

The HyperC Programming Language

Martin Dowd

1. Introduction. C++ is in wide use as a production programming language. However it can be argued to have deficiencies, and that there are extensions of C which are actually better suited to production programming. The source of these deficiencies is paradoxically the rationale for the language, namely, data abstraction.

Deficiencies of C++ that can be cited include the following.

- Increased static complexity. Indeed, the gains provided by data abstraction must be contrasted against the increased probability of error due to increased amount of source code.
- Misbehavior of the runtime facilities involved in C++ data abstraction, an infamous example being errors that can result in freeing storage with class destructors.
- Necessity of using “friend” in heterogeneous (involving multiple data types) operations.
- Cumbersomeness of the virtual function method of varying method implementation among related data types.

A suggested cure for these deficiencies would be an extension of C, which provided data abstraction in a more transparent and minimalist manner. HyperC is such a language. Design principles of HyperC include the following.

- Transparency should rarely be sacrificed in layering software.
- Facilities should have interfaces and implementations which are as simple and straightforward as possible.
- Redundancy among features should generally be avoided.

The foregoing approach is in contrast to much current “orthodox” thinking in programming language design, “the further from assembly language the better”. The static complexity explosion of C++ suggests that a better compromise with transparency should be made. This perspective has been adhered to already in important software projects, the GNU C compiler suite itself for example.

The authors of the GNU Cobol compiler, in section 14.3.1.3, “Modularity”, complain about C that “It does not fully support abstract concepts like data types and encourages the data interchange between modules using global variables. Object oriented programming, which offers much for the implementation of the translator, is only possible with great difficulty.” This is only partly true, since the “global variables” can be structures, and indeed this is common practice in modularizing large C programs. Further, this is an efficient method. “Shoehorning” global data into classes is a source of added complexity in C++ programs. Re-entrance is a special consideration which can be solved by other methods (shared code for example).

On the other hand various source code, for example the GCC source code, could be simplified if additional language constructs were available. The intent of HyperC is to provide additional tools for data abstraction and modularization. The model of data abstraction is basically that of C, with some added features from C++:

- a structure type specifies an object descriptor,
- a structure embodies an object, and
- a function with typed arguments realizes a homogeneous or heterogeneous method.

Thus, C itself, a full-fledged functional programming language, provides some support for abstract data types, in a simple and efficient manner. HyperC adds a few features expanding on this theme. Additional

ability to implement abstract data types is provided by adding minimal, transparent features to a modern functional programming language.

To quote “Object-Oriented Programming Versus Abstract Data Types” by William R. Cook, C++ supports both abstract data types and object-oriented programming. HyperC can be seen as de-emphasizing object-oriented programming in favor of abstract data types, in that objects are structures on which methods act, rather than having the methods bound to the objects in some way.

Providing a new language rather than using C++ in minimalist manner has the advantage, in keeping with the philosophy of such a language, of “regimenting” coding practice in the pursuit of less error-prone programming styles. Other advantages include briefer and more readable documentation, and improved compile time efficiency.

2. Features. HyperC is an extension of GNU C, which itself is an extension of ISO C, although some changes are made. The features included in the extension are meant to provide the advantages of C++ (encapsulation and memory management), without the disadvantages. Some of them are taken directly from C++.

The list of changes is as follows.

- Structure type names (tags) may be referenced without the “struct” prefix, and similarly for union and enumeration tags.
- The compiler option “traditional-cpp” cause the compiler to abort.

The list of additional features is quite short, as follows.

- Inheritance in structure declarations, using the “inherit” keyword.
- Overloading of function names. Names which are overloaded, and hence “decorated” for compilation and linking, must be explicitly declared as such using the “overload” keyword. Otherwise HyperC overloading is similar to GNU C++ overloading. In particular the function declaration must be of the prototype variety.
- Operators, a specialized form of C++ operators. Operator functions automatically have the overload qualifier.
- Compilation unit sectioning, using the “section”, “public”, and “private” keywords.
- “Loop” statement.
- Pool-based automatic deallocation of dynamic storage, using the “setpool” and “freepool” functions.
- Additional dynamic storage function “freez”.

