Smart Garbage Collector

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF ENGINEERING IN

COMPUTER SCIENCE AND ENGINEERING

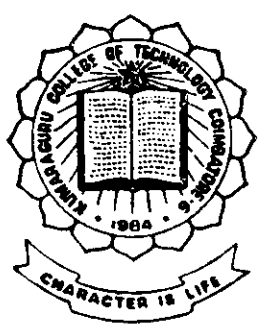
OF THE BHARATHIAR UNIVERSITY

SUBMITTED BY

C. S. SOWMYA
Reg. No. 9937K0012

GUIDED BY

Mrs. L. S. JAYASHREE M. E., MISTE.



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Kumaraguru College of Technology

COIMBATORE-641 006

2000 - 2001

Department of Computer Science and Engineering

Kumaraguru College of Technology

(Affiliated to the Bharathiar University)

Coimbatore – 641 006

Project Work

CERTIFICATE

Bonafide record of the project work

Smart Garbage collector

Done by

C.S.SOWMYA

(Reg. No. 9937K0012)

Submitted in partial fulfilment of the requirements

For the degree of Master of Engineering

in Computer Science and Engineering

Of the Bharathiar University

Faculty Guide

Head of the Department

Submitted for Viva-Voce Examination held on

Internal Examiner

External Examiner

Declaration

I, **C.S.SOWMYA** hereby declare that this project work entitled **"Smart Garbage Collector"** submitted to **Kumaraguru College of Technology, Coimbatore** (Affiliated to Bharathiar University) is a record of original work done by me under the supervision and guidance of **Mrs. L.S.JAYASHREE M.E., MISTE.** Department of Computer Science and Engineering.

Name of the Candidate	Register Number	Signature of the Candidate
C.S.SOWMYA	***9937K0012***	

(C.S.SOWMYA)

Countersigned by: Staff in Charge

Mrs. L.S.JAYASHREE M.E., MISTE.
Lecturer
Department of Computer Science and Engineering
Kumaraguru College of Technology
Coimbatore – 641 006

Place : Coimbatore

Date : [illegible]

Contents

Acknowledgement

I wish to express my sincere gratitude to **Dr.K.K.Padmanabhan, B.Sc. (Engg)., M.Tech., Ph.D.,** esteemed Principal, Kumarugu College of Technology for providing me the necessary facilities in the college.

I wish to express my heartfelt thanks and a deep sense of gratitude to Prof.**Thangaswamy,B.E.(Hons), Ph.D.,** The Head Of Department Of Computer Science And Engineering who motivated me by giving valuable ideas and suggestions.

I express my sincere thanks to my Guide **Mrs.L.S.JayaShree M.E., MISTE.** Lecturer, Computer Science And engineering without whose motivation and guidance I would not have been able to embark on a project of this magnitude.

I also like to express my indebtedness to my class advisor **Mr.R.Kannan M.E.** for his constant support and guidance, without which I could not have been able to complete my project successfully.

Last but not the least , I thank my beloved parents, friends, department teaching and non teaching staffs who have been a pillar of support right from the start, until the completion of the project.

Synopsis

The project entitled " **Smart Garbage Collector** " was developed for maintaining efficient Memory Management in C++.

Smart garbage collector implements new memory management technique that allows predictable real time performance even for systems that make heavy use of dynamic allocation and rely on the garbage collector to recycle sufficient memory to satisfy all future allocation request in short and bounded time.

The Smart Garbage Collector is implemented using the concept of Smart Pointers. Smart Pointers look, act and feel like pointers but they are smarter. They act as **proxies of ordinary pointers.**

The underlying technique used for smart garbage collector is the **Reference Count Method,** in which whenever a reference is made to the object, the reference count of the object is incremented by 1 and when the object loses its scope, the reference count is decremented by 1. When the reference count is equal to zero, which means there is no more reference to the object and which in-turn means it is no longer needed , so the object could be removed from the memory.

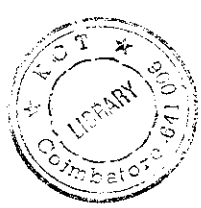
The Smart Garbage Collector is plugged into an application called Machine Shop Simulation to visualize the actual working of the Garbage Collection .The simulation results are found to agree with the analytical results to an appreciable degree.

1. Introduction

The lack of reliable hard real time memory management is the biggest hindrance for the application of modern programming languages, for the growing market of real - time systems. But even traditional languages like, C, C++ or ADA that are used in this area do not provide predictable memory management.

