Library of Practical Abstractions

Version 1.2

Eric Sven Ristad and Peter N. Yianilos

1 Overview

The library of practical abstractions (LIBPA) provides efficient implementations of conceptually simple abstractions, in the C programming language. We believe that the best library code is conceptually simple so that it will be easily understood by the application programmer; parameterized by type so that it enjoys wide applicability; and at least as efficient as a straightforward special-purpose implementation. You will find that our software satisfies the highest standards of software design, implementation, testing, and benchmarking.

The current LIBPA release is a source code distribution only. It consists of modules for portable memory management, one dimensional arrays of arbitrary types, compact symbol tables, hash tables for arbitrary types, a trie module for length-delimited strings over arbitrary alphabets, single precision floating point numbers with extended exponents, and logarithmic representations of probability values using either fixed or floating point numbers.

We have used LIBPA to implement a wide range of statistical models for both continuous and discrete domains. The time and space efficiency of LIBPA has allowed us to build larger statistical models than previously reported, and to investigate more computationally-intensive techniques than previously possible. We have found LIBPA to be indispensable in our own research, and hope that you will find it useful in yours. If you find LIBPA useful, please let us know!

LIBPA is not backward compatible. We strive to provide the best libraries for each release, and do not hesitate to completely redesign a module to improve clarity or performance. As a result, upgrading your LIBPA installation to the latest release may require you to rewrite portions of your code. We will not change the semantics of a function without also changing the prototype for that function. This policy should help you quickly upgrade your software using the latest LIBPA release.

1.1 Installation

1.1.1 Licensing

Every module has it's own license, written by the module's authors. Before you install or use a LIBPA module, you must agree to the terms of that module's license. Typically, authors will ask that you use LIBPA for not-for-profit research purposes only, and that you follow standard academic practice in acknowledging your use of LIBPA in any relevant reports or publications.

1.1.2 Getting the Source Code

Source code for entire library may be obtained from

```
ftp://ftp.cs.princeton.edu/pub/packages/libpa/
```

Please use *GNU* `tar -cf` to unpack the '`.tar.gz`' file.

1.1.3 Environment Variables

Once you have ftped and unpacked the distribution, you must initialize the following environment variables.

`LIBPA_GROUP`
: group with read/write permissions in LIBPA installation

`LIBPA_ARCH`
: current machine architecture (cpu/OS combination)

`SCC` safe compiler to produce debuggable object code

`SLD` safe loader to produce debuggable binaries

`UCC` unsafe compiler to produce optimized object code

`ULD` unsafe loader to produce optimized binaries

`RANLIB` ranlib

`MAKE` make or gmake

The `LIBPA_GROUP` environment variable should be set to the name of the group which will have read/write permissions in the installation. We source the '`libpa-1.2/bin/cshrc`' shell script on our system to set these environment variables properly.[1]

1.1.4 Creating Object Code

After you have set these environment variables, run *`make install`* from '`libpa-1.2/src`' to install the libraries, and then run *`make tests`* from '`libpa-1.2/src`' in order to verify our implementations in your operating environment. When you are satisfied that in the outcome of these tests, run *`make clean`* to delete extraneous object code files and archives in the source directories.

As explained below, we provide safe and unsafe versions of all object code. The safe versions contain unoptimized object code with symbol tables for debugging purposes. The unsafe versions contain optimized object code without symbol tables. We recommend using safe object code on a routine basis, and only using unsafe object code when performance is of utmost importance. The '`libpa.a`' and '`libpa_u.a`' archives contain the safe and unsafe versions of all object code, respectively.

1.1.5 Supported Environments

All modules are supported on the following operating environments:

alpha Digital Alpha with Digital UNIX 4.0

i586 Intel Pentium with Linux 2.0

[1] Note that we do not recommend using gcc to compile the modules because gcc has a proprietary implementation of the `assert()` macro. Therefore, once you compile a module using gcc, all executables based on that module must link libgcc.a. This is particularly annoying if you are trying to track down a compiler bug.

sun5 Sun SPARC with SunOS 5.5 (Solaris)

Most modules support a wider range of UNIX operating environments, including

hppa HP PA-RISC 9000/7xx with HP-UX 9.0

sgi SGI MIPS with IRIX 5.x or 6.x

sun4 Sun SPARC with SunOS 4.3.1

1.2 Mailing Lists

1.2.1 Staying Informed

We encourage you to subscribe to libpa-announce@cs.princeton.edu, to learn about upgrades and new modules as they are published. Send mail to majordomo@cs.princeton.edu with the single line:

```
subscribe libpa-announce
```

1.2.2 Getting and Giving Help

If you are having trouble installing or using LIBPA, or have other questions about the library of practical abstractions, send mail to libpa-help@cs.princeton.edu. If you are a skilled user of LIBPA, then we encourage you to subscribe to this majordomo mailing list and to help field the questions that are posted.