A name which is declared as a structure type name (structure tag) may be used without prefixing it with “struct”; and similarly for union and enumeration tags. A structure, union, or enumeration tag is no longer in a separate “overloading class”; rather it is in the main overloading class. However, for compatibility if a name is declared as both in a scope, a warning will be issued, and it will be treated as a symbol. Some such uses occur in the header files; the warning will be suppressed unless “-Wsystem_headers” is specified. In some cases the the symbol is a typedef for the tag; In other cases the user must declare the tag within a scope.

The “inherit” keyword is an additional qualifier allowed in declarations of structure fields. The type T must be a structure type. A name N is optional, and if present there can only be one. Recursively, the fields and substructures for T are added as fields and substructures of the structure type being declared; and if N is specified it names the substructure of type T . Within a structure type, no field or substructure may have the same name. Note that this precludes two inherited substructures having the same type.

If “inherit” is not specified for a field with a structure type then the behavior is as in C. The name of such a structure field cannot be the same as that of any inherited field or substructure. An inherited field or substructure is referenced with a single dot, i.e., as $S.N$ where S is a structure and N the name of the

inherited field or substructure. An inherited substructure reference may be used as any structure reference. There are conversion rules for function overloading, described below.

The “inherit” keyword is allowed for an unnamed field, but does nothing.

The overloading rules of HyperC are as follows. If a declaration has the same mangled name as one already seen, the first declaration becomes shadowed by the second. For each visible declaration, for each actual argument, it is determined if there is a conversion to the type of the formal argument. If so the conversion is assigned a weight. For actual argument position i let $w(i)$ be the minimum weight; let Cnd be the set of visible declarations which have weight $w(i)$ at each position i . Cnd must be nonempty.

Suppose T is the actual argument type and U the formal argument type. If T is a pointer let T' denote the target type, and if U is a pointer let U' denote the target type. U can only have the same or more qualifiers than T ; and if T and U are both pointer types the same applies to U' and T' . The weight equals $2w+q$ where q is 1 if U has more qualifiers than T , and 0 if it has the same qualifiers (including the qualifiers of T' and U' when T and U are both pointer types). w is as follows, where qualifiers have been removed from T and U (and T' and U').

- 0 if T and U are compatible,
- 0 if T and U are both pointers and T' and U' are compatible,
- 1 if T is char or short and U is int
- 1 if T is unsigned char or unsigned short and U is unsigned
- 1 if T is float and U is double
- 2 if T and U are arithmetic or boolean types
- 2 if T is 0 and U is a pointer
- 2 if T is a pointer, with no qualifiers, and U is a void pointer
- 3 if T is a structure type and U is the type of an inherited substructure of T
- 3 if T is a pointer to a structure type T' , U is a pointer to a structure type U' , and U' is the type of an inherited substructure of T'
- 4 if T is any type and U is ellipsis ($q=0$ in this case)

There are some variations from C++.

Let P be the argument positions where the minimum weight is 3. At such a position the argument of each declaration in Cnd must be a structure type or pointer to such; among these structure types there must be a minimum enclosing the actual argument structure type. There must be a unique declaration in Cnd which has this minimum in each position in P ; if so, this declaration matches the call.

Overloaded names may be used for assignment to function pointers, provided the declaration of the pointer suffices to determine a unique matching declaration. This is true for implicit assignments, via initialization, argument binding, or return value binding.

In HyperC, all structure arguments to operator functions are passed by reference. To pass structures by value, ordinary functions must be used. An operator function declarator must have one of the following forms.