System implementers are often forced to restrain from using dynamic memory management in real time code together , so that modern object oriented development techniques can not be applied, memory footprint is increased due to static allocation and system becomes more complex to build and maintain.

Since computer software represents an increasing share of the complexity of systems for automatic control and other types of critical equipment, the issue of software quality becomes more and more important. One factor that influences the complexity, especially with the introduction of object-oriented implementation techniques, is how memory management is handled. Techniques like automatic memory management, garbage collection can decrease the risk of software errors and shorten development time.



Smart garbage collector resolves the above problem and provides effective memory management by using Smart pointers. The Garbage Collector will provide management for memory blocks allocated during program execution. The management of the memory blocks includes keeping reference counts on the memory, allocating memory, and freeing memory when reference counts go to zero. The garbage collector runs a process to remove all memory blocks that have a zero reference count.

1.1 Scope :

Smart Garbage Collector implements new memory management techniques that allow predictable real time performance even for systems that make heavy use of dynamic allocation and rely on the garbage collector to recycle sufficient memory to satisfy all future allocation request in short and bounded time.

1.2 Features:

To provide effective Memory management

Avoids memory leakage

Avoids Dangling references.

Avoids problems due to Auto pointers.

Performs all operations on Standard Template Library

1.3 H/w & S/w requirements :

Hardware :

PROCESSOR	:	Pentium Celeron @ 500 Mhz
RAM	:	64 MB
HARD DISK	:	10 GB
Software	:	VC++ for Windows
Platform	:	Windows 95.

2. System Study and Analysis

2.1 Memory Management in C++

C++ added the necessary language constructs to the memory model of C to support object semantics. In addition, it fixed some loopholes in the original model and enhanced it with higher levels of abstraction and automation.

Types of Storage

C++ has three fundamental types of data storage: automatic storage, static storage, and free store. Each of these memory types has different semantics of object initialisation and lifetime.

Automatic Storage

Local objects that are not explicitly declared static or extern, local objects that are declared auto or register, and function arguments have *automatic storage*. This type of storage is also called *stack memory*. Automatic objects are created automatically upon entering a function or a block. They are destroyed when the function or block exits. Thus, on each entry into a function or a block, a new copy of its automatic objects is created. The default value of automatic variables and nonclass objects is indeterminate.

Static Storage

Global objects, static data members of a class, namespace variables, and static variables in functions reside in static memory. The address of a static object remains the same throughout the program's execution. Every static object is constructed only once during the lifetime of the program. By default, static data are initialised to binary zeros. Static objects with a *nontrivial constructor* are subsequently initialised by their constructors. Objects with static storage are included in the following examples:

```
int num; //global variables have static storage
int func( )
{
static int calls; //initialised to 0 by default
return ++calls;
}
class C
{
private:
static bool b;
};
namespace NS
{
std::string str; //str has static storage
}
```

Free Store

Free store memory, also called *heap memory* or *dynamic memory*, contains objects and variables that are created by operator new. Objects and variables that are allocated on the free store persist until they are

explicitly released by a subsequent call to operator delete. The memory that is allocated from the free store is not returned to the operating system automatically after the program's termination. Therefore, failing to release memory that was allocated using new generally yields memory leaks. The address of an object that is allocated on the free store is determined at runtime. The initial value of raw storage that is allocated by new is unspecified. Garbage Collection is the term used to describe an algorithm that frees allocated memory when it is no longer in use.

2.2 Problems in Existing System

One aspect of C and C++ programming in that seems to always cause problems for beginners is dynamic memory allocation. If we declare an object within a function, we know it exists within stack memory and that the compiler will delete it when the function leaves scope. However, if we allocate memory from the heap by using malloc (for C programs) or new (for C++ programs), we are responsible for freeing or deleting the object.

In situations where the creation or destruction of an object occurs at consistent, logical places, this isn't a problem. However, if we write a function that contains multiple return statements or uses Exception Handling, destroying these dynamically allocated objects is much more complex.

smart pointer classes can be used to simplify the cleanup of dynamically allocated objects.

Problems in the existing System are :

Object Leakage

Dangling Reference

problems due to explicit deallocation of memory.

Object leakage :

The primary function of the collector is to report objects that are allocated, not deallocated, but are no longer accessible. since the object is

no longer accessible, there in normally no way to deallocate the object at a later time; thus it can safely be assumed that the object has been "leaked".