1.2.3 Bug Reports

We are unable to track down bugs in your code or in your operating environment (compiler, operating system). We have spent a significant amount of time and care writing the LIBPA modules, and have published the source code and test suites. So if you think you have found a bug in our code, please help us by sending a thoroughly tested bug fix along with your bug report. When you have finished testing your bug fix, we would be grateful if you would please send it to libpa-bugs@cs.princeton.edu along with a detailed explanation.

1.3 Acknowledgments

1.3.1 Affiliations

Eric Sven Ristad is with the Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544. He is partially supported by Young Investigator Award IRI-0258517 from the National Science Foundation. Email: ristad@cs.princeton.edu.

Peter N. Yianilos is with the Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544 and with the NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. Email: pny@research.nj.nec.com.

1.3.2 Thanks

We are grateful to David Hanson <drh@microsoft.com> for his feedback, and for providing us with so many well-taught students while he was with us at Princeton. Robert Thomas <rgt@cs.princeton.edu> provided invaluable input in the design and redesign and reredesign of the LIBPA modules. Kirk Kanzelberger <kirk@research.nj.nec.com> assisted with the initial implementation of the `vector_t` module in Summer 1993.

Documentation was created in GNU texinfo format, from which info and postscript formats were derived directly. html format was derived using texi2html from Lionel Cons.

2 Library Modules

Currently we publish modules for basic memory management, tables of arbitrary types, and extremal numerical values:

`memory_t` efficient portable memory management

`vector_t` one dimensional arrays of arbitrary types

`table_t` compact symbol table for <key,datum> pairs.

`trie_t` trie for mapping strings to integers

`hash_table_t`
hash table toolkit, to build your own high-performance hash tables.

`symbol_table_t`
hash table for symbols

`string_table_t`
hash table for NULL-terminated character strings

`unigram_t`
compact table of symbol frequencies with dynamic counters

`balanced_t`
single precision floating point numbers with extended exponents

`L_t` fixed point logarithmic representation of probability values

`logpr_t` floating point logarithmic representation of probability values

`pr_t` generic interface to probability value libraries

2.1 Memory Management

The `memory_t` module is the foundation of our library. All memory management is performed in a portable manner using the `memory_t` module, which helps identify memory leaks without using expensive and nonportable third party applications, such as purify. The `memory_create()` function replaces `malloc()`, while the `memory_destroy()` function replaces `free()`. Other functions are provided to replace the standard C memory functions.

The `vector_t` module is also widely used, and is equally indispensable. A `vector_t` is a length-delimited sequence of objects of arbitrary type. The objects are parameterized by the size of their representations, via the `sizeof()` primitive. The `vector_t` module supports dynamic resizing, concatenation, insertion, sorting, and portable input/output. Indeed, we often use `vector_t` where LISP programmers would use lists. Since the `vector_t` objects aren't boxed and are arranged linearly in memory, `vector_t` operations are at least twice as fast as lists and require significantly less storage than lists (eg., up to 16 times less storage for one byte objects on a machine with 8 byte pointers).

2.2 Tables

LIBPA provides four ways to store information in tables: `table_t`, `trie_t`, `hash_table_t`, and `unigram_t`.

The `table_t` module is a compact table for arbitrary <symbol,datum> pairs. Lookups are performed in O(log n) using binary search, while insertions and deletions are O(n) for a table containing n entries. The `table_t` module is typically used as a building block for other modules, such as the `trie_t` module, where memory usage is the single most important performance desiderata.

The `trie_t` module maps length-delimited strings over arbitrary alphabets to consecutive unsigned integers. It also provides the inverse map, from indices to length-delimited strings.

The `hash_table_t` module supports the easy construction of compact hash tables for arbitrary <key,datum> pairs with open addressing, double hashing, and dynamic resizing. We used the `hash_table_t` module to implement two useful modules: the `symbol_hash_t` for arbitrary symbols and the `string_hash_t` for NULL-terminated character strings. Again, we strove to minimize the space requirements of the implementation so that large numbers of `hash_table_t` can be used in even the most memory-intensive computations.