- 1 operator $U(Tx)$ where U is one of $+$, $-$, \sim , $!$.
- 2 operator $B(Tx,Ty)$ where B is one of $+$, $-$, $*$, $/$, $\%$, \wedge , $\&$, $|$, $\&\&$, $||$, $<<$, $>>$.
- 3 operator $R(Tx,Ty)$ where R is one of $<$, $>$, $<=$, $>=$, $==$, $!=$.
- 4 operator $[(Tx,Ty)$.
- 5 operator $=(Tx,Ty)$.
- 6 operator $A(Tx,Ty)$ where A is $B=$ and B is as in case 2.
- 7 operator $P(Tx)$ where P is one of $++$, $--$.
- 8 operator $Q(Tx,int)$ where Q is one of $++$, $--$.

Neither T_x nor T_y can be a structure type, and at least one of the must be “struct $T *$ ” for some tag “ T ”, possibly with qualifiers. Case 7 is used to define prefix operators, and case 8 postfix. In case 5, T_y may not be the same as T_x .

If an argument of an operator is of type “struct T ”, a search is made for a matching operator function, by the overloading rules given above, with the type considered to be “struct $T *$ ”. The address of the actual argument is passed to the operator routine. The actual argument may be an lvalue, or a function call expression whose value is a structure of type T , in which case a temporary object is created. Unlike C++, a formal argument need not have the “const” qualifier to be bound to a temporary object; the programmer must exercise appropriate care.

In all the above cases in C, if an operator argument has structure type an error occurs. In HyperC, a check will be made for a unique best overload match. The C++ syntax “operator<op>” for an operator reference is not supported. As a result, the address of an operator function may not be taken by this method; see section 6 for a method for doing so.

The “section” statement has the form “section B ” where B is a compound statement (the section body). Sections may be nested, but must be toplevel within the enclosing section. All function and variable declarations in any section are toplevels, but their visibility is restricted. Names which are typedef names, struct, union, or enum tags, or enum constants may be declared within a section, but are declared as ordinary toplevels.

The labels “public” and “private” may be used in a section. Toplevels following a “private” label are not visible past the end of the section. Those following a “public” label are, within the enclosing section if any. At the start of a section, toplevel declarations are private by default. Nested sections may appear anywhere; the public and private qualifiers have no effect on their behavior.

There is no shadowing arising from sectioning; visible declarations must all have distinct names. Other restrictions are as follows.

- “main” cannot be declared within a section.
- declarations of external scope must occur at level 0, or as public at level 1.

The form of the loop statement is “loop (E) B ” where E is an expression of integer type and B is a compound statement. B is executed n times, where n is the value of E when the loop statement is executed ($n < 0$ is equivalent to $n=0$). The break and continue statements behave as usual in B .

3. Additional functions.

The “int setpool(unsigned char i)” function sets the current “storage pool”. Dynamic memory allocated subsequent to setting the pool to i is marked as belonging to pool i . Pool 0 is ordinary dynamic memory, and receives no special handling. Only the low order byte is significant, so there are at most 255 pools.

The “void freepool(unsigned char i)” function frees the pool i . If $i = 0$ no action is taken.

The “void freez(void *)” function frees the dynamic storage pointed to by the pointer, and sets the pointer to 0. The argument must be an lvalue.

In this version of HyperC, these functions are implemented using a header file “hyc_mem.h”, and a corresponding source file. The standard functions “malloc”, “calloc”, “realloc”, and “free” are defined as macros in the header file; and “freez” also. Note that the compiler will report an error if the argument of freez is not an lvalue. It is necessary to include “hyc_mem.h” after the declarations of malloc et al.

Eventually these should be implemented as built-ins. Built-ins are implemented in the “middle-end” of the compiler “cc1”. The compiler “hyc” is obtained by changes only to the front end. In fact, the memory allocation functions require no changes to the compiler at all, and can be used with any C compiler.

4. Preprocessor. For production programming a preprocessor with enhanced functionality over the C preprocessor is advantageous. The authors of the C language were not concerned with this issue, and designed

a minimal preprocessor, compared to preprocessors which were in use in assembly language programming at the time.

HyperC provides an enhanced preprocessor, incorporating some features from a stand-alone preprocessor which was marketed by Hyperon Software in the 1980's, into the GNU "CPP" built-in preprocessor.