Dangling Pointers

A common pitfall of regular pointers is the dangling pointer: a pointer that points to on object that is already deleted.

Problems due to Explicit Deallocation of memory

1. Bugs due to errors in storage deallocation are very hard to find, although products are available which can help.

2. In some circumstances the decision about whether to deallocate storage cannot be made by the programmer. Drawing editors and interpreters often suffer from this. The usual result is that the programmer has to write an application-specific garbage collector.

3. An object which is responsible for deallocating storage must be certain that no other object still needs that storage. Thus many modules must co-operate closely. This leads to a tight binding between supposedly independent modules.

4. Libraries with different deallocation strategies are often incompatible, hindering reuse.

5. In order to avoid problems 3 and 4, programmers may end up copying and comparing whole objects rather than just references. This is a particular problem with temporary values produced by C++ overloaded operators.

6. Because keeping track of storage is extra work, programmers often resort to statically allocated arrays. This in turn leads to arbitrary restrictions on input data which can cause failure when the assumptions behind the chosen limits no longer apply.

2.3 Need for Garbage Collection in Object-Oriented Programming

There are several reasons why true object-oriented programming requires garbage collection.

1. Manual Memory Management Breaks Encapsulation :

Program components frequently need knowledge of an entire program to determine the last use of an object and provide deletion. This makes reuse of the component nearly impossible. For example, methods and functions taking a container as an argument need to know of or make assumptions about the rest of the program to determine ownership of the objects in the container.

2. Implementation Hiding is not Compatible with Manual Memory :

Management Implementation hiding is a pillar of object-oriented programming, but explicit memory management requires implementation-dependent low - level knowledge of how memory is structured. For example, programmers often use "copy on write" to efficiently implement pass-by-value semantics. However, to manage memory explicitly, a program has to know if it has a copy of an object or the object itself. Programmers sometimes use reference counting to encapsulate copy-on-write memory management. However, this only works well in simple cases like strings where the data is not polymorphic and does not contain pointers.

3. Message Passing Leads to Dynamic Execution Paths :

Manual memory management must make assumptions about a program's order of execution to make a compile-time determination of the last user of an object. While this is often possible in procedural languages, the object- oriented paradigm of objects sending messages to each other (possibly from different threads) makes it impossible to statically determine the last user of an object. For example, event driven GUI programs frequently have no clear order of execution. Other dynamic control structures, such as exceptions, also make static analysis of memory usage at compile-time impossible.

4. Differing Memory Management Schemes Hinder Reuse :

Because no memory management scheme is universal enough for all applications, manually managed components and libraries often use incompatible memory management schemes. For example, there are common container libraries using each of the following schemes:

a) Doubly specified empty and remove methods with one including a memory delete, placing the memory management burden on the client, who must call the appropriate method.

b) Switches indicating deletion. Many applications must clear the switch to support long-lived data and keep track of ownership of data shared by multiple containers, leaving many memory management issues unaddressed.

c) Value semantics store objects rather than references, inhibiting data sharing and carrying expensive performance costs when complex objects are copied by value.

5. Garbage collection has advanced as rapidly as most computer-related technologies and is now a robust, mature technology. Many object-oriented languages specify garbage collection for all or part of their memory. Garbage collected programs are usually as fast and responsive as their manually managed brethren, of course, garbage collected programs are generally more reliable and easier to develop, maintain, and reuse than manually managed programs.

2.4 Requisites of a Garbage Collector

1. **Generic :** It has to work for any kind of data, so that we do not have to uplicate the code for every class. It behaves as a mother class of all other classes and allows the communication between them.

2. **Automatic :** It has to detect automatically that a memory space is free and delete it automatically. No delete operation is available . This is very important if we do not want to loose memory when a function throws an exception.

3. **Easy to use :** When incorporating a garbage collector to a class we do not need to type any more lines or to over load any function. To allocate memory we do not need to learn a new command and we can use the usual c++ functions.

4. **Compatible with object oriented programming :** When incorporating a garbage collector to a class, it works automatically for its sub classes and it takes into account the difference in size between them.

2.5 The Proposed System

The drawbacks of the present system can be overcomed by developing a Smart Garbage Collector.