Finally, the `unigram_t` provides a space-efficient representation for the frequencies of symbols drawn from an arbitrary alphabet. Our `unigram_t` implementation uses dynamic counter resizing to minimize the amount of storage required to store the symbol frequencies. Thus, it may be used to efficiently store the state transition frequencies in a very large Markov model.

2.3 Numerical Values

LIBPA provides three ways to represent extremely small numerical values, such as are commonly encountered in statistical modeling applications: `balanced_t`, `L_t`, and `logpr_t`. The `pr_t` module provides a generic macro interface to all three ways of representing probability values. The three modules offer a wide range of performance/accuracy tradeoffs. `L_t` is the fastest and requires the least space. `logpr_t` is the most accurate but the slowest. `balanced_t` is nearly as accurate as `logpr_t` and can be significantly faster.

The `balanced_t` type represents single precision floating point numbers with extended 32-bit exponents. As a result, the `balanced_t` module supports floating point arithmetic for extremely small and extremely large numerical values – values which would cause overflow or underflow in double precision floating point arithmetic. Depending on your machine, `balanced_t` arithmetic is from 10 to 20 times slower than double precision floating point arithmetic and is only slightly less accurate that single precision floating point arithmetic. Unlike the other LIBPA numerical types, the `balanced_t` type is capable of representing a much larger range of numbers, such as very large numbers and negative numbers.

The `L_t` type represents probability values using fixed point numbers — `char`, `short`, `int`, and `long` — where larger types are capable of representing ever-smaller probability values. Depending on your machine, `L_t` arithmetic is from 1.2 to 1.8 times slower than double precision floating point arithmetic. It is also considerably less accurate. `balanced_t` arithmetic is more accurate than `L_t` arithmetic, but is more than five times slower and typically requires twice as much memory.

Finally, we also provide a standard `logpr_t` type for the logarithmic representation of probability values using double-precision floating point numbers. Although this is the most accurate representation of probability values, it is also the slowest. Depending on the math libraries provided by your operating system, `logpr_t` arithmetic can be 13 to 60 times slower than double precision floating point arithmetic.

The following table reports timing results of the '`pr.bench`' benchmark on four different machines. All numbers are user times in seconds, as reported by '`/usr/bin/time`' or '`/usr/bin/timex`' (lower is better).

	double	logpr_t	balanced_t	L_t
AlphaStation 500/500	121.8	1619.8	1292.7	207.2
Sun Ultra2 1200	301.2	10166.4	5037.2	541.9
SGI O2 180mhz	371.4	8570.8	4830.1	546.6
HP 9000/755	369.2	19643.6	7497.1	638.0
Dell GXM 5166	805.5	12010.6	5118.6	933.3

3 Library Organization

3.1 Directory Structure

The library of practical abstractions is organized as follows:

‘libpa-1.2/include’
: include files

‘libpa-1.2/lib.*arch*’
: object code archives for *arch*

‘libpa-1.2/src/*module*’
: source code for *module*

‘libpa-1.2/bin’
: shell scripts

‘libpa-1.2/bin/*arch*’
: architecture-dependent executables for *arch*

‘libpa-1.2/doc’
: miscellaneous documentation

This allows you to compile with -I‘libpa-1.2/include’ and link with -L‘libpa-1.2/lib.*ARCH*’. You may also want to put ‘libpa-1.2/bin’ and ‘libpa-1.2/bin/*ARCH*’ on your PATH environment variable.

3.2 File Naming Conventions

A LIBPA module named *module* is implemented in the following files:

‘*module*.h’
: public prototypes and typedefs for *module*

‘*module*.c’
: source code for *module*

‘*module*.test.c’
: source code for *module* test suite

‘*module*.test’
: safe executable for *module* test suite

‘*module*.test_u’
: unsafe executable for *module* test suite

‘Makefile’
: Makefile for *module*

‘license’
: Copyright notice, license, and disclaimer

The implementation may include other files, named according to a similar naming convention, such as:

'*module*.bench.c'
: source code for *module* comparative benchmark

'*module*.bench'
: executable for *module* comparative benchmark

'*module*.pure'
: purified test suite for *module* (safe)

'*module*.pure_u'
: purified test suite for *module* (unsafe)

Special module functions may be provided a separate interface. For example, the `memory_t` library includes a '`memory.spartan.h`' header file for the special-purpose spartan `memory_t` functions. As explained below, the object code for *module* is placed in an archive named lib*module*.a in the '`libpa-1.2/lib.`*ARCH*' directory.