Briefly, the current commands may be described as

```
#define name[(args)] token_string
#if expression / #elif / #else / #endif
```

The following new commands are added.

```
#set name expression
#while expression / #continue / #break / #endwhile
#func name[(args)] / #endfunc token_string
```

There is a new predefined macro,

```
--INDEX__(N,I).
```

In an invocation, the argument N must be a simple (not function-like) macro name. The argument I must be a set macro name. Macro expansion is not performed on the arguments. If *i* is the value of I, the value of the invocation is the *i*th token in the token list giving the definition of N (with indexing starting at 0); or empty if *i* is out of range.

In a macro replacement, a "#set" name is replaced, as the token representing it's value. A "#func" name is replaced by the value of the corresponding "#endfunc" token_string, after executing the function body. Note that the latter may be empty; the body is thus executed only for side effects.

The value of a set macro is considered to overflow if it overflows internal precision during the computation, or the final result does not fit in a signed long. In the case of overflow, the value of the expression is set to 0.

While "executing" a #func macro body, argument replacment is applied to each directive before it is executed. The name in a #define, #undef, or #set directive in a #func body cannot be an argument; and neither can the formal arguments in a function type #define directive. Note that in the gcc preprocessor, argument replacement involves macro expansion of the actual arguments.

It is worth noting that with the above rules, the token string assigned to a name by a "#define" directive can be "computed" (by defining it in a func body and then invoking the func macro).

Restrictions on the constructs are as follows.

- #if / #while / #func constructs must be properly nested.
- A construct which begins in an included file must end in the file.
- A #func body can only contain #define, #undef, and #set commands, #if constructs, or #while constructs, or null directives.
- A #while construct cannot contain #include or #func commands.
- Func bodies receive a pre-scan, and if there is an error the body is made empty.
- The name in a #define, #undef, or #set directive in a #func body cannot be an argument.
- As already noted, the name in a #define, #undef, or #set directive in a #func body cannot be an argument; and neither can the formal arguments in a function type #define directive.

Other restrictions are as follows.

- Once a func or set macro is defined, it cannot be undefined, or defined as a macro of any other type.
- A func macro can defined only once.
- As for any other use, a poisoned name cannot be used as a set or func macro name
- Comments are ignored in func bodies.

- Various values have a 65535 max, including the number of overload functions, set macros, and func macros; number of tokens in a func or while body line; and total number of lines in func and while bodies.

The enhanced preprocessor is incorporated into the “hyc” compiler, so is automatically used by it.

5. Rationales and possible alternatives. The features listed in preceding sections have variations, and other features could be included. In this section a discussion will be given of rationales for the features as presented. This is the first release of HyperC, and changes can readily be made in later releases.

The “inherit” specifier provides a general form of inheritance, with minimal changes to C. As such, it exemplifies the design philosophy of HyperC. Together with the overloading rules, it supports replacing “class libraries” of C++ with “structure type libraries”. The resulting facility is both more general and more efficient. The “unnamed field” GCC extension could be deprecated, as unwieldy and redundant. The current behavior of inherit on “unnamed” fields is to ignore it. This was done for convenience.

The overloading rules prefer conversion to “void *” to conversion to “U *” when an actual argument has type “T *” and “T” has an inherited substructure of type “U”. This was done for convenience, and possibly should be changed. For one example, “free” can be used for “void *”, with an overloaded “delete” for any desired structure type pointer. Note that if “T” has an inherited substructure of type “U”, and a delete function is provided for type “U”, then one should probably be provided for type “T”.

The requirement that structure tags be in the main name class allows omitting “struct” when declaring a variable of some structure type. In fact, specifying “struct” could be disallowed, eliminating a useless redundancy. Union and enumeration tags are treated the same way for uniformity; but the real advantages are for structure types. Unions could in fact be deprecated.

The “single dot” rule for referencing inherited fields and substructures is the simplest, and involves no loss of generality. Again, this exemplifies the design philosophy.

