The Obstack class

The **Obstack** class is a simple rewrite of the C obstack macros and functions provided in the GNU CC compiler source distribution.

Obstacks provide a simple method of creating and maintaining a string table, optimized for the very frequent task of building strings character-by-character, and sometimes keeping them, and sometimes not. They seem especially useful in any parsing application. One of the test files demonstrates usage.

A brief summary:

grow	places something on the obstack without committing to wrap it up as a single entity yet.
finish	wraps up a constructed object as a single entity, and returns the pointer to its start address.
copy	places things on the obstack, and *does* wrap them up. copy is **always** equivalent to first grow, then finish.
free	deletes something, and anything else put on the obstack since its creation.

The other functions are less commonly needed:

blank	is like grow, except it just grows the space by size units without placing anything into this space
alloc	is like **blank**, but it wraps up the object and returns its starting address.

chunk_size, **base**, **next_free**, **alignment_mask**, **size**, **room** returns the appropriate class variables.

grow_fast places a character on the obstack without checking if there is enough room.

blank_fast like **blank**, but without checking if there is enough room.

shrink(int n) shrink the current chunk by n bytes.

contains(void* addr) returns true if the Obstack holds the address addr.

Here is a lightly edited version of the original C documentation:

These functions operate a stack of objects. Each object starts life small, and may grow to maturity. (Consider building a word syllable by syllable.) An object can move while it is growing. Once it has been ``finished'' it never changes address again. So the ``top of the stack'' is typically an immature growing object, while the rest of the stack is of mature, fixed size and fixed address objects.

These routines grab large chunks of memory, using the GNU C++ **new** operator. On occasion, they free chunks, via **delete**. Each independent stack is represented by a Obstack.

One motivation for this package is the problem of growing char strings in symbol tables. Unless you are a ``fascist pig with a read-only mind'' [Gosper's immortal quote from HAKMEM item 154, out of context] you would not like to put any arbitrary upper limit on the length of your symbols.

In practice this often means you will build many short symbols and a few long symbols. At the time you are reading a symbol you don't know how long it is. One traditional method is to read a symbol into a buffer, **realloc()**ating the buffer every time you try to read a symbol that is longer than the buffer. This is beaut, but you still will want to copy the symbol from the buffer to a more permanent symbol-table entry say about half the

time.

With obstacks, you can work differently. Use one obstack for all symbol names. As you read a symbol, grow the name in the obstack gradually. When the name is complete, finalize it. Then, if the symbol exists already, free the newly read name.

The way we do this is to take a large chunk, allocating memory from low addresses. When you want to build a symbol in the chunk you just add chars above the current ``high water mark'' in the chunk. When you have finished adding chars, because you got to the end of the symbol, you know how long the chars are, and you can create a new object. Mostly the chars will not burst over the highest address of the chunk, because you would typically expect a chunk to be (say) 100 times as long as an average object.

In case that isn't clear, when we have enough chars to make up the object, *they are already contiguous in the chunk* (guaranteed) so we just point to it where it lies. No moving of chars is needed and this is the second win: potentially long strings need never be explicitly shuffled. Once an object is formed, it does not change its address during its lifetime.

When the chars burst over a chunk boundary, we allocate a larger chunk, and then copy the partly formed object from the end of the old chunk to the beginning of the new larger chunk. We then carry on accreting characters to the end of the object as we normally would.

A special version of grow is provided to add a single char at a time to a growing object.

Summary:

- We allocate large chunks.
- We carve out one object at a time from the current chunk.
- Once carved, an object never moves.

- We are free to append data of any size to the currently growing object.
- Exactly one object is growing in an obstack at any one time.
- You can run one obstack per control block.
- You may have as many control blocks as you dare.
- Because of the way we do it, you can `unwind' a obstack back to a previous state. (You may remove objects much as you would with a stack.)

The obstack data structure is used in many places in the GNU C++ compiler.

Differences from the the GNU C version

1. The obvious differences stemming from the use of classes and inline functions instead of structs and macros. The C **init** and **begin** macros are replaced by constructors.
2. Overloaded function names are used for grow (and others), rather than the C **grow**, **grow0**, etc.
3. All dynamic allocation uses the the built-in **new** operator. This restricts flexibility by a little, but maintains compatibility with usual C++ conventions.
4. There are now two versions of finish:
    1. finish() behaves like the C version.

2. finish(char terminator) adds **terminator**, and then calls **finish()**. This enables the normal invocation of **finish(0)** to wrap up a string being grown character-by-character.

5. There are special versions of **grow(**const char* *s***)** and **copy(**const char* *s***)** that add the null-terminated string *s* after computing its length.

6. The shrink and contains functions are provided.