3.3 Test Suites (required)

Comprehensive test suites are provided for all code. A test suite is a certificate of correctness. It must convince an aggressive skeptic of the correctness of the implementation. The best way to establish correctness is to verify the behavior of the implementation against an independent implementation. The independent implementation should be so simple that it is plausibly correct. Even the weakest test suite must exercise all functions in reasonable as well as unreasonable operating ranges.

A test suite should run to completion in a reasonable amount of time. It should describe the tests being performed as they are executed, but should not display too much information either, rarely more than a page. It should end by announcing the successful completion of the test or by dumping core via an `abort()` call or an `assert()` failure.

3.4 Benchmarks (optional)

A benchmark is a comparative performance analysis. The goal of a benchmark is to reveal the performance tradeoffs made in your implementation. Accordingly, the ideal benchmark compares the actual time/space requirements of your implementation with a range of alternative implementations in a realistic application. In practice, a benchmark should compare your implementation to at least one alternative implementation in a realistic sequence of function calls.

3.5 Object Code and Archives

The library of practical abstractions provides two versions of all object code: safe and unsafe. We recommend the routine use of safe object code. Safe object code is intended for everyday use. Safe object code has been compiled with no optimizations, with all `assert()` macros, and with full symbol tables for debugging. Safe libraries are named '`lib`*module*`.a`'. The '`libpa.a`' archive includes all safe object code in the current release.

Unsafe object code is intended for the most demanding applications. Unsafe object code has been compiled with maximum optimizations, the the *NDEBUG* macro defined, and without any symbol tables. We strongly discourage the routine use of unsafe object code, which is why we gave it such a frightful name. Under no conditions should unsafe object code be used during debugging, because optimizations are a major source of compiler errors and erroneous debugger information as well. Unsafe libraries are named '`lib`*module*`_u.a`'. The '`libpa_u.a`' archive includes all unsafe object code in the current release.

3.6 Makefiles

Our Makefiles include the targets "install", "tests", and "clean". Use '`libpa-1.2/src/Makefile`' to perform these operations on all modules.

`make install`

Installs the header files in '`libpa-1.2/include`' and then installs the safe and unsafe object code archives in '`libpa-1.2/lib.`*ARCH*'.

`make tests`

Creates and runs safe and unsafe versions of the current module's test suite.

`make clean`

Deletes all object code and archives in the current directory.

4 Programming Style

The central goal of module implementation is correctness. Performance is important, but secondary to correctness. By correctness, we mean that all specified functions are implemented strictly according to the module specification, without any memory leaks or segmentation faults. Aside trivial cases, there will be no proof of correctness, only degrees of belief in correctness. And so the astute reader has already realized that "correctness" really means "belief in correctness".

Our belief in the correctness of an implementation is principally influenced by clarity of the implementation. It is much easier to develop confidence in code that is easy to understand than in code that is difficult to understand. Clarity is determined by degree of conformance to a style guide, appropriate use of abstractions, appropriate symbol names (that obey reasonable naming conventions), useful but sparing comments (when essential to understanding), and consistent formatting.

As we hope you will see, our software places an unusual premium on clarity, both directly and indirectly. Our belief in the correctness of an implementation is also affected by the rigor of the programmer's test suite and the abundance of assertions. Accordingly, our software always includes a comprehensive test suite as well as aggressive use of assertions.

4.1 Desiderata

We evaluate our code based on the three criteria of correctness, clarity, and performance.

- Correctness: Does the code correctly implement the specification?
  - all specified functions implemented strictly correctly
  - abundant use of `assert()` for correctness and clarity
  - a comprehensive test suite is provided by the programmer
  - no memory leaks or segmentation faults
- Clarity: How easy is it to understand the code?
  - conceptually elegant specification
  - appropriate use of abstractions
  - degree of conformance to style guide
  - appropriate variable names, that obey naming conventions
  - useful comments when essential to understanding
  - proper formatting
- Performance: How well does the code perform (time, space)?
  - speed of the code in real-world use
  - efficiency of memory management
  - performance relative to other implementations

4.2 Identifier Naming Conventions

Clarity is of the utmost importance in naming your identifiers. Your code should read like a great novel. Replace generic names, such as `cnt` or `count`, with maximally specific names, such as `vertex_count` or `edge_count`. Refrain from using meaningless names, such as `temp`, `tmp`, or variants of the dreaded `xxx`. Single character names are reserved for integer iteration variables.