The “setpool” and “freepool” functions provide simple, easy to use, and fairly powerful garbage collection. The “setpool” function seems preferable to specifying the pool per allocation. In well-structured code, the pool should be changed at strategic points. Note also that it is a single line of assembler code.

The “freez” function may be used to explicitly free memory which would be freed by exit code. This feature relies on the ISO C behavior of the “free” function, that it has no effect if the pointer is null.

A function “freall” would be useful for processes which clean up, but do not exit. This would have to be added to libc.

There is a powerful mechanism for freeing dynamic memory available within GCC, which should be used as much as possible, with garbage collection used only when necessary. By the same token, it should be used in favor of “_ATTRIBUTE_ cleanup(...)”.

Suppose at some point in a function body a temporary array “a” of length “n” is required, where “n” is a value which has been computed. The code “{<type> a[n]; ...}” makes the array available for the required computation. If the block is exited with “goto” or “longjump” the array is automatically freed. If it is desired to make “a” available to a subroutine, it can either be passed as an argument, or a global variable “<type> *ap;” can be declared, and “ap=a;” executed at the start of the block. Note that in the latter case, the subroutine can be used by any code that sets “ap”, by any method. In HyperC, “ap” and subroutines which use it can be placed in a section.

For the usual reasons, it can be argued that no “syntactic sugar” need be added for using this method; the “boilerplate” code is completely straightforward.

The rules for the “section” statement are the simplest possible. Allowing private typedef names, and struct, union, and enum tags, should be considered. Private fields in public structures should also be considered. Named sections might be useful in one context; see below.

The “loop” statement is redundant, but is such a common iterator that it should be included in any programming language. Optimization can take note of it.

The overloading rules for substructures preclude the reference $f(T)$ when $f(U)$ and $f(V)$ are declarations with U and V distinct maximal substructures of T . They also preclude $f(T_1, T_2)$ when $f(U_1, V_2)$ and $f(U_2, V_1)$ are declarations, with U_1 and V_1 greatest.

There is a fact of partial order theory which illustrates the sensibility of the overloading rule. Suppose \leq_i is a partial order on S_i , and let \leq be the product order on $S_1 \times \cdots \times S_k$. Suppose $T \subseteq S_1 \times \cdots \times S_k$. Then $\langle t_1, \dots, t_k \rangle$ is a largest element of T iff t_i is a largest element in each $\pi_i[T]$ where π_i is the projection map.

The overloading rules of HyperC were designed to keep the implementation as simple as possible, in particular minimizing the changes required to the C parser. For this reason default arguments have been omitted from the initial version of HyperC, and default argument variants must be given separate declarations. In a final version default arguments might be added.

Full C++ references complicate the language and compiler significantly. In the case of ordinary function calls, reference style argument passing can be explicitly coded. However, this is not true for operators. In the case of assignment operators, passing the left hand side by reference is highly desirable. In general, passing larger structures by reference is desirable also. The implementation of operators in this initial version of HyperC was designed to address these issues. Support for other forms of declarators might be of interest, however. Another variation of operators would allow statement expressions as arguments.

Implicit arguments to functions could be added, in a more general way than in C++. An argument of type “pointer to T” where “T” is a structure type could be “promoted”, so that its fields could be referenced with a single identifier. This seems to introduce additional possibility of error, and be less self-documenting. Single character formal arguments keep the additional static complexity to a minimum.

Multilevel break, continue, and return are possible additions. However these can readily be simulated using goto and longjmp, so the utility of these additions is not great. These features are very useful in “real-world” programming, and have an undeserved bad reputation.

This version of HyperC is an extension of gcc partly as a matter of convenience. In a more definitive version some features of gcc might be deprecated. Two examples of such are the boolean and complex types. Boolean was treated as integer in the original design of C, because this was a simple method familiar to system programmers. Re-instating a separate type can be seen as reversionism. The “complex” type can be declared, and methods as suggested above might achieve optimum performance.

Another change worth considering is deprecating the keywords “static” and “extern”, in favor of the keywords “import”, “export”, and “global”.