Smart Garbage Collector implements new memory management techniques that allow predictable real time performance even for systems that make heavy use of dynamic allocation and rely on the garbage collector to recycle sufficient memory to satisfy all future allocation request in short and bounded time and there by reclaiming unused memory to the memory pool.

The Smart Garbage Collector is implemented using the concept of Smart Pointers. Smart Pointers look, act and feel like pointers but they are smarter. They act as **proxies of ordinary pointers.**

The underlying technique used for smart garbage collector is the **Reference Count Method,** in which whenever a reference is made to the object, the reference count of the object is incremented by 1 and when the object loses its scope, the reference count is decremented by 1. When the reference count is equal to zero, which means there is no more reference to the object and which in-turn means it is no longer needed , so the object could be removed form the memory.

3. Conceptual Perspective

3.1 The idea behind smart pointers :

The Smart Pointer class gives a safe way of memory management which avoids memory leaks and accessing of freed memory. The deallocation of memory takes place automatically when the last pointer to a block of memory disappears. A smart pointer to some class A is a small class that contains a pointer to A. For the smart pointer to A to work, the class A must implement Protect() and UnProtect() methods which do reference counting.. This is true of any class subclassed off RefCntTimeStampMixin, which is intended as the model. Many TargetJr classes do sub-class off RefCntTimeStampMixin, so smart pointers to these classes may be defined. The idea behind the smart pointer is that it keeps track of how many pointers are pointing to a given instance of A, and when there are no more pointers, then the object of class A is deleted. Thus, when an object is no longer accessible it is automatically deleted, but as long as there is still a smart-pointer pointing to the object it will be kept. This reference counting is done by using the reference count implemented in RefCntTimeStampMixin. However, the Protect( ) and UnProtect( ) methods of RefCntTimeStampMixin are not called by the user. They are called automatically whenever an assignment of a pointer or a smart-pointer to a Smart Pointer variable takes place.

3.2 Defining Smart Pointers

Any class that is subclassed off RefCntTimeStampMixin supports smart pointers. To create a Smart Pointer class to a class that is a subclass of RefCntTimeStampMixin, the following is to be done.

```
#include <SomePackage/SomeClass.h>    //
Defines the class SomeClass #include
<Basics/smart_ptr.h> template class
IUE_smart_ptr<SomeClass>;
```

This should be kept in a file IUE_smart_ptr+SomeClass-.C in the directory SomePackage/Templates. Under some compilers it is not actually necessary to instantiate the class, but it should be done so as to accommodate other compilers.

3.3 Typedef alias for IUE_smart_ptr<SomeClass>

For simplicity, it is customary to typedef the template class as follows :

```
typedef IUE_smart_ptr<SomeClass>
SomeClass_ref;
```

This declaration ought most conveniently to be placed in the file SomePackage/SomeClass.h

Smart pointers are used much the same as pointers. Some of the following examples will be for the class IUPoint_ref which is a smart pointer to an IUPoint class.

3.4 Creation of a smart pointer :

Create a null smart pointer :

```
IUPoint_ref apoint;  //Creates a NULL smart
pointer
```

Allocate a smart pointer and allocate memory (various alternatives)

```
IUPoint_ref bpoint = new IUPoint (x, y, z);
IUPoint_ref bpoint (new IUPoint (x, y, z));
```

Create a smart pointer from an existing pointer (various alternatives)

```
// Old style function returning IUPoint *
IUPoint *getRandomIUPoint();
```

IUPoint_ref cpoint (getRandomIUPoint()) ;

IUPoint_ref dpoint = getRandomIUPoint()) ;

Duplicating a smart pointer

```
IUPoint_ref apoint = new IUPoint(x, y, z);
IUPoint_ref bpoint = apoint;
```

apoint and bpoint now both point to the same point.

3.5 Uses and various operation of smart pointers

Assigning smart pointers :

The following work in the way that may be expected.

```
IUPoint_ref apoint;
apoint = new IUPoint (0.5, 0.6, 1.0);
IUPoint_ref bpoint = new IUPoint (2.0, 2.3, 1.0);
apoint = bpoint;
```

Smart Pointers are actually classes instances, but they act as if they were pointers, so the result of the last statement is that apoint and bpoints point to the same object. It is to be noted that the IUPoint originally pointed to by apoint is lost, but its memory is deleted.

We do not have to delete a Smart Pointer.