Vowels are obligatory! Even standard abbreviations (eg., `defn`, `dir`) should not be used unless they substantially improve clarity. If your typing skills are inadequate to type long names quickly, then take a typing class, but don't burden others with your sloth.

4.2.1 Constants and Variables

Constants created using `#define` should be all upper case. Macros created using `#define` should be either initial letter upper case or all lower case (to allow replacement by procedures) when they take arguments. Global variables should be first character upper case. Procedures and all local variables should be all lower case.

Compound names are formed with underscore separators, eg., `CONSTANT`, `nifty_macro()`, `Global_Variable`, `local_variable`.

Don't reuse variable names. A single variable name should have a single meaning in the widest possible scope (minimally within the body of a procedure, preferably within an entire library). Iteration variables should always iterate over the same domain.

4.2.2 Pointers and Handles

When it improves clarity, variables that are pointers to a single object should be named with the `_p` suffix. Similarly, a pointer to a pointer to an object (also called a "handle") should be named with the `_pp` suffix when it improves clarity.

```
type_t    *name_p, **name_pp;
```

This convention might be used, for example, to distinguish a one dimensional array of pointers (`void **name_p` or `void *name_p[]`) from a two dimensional array of objects (`void **name` or `void name[][]`), or when an object is passed directly (ie., by value) and indirectly (ie., by reference) in the same piece of code. When an object is always passed by reference, adding the `_p` suffix does not improve clarity. Note that arrays are NOT conceptually considered pointers, and therefore they should not be named according to the `_p` or `_pp` suffix convention (unless the arrays are arrays of pointers or arrays of pointers to pointers, respectively).

4.2.3 Procedures

When possible, procedure names begin with their module name and their prototypes obey the following naming conventions (see module design).

4.2.3.1 Type Definitions

`typedef ...` *type*`_t`;
    typedef for objects of type *type*`_t`.

type`_tag` internal tag for union or struct typedef.

4.2.3.2 Constructors and Destructors

type`_t *`*type*`_create(...);`
: Creates a new object of type *type*`_t` and returns its address.

`void` *type*`_destroy(`*type*`_t *object);`
: Permanently destroys an object of type *type*`_t`; all subsequent operations on that object are undefined.

type`_t *`*type*`_copy(const` *type*`_t *object);`
: Creates an exact copy of an object of type *type*`_t` and returns its address.

`void` *type*`_initialize(`*type*`_t *address, ...);`
: Given the address of `sizeof(`*type*`_t)` bytes, initializes that memory location to contain a valid object of type *type*`_t`.

`void` *type*`_finalize(`*type*`_t *object);`
: Given a valid object of type *type*`_t`, finalizes that object so that the subsequent freeing of `sizeof(`*type*`_t)` bytes at the given address will completely destroy the given object.

4.2.3.3 Predicates, Selectors, and Mutators

`boolean_t` *type*`_is_`*predicate*`(const` *type*`_t *object);`
: Returns `TRUE` if and only if the Boolean *predicate* holds for a given object of type *type*`_t`.

`value` *type*`_`*attribute*`(const` *type*`_t *object);`
: Returns `value` of given *attribute* for given object of type *type*`_t`, or the address of that value.

`void` *type*`_set_`*attribute*`(`*type*`_t *object, ...);`
: Change an object of type *type*`_t` so that the desired *attribute* now holds for that object.

`void *`*type*`_workspace_p(const` *type*`_t *object);`
: Returns address of caller workspace in object of type *type*`_t`, which caller is allowed to modify.

4.2.3.4 Type Coercion

typeB`_t *`*typeA*`2`*typeB*`(const` *typeA*`_t *object);`
: Given an object of type *typeA*`_t`, creates an equivalent object of type *typeB*`_t` and returns it.

4.2.3.5 File Input/Output

`boolean_t` *type*`_write(const` *type*`_t *object, FILE *stream);`
: Writes objects of type *type*`_t` to the binary file stream in self-delimiting format and returns TRUE if and only if the write succeeds.

type`_t *`*type*`_read(FILE *stream);`
: Reads object of type *type*`_t` from the binary file stream in self-delimiting format and returns it.

`boolean_t` *type*`_fprintf(const` *type*`_t *object, FILE *stream);`
: Formats object of type *type*`_t` on ASCII file stream in self-delimiting human-readable format.

type`_t *`*type*`_fscanf(FILE *stream);`
: Reads object of type *type*`_t` from ASCII file stream in self-delimiting human-readable format and returns it.