Toplevels which are externally visible must be declared with the qualifier “export”. If there is a declaration, followed by a definition, “export” must be specified on the definition.

Toplevels which are defined in another module must be declared with the qualifier “import”. The “import” qualifier can be specified on the declaration. Alternatively, a declaration without import may be followed by “import N ” where N is the name.

Toplevels which are neither exported nor imported have scope local to the module; thus, it is no longer necessary (or legal) to specify “static”. For the other use of “static”, to specify that variables defined in a block are actually global, the “global” qualifier is provided.

Section names could be used for making sections externally visible. A named section may be made externally visible adding the “export” specifier to the section statement, or the second occurrence if the name occurs twice. Any declaration visible in the defining module is externally visible. An exported named section may be used in a client module by adding the “import” specifier to the section statement; the contents must consist of public declarations. Alternatively, the declaration may be followed by “import N ” where N

is the name.

Requiring the “export” specifier on the definition rather than the declaration, if any, facilitates including a header file in the implementation module, for type checking. In a client module, the header file inclusion merely need be followed by “import *N*” where *N* is the name. To avoid lengthy import lists, a named section may be used. These rules are simple, and eliminate the arcane and sometimes platform-dependent rules of C.

In the current version of HyperC, abstract types can be implemented, simply as incomplete structure types. Functions can be declared having pointers to such types, and overloading will perform properly. A pointer to an object can be cast to a pointer to an abstract type. Functions can rely on the C rule that the first field of a structure has the same address as the structure.

It might be worth enhancing this feature, at the very least providing a type specifier alternative to “struct” (such as “target”), for types that are intended to remain incomplete, and serve as a placeholder for overloading. A more extensive enhancement would allow specifying a list of concrete structure types, which could undergo pointer conversion to the abstract type in overload resolution; and cause a warnings if a cast to some type not on the list is specified.

Another type management feature that might be enhanced is casting of pointers. Generally, a pointer to an inherited substructure can be obtained by taking its address; however if it is hierarchically the first component of a chain of inherit's, it can be obtained by a simple typecast. A “safe” cast operator which checks for this could be added, say as an overloaded builtin function “safecast” with two pointer arguments, the first implicitly by reference.

The enhancements to the preprocessor make it much more robust, and are relatively easy to add to the existing preprocessor. Various additions are worth considering, in particular

- updatable variables with token string values,
- `__LENGTH(N)` builtin macro, and
- templates.

The rules for `__INDEX__` avoid having to relax the language to allow an expression to occur outside a directive. Further, commonly a set variable would be used as the index.

6. Compiling and debugging. Compilation of HyperC (see below) produces a stand-alone processor “hyc”, which translates a “.c” file to a GNU “.s” file. Thus, it is intermediate between a preprocessor and a full-fledged addition to the gcc compiler suite. To use hyc, gcc and its associated libraries, header files, etc., must be installed. Installation of hyc is left to the user (see below). Since hyc is a modified version of cc1, it should use the standard gcc support files.

The default for the stand-alone cc1 compiler, and so the hyc compiler, is to produce some timing messages. This can be turned off with the “-quiet” option. The messages don't occur if the compiler is run with the driver, because this passes “-quiet” by default.

The modified parser produces a “generic” parse tree, as the C compiler would from a suitably translated source file. The gdb debugger may thus be used as is, provided the user is aware of the conventions used in the translation.

- Fields accessed via inherited subfields are converted to the equivalent complete sequence of substructures.
- Overloaded functions are renamed, according to the C++ ABI.

There are some variations from the C++ mangling. Array arguments are mangled as pointers. Operators are coded as functions, whose name is “_Z<op>” where <op> is the two letter code; an overloaded operator thus starts with “_Z4_Z”. Sections are numbered consecutively from 1, and a toplevel belonging to section “<num>” has “_ZN<num>_” prepended. Public toplevels in a section are considered as belonging to the parent section, or no section if the section is not nested.