Smart pointers may also be assigned to ordinary pointers. Use of pointers is discouraged, but sometimes one will use previous code that returns a pointer. In such cases, the right thing is to assign it to a smart pointer :

```
IUPoint *old_function ();
IUPoint_ref apoint = old_function();
```

Deleting Smart Pointers:

One does **not** delete a smart pointer, or the block of memory that it points to. That is done automatically.

Test equality or inequality of pointers :

Smart pointers are tested against others as if they were pointers.

```
apoint = bpoint;
if (apoint == bpoint)      { ... }
```

This will return true. The following statements are also valid

```
if (apoint != bpoint) { ... }
if (apoint == NULL) { ... }
```

It is also possible to test smart pointers against ordinary pointers.

```
IUPoint *old_function(), *another_old_function();
IUPoint_ref apoint = old_function();
if (apoint == another_old_function()) {}
```

This tests whether the two IUPoints *s returned are the same.

Accessing members of the base class:

As far as accessing members of the class is concerned, smart pointers act just like pointers. Thus,

```
IUPoint_ref A = new IUPoint (2.0, 3.0, 1.0);
float x = A->GetX();
```

Lists of smart pointers;

One of the greatest advantages of smart pointers is their use with lists. When the list is deleted or goes out of scope, the objects in this list will also be unreferenced. Thus :

```
{ CoolList<IUPoint_ref> alist;
```

```
for (i=1; i<=10; i++)
{IUPoint_ref newpt = new IUPoint((float)i,1.0. 1.0);
alist.put_end (newpt);        } }
```

When the list alist goes out of scope and is deleted, all the IUPoints in the list will also be deleted, unless they are referenced by another smart pointer.

Passing smart pointers to subroutines.

Smart pointers may be passed to subroutines. There is very little difference between passing by value or passing by reference.

```
void do_some_action (const Image_ref &animage) { }
void do_some_action (Image_ref animage) { }
```

Returning smart pointers from functions ;

Smart pointers may be returned directly from functions. Thus

```
IUPoint_ref getClosestPoint (float x, float y)
{IUPoint_ref apoint;
```

apoint = new IUPoint (x, y, 0,0} // Do some processing to find a point

return apoint; } // Return the point

Calling existing routines that expect a pointer:

Many existing routines in TargetJr expect pointer variables as arguments. For instance :

```
void SomeFunction (Image *animage);
```

In order to call such a function using a smart pointer of type Image_ref, one must get the pointer itself. This is done using the ptr() method of the Smart Pointer class, which returns a pointer to the object referred to by the smart pointer. An example is

```
Image_ref anim = openImage ("somefname");
SomeFunction (animage.ptr());
```

Casting :

To give a concrete example, suppose PerspectiveCamera is a subclass of BasicCamera (which in fact it is). It does not follow that PerspectiveCamera_ref is a subclass of BasicCamera_ref. However, the correct casts occur automatically for this code to compile correctly.

```
PerspectiveCamera_ref pcam = new PerspectiveCamera (...);
BasicCamera_ref bcam = pcam;
```

This is because pcam is automatically cast to a PerspectiveCamera*, which is then used to create a smart pointer of type BasicCamera_ref.

The Unprotect( ) method :

An UnProtect( ) method is supplied for a smart pointer. This is seldom used. It is to remove the protection conferred by the smart pointer. The reference count is decremented, which may result in the memory being deallocated. Though not often done, it is (by design) still possible to reference through an UnProtected Smart Pointer. The user must take

account of the fact that the memory may potentially be deallocated. To avoid this eventuality, it is necessary to know that the block of memory is still being held by some other pointer.

One possible use of Unprotect( ) is to announce that one has finished with this data, even though the Smart Pointer is not going out of scope quite yet, as it is given below :

```
{ Image_ref anim = new Image ();// do some stuff
// with the image anim.
UnProtect(); // do some more stuff that does not
// use the image.}
```

Another use of UnProtect() is to get rid of deadlocks which can (rarely) occur through cyclic references. The user normally does not have to worry about this eventuality in simple programming. The best policy is to avoid using UnProtect ( ) unless we are sure that we need it.

There is no Protect( ) method on Smart Pointers.

General recommendations.

There is very little that we can do wrong with smart pointers. We should never have to delete memory that is assigned to a smart pointer. The only rule of thumb is :

we should not make an explicit assignment to an ordinary pointer. Always assign to smart pointers.

