE-754 like floating point value, with a specified mantissa and exponent size.** |
| **ISO\_STRING** | **A pointer to a zero-terminated string of 8-bit characters.** |
| **ISO\_COMPOSITE** | **A fixed collection of sub-types, where each element can be named, and is at fixed offset from the start of the collection.** |
| **ISO\_ARRAY** | **A fixed-size array of a specific sub-type.** |
| **ISO\_OPENARRAY** | **An array of a specific sub-type but an arbitrary length (stored with the instance of the data.)** |
| **ISO\_REFERENCE** | **A pointer to another piece of data, of a specified sub-type.** |
| **ISO\_VIRTUAL** | **Types which are handled via a callback mechanism.** |
| **ISO\_USER** | **A named type – with an underlying representation given by its sub-type.** |

With the exception of ISO\_UNKNOWN and ISO\_STRING, each of these type-classes represents many distinct types. For example, the type of an 8-bit integer and a 16-bit integer are separate, but both belong to the ISO\_INT class.

### Obtaining an ISO\_type

The ISO\_type of a type T in C++ can be obtained via the templated functions ISO\_getdef<T>() or ISO\_getdef(const T &t). These rely on an implementation of ISO\_def<T> being available. The following are always available:

|  |  |
| --- | --- |
| **class** | **C++ types** |
| **ISO\_UNKNOWN** | **void** |
| **ISO\_INT** | **hexint<T>**  **bool, int8, uint8, int16, uint16, int32, uint32, int64, uint64** |
| **ISO\_FLOAT** | **float, double** |
| **ISO\_STRING** | **const char\*, string, tag** |
| **ISO\_COMPOSITE** | **WRAP<T>**  **pair<A,B>**  **tuple<T>** |
| **ISO\_ARRAY** | **T[N]**  **fixed\_array<T,N>**  **ISO\_array<T,N>** |
| **ISO\_OPENARRAY** | **ISO\_openarray<T>** |
| **ISO\_REFERENCE** | **ISO\_ptr<T>**  **T\*** |
| **ISO\_VIRTUAL** | **dynamic\_array<T>**  **dynamic\_array<ISO\_ptr<T> >**  **WRAP\_PTR<T>** |
| **ISO\_USER** | **-** |

Due to the lack of type reflection in C++, the ISO\_type of general structures cannot be generated automatically through templates alone. There are a number of ways to handle this situation.

Assume we have the following type Example in a file example.h.

struct Example {

int an\_int;

float a\_float;

};

1. We can specialize the **ISO\_def** template for our type. A number of macros aim to make this a less error-prone and tedious exercise, but is still quite cumbersome:  
   Here, we define **ISO\_def<Example>** by inheriting from **CISO\_type\_user\_comp** – a convenience class for generating **ISO\_types** that are **ISO\_USER** types which reference **ISO\_COMPOSITE** types.  
   In an **ISO\_COMPOSITE** type, the elements that comprise the type must appear immediately after the type header, and so we declare **CISO\_elements** with the same names as actual structure fields at the start of the class. These are then initialized in the constructor using the **ISO\_ELEMENT** macro. This ensures that the fields exactly correspond to the naming and offsets of the structure.  
   The constructor is also responsible for initializing the name of this type (“Example”), and specifying the number of elements in the structure.

template<> struct ISO\_def<Example> : public CISO\_type\_user\_comp {

CISO\_element an\_int, a\_float;

ISO\_def() : CISO\_type\_user\_comp("Example", 2),

ISO\_ELEMENT(Example, an\_int),

ISO\_ELEMENT(Example, a\_float)

{}

};