The mangled name of a function or operator, or the name “ \underline{Z} <op>” for the base name of an operator, may be used anywhere this makes sense. In particular, the latter may be used in an assignment, where the left hand side suffices to resolve the overload, just as for an ordinary function.

There is no driver, or stand-alone preprocessor, in the initial version of HyperC. Other utilities might need revising in a fully integrated version; in particular the mangler and / or demangler probably need improvement to ensure demangled printing in various contexts.

7. Compiling and installing the compiler. The HyperC distribution consists of a directory “hyc”; it has one subdirectory, containing examples. The result of compilation is a stand-alone compiler “hyc”; and an object module “hyc_mem.o” which contains the implementation of the HyperC memory allocation functions.

Prior to compiling hyc, the following steps must be performed.

1. Obtain the gcc-4.1.1 distribution “gcc-4.1.1.tar.bz2”. This is available at “gcc.gnu.org”.
2. Extract the gcc-4.1.1 tar file in the parent directory of hyc, producing the directory “gcc-4.1.1” parallel to hyc.
3. Create the directory “cc1o” parallel to hyc, so that the parent of hyc has three subdirectories, “cc1o” (empty), “hyc”, and “gcc-4.1.1”.

Compiling with other versions of gcc might be possible. However all testing done by Hyperon Software has been with version 4.1.1.

Compilation of hyc will probably succeed using various compilers; however it is safest to use gcc. It is not necessary to use version 4.1.1; for example hyc has been compiled with version 2.95.3.

Compilation of cc1 and other GNU modules uses the GNU Makefiles. The GNU modules are compiled using the native C compiler, and are not “bootstrapped”. This method allows debugging. Again, version 2.95.3 of gcc is suitable as a native compiler.

Steps to compile the standard modules are as follows.

1. Change to the “cc1o” directory.
2. `../gcc-4.1.1/configure`
3. `make configure-host`
4. `make all-build-libiberty`
5. `make all-libcpp`
6. `cd gcc`
7. `make c`

The preceding steps will compile the standard modules with the “-g” option. This can be remedied as follows. After step 3,

in Makefile, remove -g from CFLAGS_FOR_BUILD and CFLAGS;

in gcc/Makefile, remove -g from CFLAGS and STAGE1_CFLAGS.

Steps to compile the modified modules are as follows.

1. Create a parallel subdirectory (e.g. “hyco”), change to it, and copy the Makefile from “hyc”. Alternatively, stay in “hyc”.
2. Issue “make”.

There is no default installation; users must install the compiler executable “hyc”, the header file “hyc/hyc_mem.h”, the object file “hyc_mem.o”, and the documentation file “hyc/usrguide.pdf” manually. These files can be left where they are; installed in a standard place such as “/usr/local/bin”, or installed in directories where “gcc” files are installed. Makefiles need only be adjusted as necessary.

hyc has a subdirectory “exmp”. This has the source code for several examples, and a Makefile to compile and run each of them. The Makefile assumes that hyc is compiled into the subdirectory “hyco” parallel to “hyc”; it may be adjusted as necessary. It assumes the native compiler is gcc.

8. Internals. As noted in section 6, the HyperC compiler “hyc” is derived from the GNU cc1 compiler, by a few modifications to the source files.

The cc1 source files modified are
c-typeck.c, c-parser.c, c-decl.c, and c-tree.h.

The modified header file is only used for the modified source files; the compilation of the standard modules use the standard header file.

In addition, the files lex.c, macros.c, directives.c, and expr.c from libcpp are modified,
There is a new file, hyc-subr.c.

All changes are marked with a comment, which includes the text “HYC”.

This is a preliminary version, and needs to be better integrated into the GNU compiler collection. It has been coded to minimize the amount of modification necessary. As a result, structures and fields which would speed up processing have not been added, and simple-minded methods used instead.

The issue of re-entrance was ignored, and global variables are freely used.

It is testimony to the quality of the gcc parser implementation that the enhancements for HyperC amounted to some 2500 lines of code.