In fact, **do not declare an ordinary pointer :**

```
Image *anim;
```

There is very little reason to do this, since using Image_ref is simpler and safer. The Smart Pointer class is designed to deter us from assigning to standard pointer.

4. System implementation

4.1 Automatically deallocating memory using smart pointers

Smart Pointers

A smart pointer is a stack-based object (an object we create locally in a function) that controls your interaction with a heap-based object (an object we create via new). To create a smart pointer, we create a class that defines (for stack-based objects) the operations we'd normally perform via a pointer.

A smart pointer that implement reference counting has three major task to perform: incrementing the reference count when we initialise a new smart pointer object, decrementing the reference count when we destory a smart pointer object, and deleting the heap object when the reference count reaches zero. (The reference count will be zero when the last smart pointer for the heap object leaves the current scope. Seamlessly smart pointers can replace raw pointers. The ideal is for client code not to care whether it is using raw pointers or smart pointers.

For example, if we want to create a smart pointer class for accessing character strings, we'll need to overload the operators we typically use with character pointers: for example, [] and =. (Obviously, the Borland C++ String class implements much of this functionality, and we'll probably want

to use it for any production code. The example demonstrate a simple smart pointer class.)

In memory, each smart pointer object will have a corresponding heap object. However, since the smart pointer class overloads the standard pointer operators, the class provides us to manipulate the heap object indirectly by passing those operator function calls to the corresponding heap object.

The smart pointer object will pass all pointer operator calls to its heap object.

Because C++ programs always destroy stack-based objects when the enclosing function exits, we can control the allocation and deallocation of the character strings from the smart pointer's constructor and destructor. Example A contains a simple program that fails to delete the variable aptr if the user calls the program with no arguments.

Example A - If a function contains multiple return statements, memory leaks can become difficult to find.

```
#include <iostream.h>
#include <string.h>
int main(int argc, char* argv[])
{
char* aptr = new char[80];
strcpy(aptr, argv[0]);
char* bptr = 0;
char* cptr = 0;
 if(argc >= 2)
{
 bptr = new char[strlen(argv[1]) + 1];
 strcpy(bptr, argv[1]);
```

```
}
if(argc >= 3)
{
 cptr = new char[strlen(argv[2]) + 1];
 strcpy(cptr, argv[2]);
}
cout << "Program - " << aptr << endl;
switch(argc)
{
case 1:
cout << "No arguments" << endl;
return 1;

case 3:
cout << "Argument 2 - " << bptr << endl;
default:
cout << "Argument 1 - " << aptr << endl;
}
delete aptr;
delete bptr;
delete cptr;
return 0;
}
```

To avoid the memory leak in this program, we need to add the line
delete aptr;
immediately above the statement
return 1;

Unfortunately, if we add additional return statements, we need to add the same line, and possibly others, in order to delete one of the other character strings. The following program is rewritten using a simple smart character pointer class to delete the dynamically allocated memory.

Example B: smart.cpp

```
#include <iostream.h>
#include <string.h>
class smartCharPtr
{
char* ptr;
public:
smartCharPtr( )
```

```
{
 ptr = new char[1];
ptr[0] = '\0';
}
smartCharPtr(char* p) : ptr(p)
{ }
~smartCharPtr( )
{
cout << "deleting smartCharPtr for ";
if (ptr)
cout << ptr;
else
cout << " -";
cout << endl;
delete [] ptr;
}
operator char*()       // for strcpy()
{  return ptr; }
operator const char*()  // for cout
{  return ptr; }
smartCharPtr& operator=(const char* newPtr)
{
delete [] ptr;
ptr = new char[strlen(newPtr) + 1];
trcpy(ptr, newPtr);
return *this;
}
};
int main(int argc, char* argv[])
{
// store program name (argv[0])
smartCharPtr aptr(new char[80]);
strcpy(aptr, argv[0]);
// set up two other pointers
smartCharPtr bptr = 0;
smartCharPtr cptr = 0;
// give program name
cout << "Program - " << aptr << endl;
// save others (if present)
if (argc > 1) {
bptr = argv[1];
cout << "Argument 1 = " << bptr << endl;
}
if (argc > 2)
{
cptr = argv[2];
```

```
cout << "Argument 2 = " << cptr << endl;
}
// if no args, get out
if (argc == 1) {
cout << "No Arguments" << endl;
return 1;
}
return 0;
}
```

Description :

In the smartCharPtr class, we provide two constructors, a destructor, two conversion operators, and an overloaded assignment operator (=) function.