1. This template specialization can be generated from isocmd using the original C++ header file as input.  
   **isocmd example.h example\_iso.h isodefs=1**
2. The type could instead be defined in the ix scripting language for direct access from ix based data files. From the ih format, both the C structure and the template specialization can be obtained using:  
   **isocmd example.ih example.h isodefs=3**In this case, the corresponding example.ih file would look like this:  
     
   define {  
    int an\_int  
    float a\_float  
   } Example
3. In many cases it is not necessary to completely specify the ISO\_type for the run-time code. The types are stored within the binary files, and so the data will be browseable regardless. A trivial stub may be sufficient to acknowledge the handling of a particular type:  
   **ISO\_DEFUSER(“Example”,void);**

### Virtual Types

The standard type classes have built-in ways in which they can be browsed and queried. The type class **ISO\_VIRTUAL** allows for arbitrary mechanisms to traverse data, including the possibility to generate or read data as it is requested.

Virtual types contain a table of six function pointers to perform basic operations on data that is of their type. This table is usually set up automatically using the TISO\_virtual template. Each function takes a pointer to the instance-data as the first parameter, though the TISO\_virtual template actually reinterprets the pointer as the **this** parameter to a member function of the data type.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Parameters** | **Returns** | **Function** |
| **Count** | **-** | **size\_t** | Counts the number of elements within this instance. |
| **GetName** | **int index** | **tag2** | Gets the name of the nth element. |
| **Find** | **tag2 &id** | **int** | Finds the index of the element corresponding to the id, or -1 if none. |
| **Index** | **int index** | **ISO\_browser2** | Indexes into the data instance. |
| **Delete** | **-** | **void** | Deletes the data instance. |
| **Update** | **const char \*spec** | **bool** | Updates the data… |

Default implementations of these functions can be inherited through the class **ISO\_virtual\_defaults**.

## ISO\_ptr

**ISO\_ptr** is a templated class that represents a reference to a known type. When the type is unknown, a template parameter of void can be used.

An **ISO\_ptr** can be freely converted to, and treated as, a normal C pointer; it is stored in the same space as a pointer. In addition it allows access to the header data prepended to the actual data, and is the mechanism by which the reference counting is maintained.

Creating ISO data is always carried out through an **ISO\_ptr** – either through some of the **ISO\_ptr** constructors (when an ID is supplied, a new instance of the data is created), or by using one of the **Create** member functions.

### Implementation

An **ISO\_ptr** is a wrapper around a normal C pointer, and overloads operators to allow the **ISO\_ptr** to behave as much as possible like a normal pointer.

The pointer, however, actually points 16 bytes past the actual allocated storage of the data it is referencing. These 16 bytes hold header data which includes the id and type of the data, reference counting information, various flag attributes and a 4-byte user field. The header data is defined by the **ISO\_value** type:

struct ISO\_value {

tag id;

const ISO\_type \*type;

void \*user;

uint16 flags;

uint16 refs;

};

The flags field contains the following attributes:

|  |  |
| --- | --- |
| **Flag** | **Meaning** |
| CRCID | The id contains a CRC, not a string. |
| CRCTYPE | The type field is actually just the CRC of a named type. |
| EXTERNAL | The data is a (unpatched) reference into another file |
| REDIRECT | A temporary flag used during patching |
| HASEXTERNAL | Indicates that there is an external reference somewhere further down within the data. |
| SPECIFIC | Indicates that a platform-specific asset has been substituted for a named file. |
| ISBIGENDIAN | The data is big-endian (if not set, it is little-endian). |
| ROOT | This pointer is at the root of a binary file. |
| ALWAYSMERGE | Allow merging of duplicated data, even if the id don’t match |
| WEAKREFS | Indicates the data has some weak references. If set, when the last strong reference has gone, the pointer is found in the weak-reference table and cleared. |
| EXTREF | Indicates a patched external reference. |
| FROMBIN | Indicates the data is embedded in a binary file. |
| NATIVE | Indicates native endian – so is defined to be the same as ISBIGENDIAN or 0, depending on the platform. |

### Weak references

**ISO\_ptr** supports weak references through the structure **ISO\_weak**.

The **ISO\_weak** information cannot be accessed directly from the header, and must be looked up in a table when needed. Luckily there are only two places where this has to happen.

