A-027

メモリリークを防ぐためのC++ライブラリの開発

Development of C++ Library Detecting Memory Leak

田中　学†
Manabu Tanaka

1.はじめに

メモリリークを防ぎ、開放を自動化する手法としてガーベジコレクタ（以下GC）、参照数カウント型スマートポインタ方式といった機構がよく用いられる。

参照数カウント型スマートポインタ方式は循環参照問題が発生した場合にメモリリークを発生してしまうという短所がある。参照カウンタ型スマートポインタを実装しているライブラリでは、それに関して所有権の無いポインタや弱参照ポインタを採用することによって対処している。

GCにはガーベジ回収処理に時間がかかり、またいつ発生するか予測できないという欠点がある。この欠点を克服するため、処理を分割し細切れに行うことによって処理を分散したり、世代型管理を行ったりすることによって対処している。

本研究では、参照カウント型のスマートポインタとGC双方の機能を取り入れた、新しいスマートポインタを実装することによって問題を解決し、その手法が実用レベルの処理効率となるかどうかについて考察する。

2.参照数カウントスマートポインタの実装

参照数カウント方式のスマートポインタの具体的な実装方法は数種類あって、それぞれに長所・短所がある。本研究で製作するライブラリではテンプレートによるポリシークラスの概念を導入する事により、ユーザーが実装方法を差し替えられるようなスマートポインタを製作した。[1]

これによって使用する環境等により臨機応変に対応したスマートポインタを使用することが出来る。

本研究では次の４種類のスマートポインタを実装した。

（１） 参照カウンタ方式

new演算子によって生成されたオブジェクトが渡されると、同時に整数型をヒープ上に生成し、その整数へのポインタをオブジェクトのポインタと一緒に共有する。

ポインタがコピーされるとカウンタが増加し、破棄されると減少する仕組みで、参照カウンタが０になった時にヒープ上に生成した整数と管理しているオブジェクトを開放する。

（２） 参照カウンタ参照方式

基本的には（１）と同じだが、本体にポインタを持たずヒープ上に参照カウンタと一緒に持たせてしまう。メモリ効率は良くなるがオブジェクトを参照するのに時間がかかる。

（３） 双方向リンクリスト方式

ヒープ上のオブジェクトを必要とせず、リンクリストによって実装する方式。他と比べてオブジェクトのサイズが大きくなってしまうのが問題。

（４） 侵入参照カウンタ方式

オブジェクト自体が参照カウンタを内蔵していて、スマートポインタはそれを増減させるだけでよい。予めスマートポインタを想定してクラスを設計しなければならないが、効率は非常に良い。

これらのうち、(1)と(2)は参照カウンタをヒープ領域に確保する必要があるが、汎用アロケータのnew演算子を使うと効率が良くないので、小さいオブジェクト用の専用アロケータを使用することにした。

3.ガーベジコレクタの実装

GCにはいくつかの種類があるが、本研究ではマーク&スイープ方式を採用した。この方式では、スタック変数やレジスタ、グローバル変数といったプログラムが直接アクセスできるルートオブジェクトと呼ばれるポインタから順にメモリ内を辿り、到達出来るか否かを調べる方式である。

ある程度メモリの使用率が高くなるとGCが起動して全てのオブジェクトを辿り、そこから到達できないヒープオブジェクトを何処からも参照されていないゴミ（=ガーベジ）として回収する方法である。

D言語, Java,C#といった最近の言語に実装されており、C/C++用GCライブラリBoehm GCといったものも存在する。[2]

C/C++用のライブラリの場合、ルートオブジェクトを決定するのは言語仕様レベルでは難しく、ＯＳ依存のコードになる場合が多い。しかし、本研究で開発したＧＣは、スマートポインタのコンストラクタ・デストラクタを利用することによって、ＯＳに依存しない汎用的なライブラリの開発に成功した。

†茨城工業高等専門学校専攻科情報・電気電子工学専攻

４.ＯＳ非依存ガーベジコレクタの実装例

本研究で実装したガーベジコレクタは次のように動作する。まず、new 演算子をオーバーロードし、ヒープ上に割り当てられたオブジェクトのポインタとそのサイズをグローバルな set クラス(g_aHeap)に格納する。この g_aHeap によって、あるポインタがヒープを指しているか否かを調べられるようにする。

次に、スマートポインタのコンストラクタで、this ポインタがヒープ領域内にあるかどうか g_aHeap を調べる。ヒープ領域内なら、そのポインタはクラスメンバとして定義されているか、new 演算子によって生成されたかのどちらかなので、ルートオブジェクトではないとする。逆にヒープ領域になかった場合、そのスマートポインタはグローバル、スタティック、スタックオブジェクトのどれかなのでルートオブジェクトとする。ルートオブジェクトと分かった場合、それのポインタをグローバルな set クラス(g_aRootObj)に格納し、それをルートオブジェクト一覧として使用する。

逆にスマートポインタのデストラクタでは、g_aRootObj を探索して自分へのポインタがあるかどうか調べ、見つけたら削除する。

ＧＣが起動すると、ルートオブジェクトからヒープ領域を辿って再帰的にマークをつけ、マークが付いていないオブジェクトはゴミとしてスイープする。

この方式の場合、BoehmGC などで採用されている保守的なＧＣと違い、スマートポインタ以外の生のポインタや参照型によって参照されているヒープオブジェクトをゴミと判断し破棄してしまう。

しかし、逆にルートオブジェクトが少なくて済むためマークにかかる時間も短縮でき結果的に高速に動作するのではないかと思う。

５.効率

製作したライブラリの効率を従来のライブラリと比較検討した。しかし、ＧＣがその能力を発揮するのはオブジェクト同士の相互関係が複雑な場合なので、テストケースを用意するのは困難と判断し、今回は参照カウンタ方式の効率測定に留めた。

ポインタでよく行う操作である、「実体への参照」「コピー」「インスタンス生成」を行い、それぞれの速度を計測した。本研究で製作した４種類のスマートポインタの他、スマートポインタでない生のポインタと boost[3]の shared_ptr と比較し、製作したライブラリがどの程度実用可能か比較・検討する。

今回コンパイラとして使用したのは gcc3.2 で、最適化は速度に関して最大となるようにした。

５.考察

今回開発したスマートポインタは boost で実装されているそれと比べて高速に動作することが確認できた。これは、boost では参照カウンタクラスに仮想関数を用いてポリモーフィックなクラスとして実装しているためのオーバーヘッドと、参照カウンタ生成に new 演算子を使っているためと考えられる。

本研究で開発したスマートポインタでは参照カウンタには単純な構造体を使用し、生成には小規模オブジェクト用の専用アロケータを用いることによって、コピーや生成の効率をあげている。

開発したライブラリの、それぞれの実装方法を比較してみると、やはり侵入型の効率が一番良く、リンクリスト方式の効率はあまりよくなかった。

これはヒープオブジェクトの生成が専用アロケータによって効率化されたため、ヒープを使わない事を利点としたリンクリスト方式と他の差がなくなってしまい、リンクリスト方式の欠点だけが表に出てしまったためと考えられる。

６.今後の課題

現時点までの研究で汎用性が高く、高性能なスマートポインタを製作することが出来た。また、ＧＣを実装することによって、循環参照によるメモリリークも防ぐことが出来た。

しかし、今回はＧＣの効率をテストするケースを用意することが出来なかったため、今後はその点について調査していきたい。

参考文献

[1] Modern C++ Design - Andrei Alexandrescu 著
[2] A garbage collector for C and C++
http://www.hpl.hp.com/personal/Hans_Boehm/gc/
[3] Boost C++ Libraries
http://www.boost.org/