The constructors allow us to build a smartCharPtr object with or without the corresponding pointer. However, we will notice that if we create a smartCharPtr object without the pointer, we initialize the ptr data member to 0. This allows us to safely call delete on this pointer at a later time.

In the destructor, we first display a message that includes the character string (if the pointer is valid). Then, we delete the ptr data member pointer.

Next, we provide two conversion operator functions. The operator char*() function allows us to pass a smartCharPtr object to a function that's expecting a char pointer. The operator const char*() function performs the same task for functions that expect a const char pointer.

Finally, we overload the assignment operator with the operator=() function. The compiler will call this function whenever we try to initialize a smartCharPtr object with a char pointer.

In the body of the main() function, we simply added the smartCharPtr objects in place of the char pointers and changed the associated initialization code. Not only do these changes eliminate the memory leak that existed before, they also make the flow of the function easier to follow.

Results of the Program Execution :

```
Program - C:\BORLANDC\SMART.EXE
No arguments
deleting smartCharPtr for -
deleting smartCharPtr for -
deleting smartCharPtr forÂ C:\BORLANDC\SMART.EXE


Arguments Entered

1 2

Program - C:\BORLANDC\SMART.EXE
Argument 1 - 1
Argument 2 - 2
deleting smartCharPtr for - 2
deleting smartCharPtr for - 1
deleting smartCharPtr forÂ C:\BORLANDC\SMART.EXE
```

The program deletes the smartCharPtr objects in reverse order. The program does this because it places (or *pushes*) variables on the program

stack as we create them and then removes (or *pops*) them off the program stack in a first-in-last-out manner.

Other enhancements

The above example describes the Usage of smart pointer to prevent memory leaks when we exit a function that uses heap objects,which is an important use of smart pointer classes.

One simple change we could make to the class is adding code that lets us to use standard pointer syntax on the smart pointer object. For example, if we add the function

```
smarCharPtr& operator *()
{
 return *ptr;
}
```

we can apply the dereference operator (*) to the smart pointer (even though it's a local object) in order to return the dereferenced pointer's value.

Templates make it simple

The above example explains a *very* simplistic smart pointer class for controlling the memory allocation for character pointers. However, for smart pointer classes to be really useful, we need to write a smart pointer template class.

By writing a generic smart pointer class as a template class, we'll be able to easily reuse the overloaded versions of many standard pointer operators (=, &, *, and so on).

Templates and Smart Pointers

C++ allows us to create "smart pointer" classes that encapsulate pointers and override pointer operators to add new functionality to pointer operations. Templates allow us to create generic wrappers to encapsulate pointers of almost any type.

The following code outlines a simple reference count garbage collector. The template class Ptr<T> implements a garbage collecting pointer to any class derived from RefCount.

```
#include <stdio.h>
#define TRACE printf
class RefCount {
int crefs;
public:
RefCount(void) { crefs = 0; }
~RefCount() { TRACE("goodbye(%d)\n", crefs); }
void upcount(void) { ++crefs; TRACE("up to %d\n", crefs);}
void downcount(void)
{
if (--crefs == 0)
{
delete this;
}
else
TRACE("downto %d\n", crefs);
}
};
class Sample : public RefCount {
public:
void doSomething(void) { TRACE("Did something\n");}
};
template <class T> class Ptr {
T* p;
public:
Ptr(T* p_) : p(p_) { p->upcount(); }
~Ptr(void) { p->downcount(); }
operator T*(void) { return p; }
```

```
T& operator*(void) { return *p; }
T* operator->(void) { return p; }
Ptr& operator=(Ptr<T> &p_)
{return operator=((T *) p_);}
Ptr& operator=(T* p_) {
p->downcount(); p = p_; p->upcount(); return *this;
}
};
int main() {
Ptr<Sample> p  = new Sample; // sample #1
Ptr<Sample> p2 = new Sample; // sample #2
p = p2; // #1 will have 0 crefs, so it is destroyed;
// #2 will have 2 crefs.
p->doSomething();
return 0;
// As p2 and p go out of scope, their destructors call
// downcount. The cref variable of #2 goes to 0, so #2 is
// destroyed
}
```

Classes `RefCount` and `Ptr<T>` together provide a simple garbage collection solution for any class that can afford the `int` per instance overhead to inherit from `RefCount`. Note that the primary benefit of using a parametric class like `Ptr<T>` instead of a more generic class like `Ptr` is that the former is completely type-safe. The preceding code ensures that a `Ptr<T>` can be used almost anywhere a `T*` is used; in contrast, a generic `Ptr` would only provide implicit conversions to `void*`.