1. If a new weak reference is being created from an **ISO\_ptr**, and
2. When the last strong reference has been removed.

In all other cases, the **ISO\_weak** is either directly available, or not needed at all.

When the last strong reference is removed, all weak references to the data will subsequently return a null-pointer.

When using weak references, the **ISO\_weak**’s can be managed locally, or be allowed to be automatically created and destroyed. It is important that only one copy of the **ISO\_weak** exist for any one pointer, and so the locally managed variety can only be used when there is no possibility of an **ISO\_weak** already existing.

The templated type **ISO\_weakref** is a wrapper around **ISO\_weak** for direct memory management. The cost of referencing data through this type should be the same as for a normal **ISO\_ptr** (which should be the same as for a normal C pointer).

When the **ISO\_weak** memory management is handled automatically, the template type **ISO\_weakptr** is used. This wraps a pointer to an **ISO\_weak**, and so incurs an extra level of indirection when used.

### External references

Some **ISO\_ptr**’s do not directly reference their data, but are in fact descriptions of locations within separate files where the data can be located.

Before dereferencing, these **ISO\_ptr**’s must be patched by first loading the external file, and finding the correct location within it. Subsequently the **ISO\_ptr** can be used as any other, and the reference into another file is invisible to the application.

The external reference is noted again when the last strong reference to the data is removed, and the reference counts in the external data are adjusted.

### IDs: tag, tag2

All referenced data is named using an id, though many will leave the id blank. It is not required for the ids to be unique, or to contain any specific naming or subset of characters.

The ids can be stored as either a character string or a CRC of such a string. The **tag** type is used to contain the id within an **ISO\_value**. On its own this type doesn’t have enough information to determine whether it is a CRC or not – that information is held in a flag in the **ISO\_value** type.

For run-time access, the **tag2** type can hold both a CRC and a character string, and it will automatically generate the CRC from a character string the first time it is needed.

When two **tag2**’s are compared, if either one does not have access to the original character string, they are compared via their CRCs – generating whichever are necessary.

## ISO\_array

**ISO\_array** is a templated class that represents an array of a known type and size. It is included primarily for completeness and the usual C array syntax can be used instead.

## ISO\_openarray

**ISO\_openarray** is a templated class that represents an array of a known type, but arbitrary size. During content creation functions the array can be resized, though at run-time it is expected that the size will be fixed but queriable.

An **ISO\_openarray** of **ISO\_ptr**’s can be searched by id.

## ISO\_browser/ISO\_browser2

ISO\_browser is the general mechanism through which ISO data can be browsed. An ISO\_browser can be initialized to refer to an arbitrary type of data, and using its browsing functions all data accessible from the starting point can be browsed to.

In addition, ISO\_browser2 modifies the reference counts of ISO\_ptr data as it encounters it, in order to prevent temporarily browsed data from disappearing before the final data has been reached.

## Utilities

A few utility functions are included with the base ISO library for general manipulation of ISO data.

### Duplicate

Creates a duplicates copy of the data. Optionally able to create a deep copy by recursive calls of the Duplicate function.

### Compare

Compares two copies of data.

### CheckHasExternals

Traverses through browseable data to verify that external data exists.

### Endian

Traverses through browseable data, making it either all big- or little-endian.

## Tools

### IsoCmd

General command line tool for data ‘conditioning’ ready for platform deployment.

### IsoMAX

A 3DS Max plugin allowing the export of ISO data from a MAX scene, and the attaching of entities to MAX scene nodes.

### IsoEditor

General GUI tool for examining ISO data – including any data that can be read or generated in that format.

### IsoCRC

A suite of tools to manage a database of CRC’s for reverse lookup during debugging and data inspection.

### IsoLink

An umbrella term for communication between a running console and a tool that handles ISO data.