`boolean_t` *type*`_backup(const` *type*`_t *object, FILE *stream);`
: Writes properties of an object of type *type*`_t` to the binary file stream in self-delimiting format and returns TRUE if and only if the write succeeds. This is used when

`boolean_t` *type*`_restore(`*type*`_t *object, FILE *stream);`
: Reads properties of an object of type *type*`_t` from the binary file stream in self-delimiting format and returns TRUE if and only if the read succeeds and the objects properties are updated from the file stream.

4.2.3.6 Argument Order

The first argument to an function should always be an object of the type defined in the function's module, eg.,

```
void   type_write(const type_t *, FILE *);
type_t *type_read(FILE *);
```

4.3 Safety and Clarity

Avoid macros! Macros are a major source of bugs. Use procedures instead. Once your code is working, and you find yourself with extra time on your hands, then you can replace some procedures with macros for minor performance gains, although we still do not recommend wasting your time this way.

All source code should make abundant use of the `assert()` macro from the '`<assert.h>`' library for both clarity and correctness. Correctness includes checking that procedures are passed the proper arguments and that they return proper values. Clarity includes asserting all known invariants and assumptions. As a rough guide, around 50% of your code should be asserts, as high as 75% when performing pointer arithmetic. No kidding. The source code must work properly even if `NDEBUG` is defined and all the `assert()` macros vanish. That means no side-effects may occur in the expressions passed to `assert()`.

Replace compound asserts with primitive asserts in order to make debugging easier. For example, the compound assertion

```
assert(exp1 && exp2);
```

should be replaced with the two primitive assertions

```
assert(exp1); assert(exp2);
```

Do not mutate a variable inside a complex expression or when it appears as an argument to a procedure call. If this restriction makes you whine about typing effort, it's time to take another typing class. Thus, `f(i++);` should be `f(i); i++`.

All blocks must be explicitly delimited by curly brackets, especially if the blocks contain exactly one statement, eg.,

```
if (predicate) { statement; }
```

Liberal use of parentheses is encouraged, particularly within macro definitions. Programming is not a test to see whether you know all the most compact legal expressions of C. Rather, it is a test to see if you can write correct code of breathtaking clarity.

Cutting and pasting code is likely to introduce errors. It is also an early warning sign of a bad modularization. So don't do it! If you are unable to break your cut-and-paste habit, then you must review newly pasted code with extraordinary diligence.

4.4 Type Declarations

Whenever possible, arguments passed by reference (ie., pointers) should also be declared const. Not only does this improve the clarity and safety of your code, but it allows other programmers to use your code. A procedure argument that is not declared const can never be const in any code that uses that procedure.

Do not mix declarations and initializations. That is, do not combine the type declaration of automatic variables with their initialization in the body of a procedure. For example, replace `int i=0;` with the declaration `int i;` in the preamble followed by the actual initialization `i=0;` in the body of the procedure.

Declare variables in the narrowest possible scope. If you only use a variable inside one of the branches of a conditional statement, then it should be declared only in that branch. This reduces the risk of improperly initialized variables and inadvertent name conflicts. It also makes the code easier to understand.

In general, typedefs should be to the object itself, rather than to a pointer to the object. This makes memory management easier in most cases, and improves clarity because the caller better understands the semantics of procedure arguments.

Objects requiring more space than two machine words should typically be passed by reference for the purposes of efficiency. However, objects that are used solely to return multiple values from a procedure call should typically be returned by value in order to simplify memory management.

We strongly encourage you to design modules that are parameterized by types. Type parameterization allows you to design efficient, reusable modules. In C, types are parameterized by the size of their objects (in bytes) along with a minimal set of procedures necessary to operate on those objects.

4.5 Additional Guidelines

4.5.1 Efficiency

No operating system calls, such as memory allocation or file i/o, should occur in time-critical inner loops. This may require design changes.

Minimize the number of pointers in your data structures. Each pointer reference requires a potentially-nonlocal memory reference, resulting in a cache miss, and can take up to 8 bytes of storage on some machines.

4.5.2 Memory Management

Even a small memory leak is unacceptable.

All LIBPA code must use the `memory_create()` and `memory_destroy()` procedures from the `memory_t` library instead of the `malloc()` and `free()` provided by the standard `stdlib`.

Every test suite must conclude with a memory leak check.

```
assert(memory_total_bytes() == 0);
assert(memory_total_blocks() == 0);
```

4.5.3 Formatting

Use GNU emacs C mode. In most cases, curly braces should be on their own line, in order to more clearly delineate the scope of the block.

Index

Table of Contents