For example, consider a class used to create and manipulate garbage collected files, symbols, strings, and so forth. From the class template `Ptr<T>`, the compiler will create template classes `Ptr<File>`, `Ptr<Symbol>`, `Ptr<String>`, and so on, and their member functions: `Ptr<File>::~Ptr()`, `Ptr<File>::operator File*()`,

`Ptr<String>::~Ptr()`, `Ptr<String>::operator String*()`, and so on.

Conclusion :

Designing programs that allocate and deallocate dynamic memory correctly can be an annoying and error-prone undertaking. By using smart pointers to automate these tasks, we can avoid memory leaks that would otherwise be difficult to overcome.

4.2 Implementation using reference count method

The underlying technique used in automatic deallocation of memory in Smart Garbage Collector is Reference Count Method :

Reference count method :

Reference counting is technique in which a smart pointer class checks to see if it's the last smart pointer referencing a heap object before it deletes the heap object.

Implementation of Reference Count Method using Smart Pointers

A pointer is a variable which holds the address of some memory location. A more interesting way to define anything in programming is by its attributes and actions. A pointer has the attribute of holding an address, and has several actions: (assumes: *int* p*)

Action	Example
Dereferencing	`int b=*p;`
Member access	`p->something`
NULL check	`if (p) ....`

A smart pointer provides all of the above requirements, and will make sure to call the *addRef()*, *subRef()* .

When the reference counting should be changed for an object. When ever someone acquires a pointer to the object, the counter should be

incremented. When ever someone lets go of a pointer to the object, the counter should be decremented.

The last one to let go of the pointer should also delete the object. Using C++ features, here is a template class for such a smart pointer. It assumes the template parameter provides the reference counting operations.

```
template<class T>
class RefCntPointer
{
public:
/** Construct from normal pointer, default to NULL */
RefCntPointer(T* ptr=0) : m_ptr(ptr) { addRef(); }
/** Construct from another smart pointer.    Copy
Constructor */
RefCntPointer(const RefCntPointer& p) : m_ptr(p.m_ptr)
{ addRef(); }
/** Destructor. */
~RefCntPointer() { subRef(); }
```

The constructors store the pointers and call addRef() to increment the counter, due to this new pointer. The destructor cleans up the counter. This is obviously not enough, since the smart pointer can be changed by assignment

```
/** Assignment operator. */
RefCntPointer& operator= (const RefCntPointer& p)
{
// Use the other operator (code reuse)
return *this = p.m_ptr;
}
/** Assignment operator. */
RefCntPointer& operator= (T* ptr)
{
 if (m_ptr != ptr)
 {
 subRef();
 m_ptr=ptr;
```

```
addRef();
}
return *this;
}
```

It is to be noted that the operator first decrements the counter for the old pointer, and then assigns the new pointer and increments its counter.

Several operators to provide the pointer actions, and some useful conversions are

```
/** Dereferencing operator. Provided to behave like the
normal pointer. */
T& operator *       ( ) const { return *m_ptr; }
/** Member access operator. Provided to behave like the
normal pointer. */
T* operator ->      ( ) const { return m_ptr; }
/** Conversion operators */
operator T*       ( ) const { return m_ptr; }
operator const T* ( ) const { return m_ptr; }
/** boolean test for NULL */
operator bool     ( ) const { return m_ptr!=0; }
/** Address of pointer.  May cause memory leaks if pointer
is modified. */
T** operator &      ( )       { return &m_ptr; }
```

The last operator, is usually removed to avoid problems.

The implementation of the private part of the class, which modifies the counter is given below :

```
private:
void addRef( )
{
// Only change if non-null
if (m_ptr) m_ptr->addRef( );
}
void subRef( )
```

```
{
// Only change if non-null
if (m_ptr)
{
// Subtract and test if this was the last pointer.
if (m_ptr->subRef( ))
{
delete m_ptr;
m_ptr=0;
}
}
}
T* m_ptr;
};
```

Usage :

Using these pointers means simply to replace the declaration of the pointer:

MyObject* p; --> RefCntPointer<MyObject> p;

All the rest of the code can remain the same, as this type provides the same semantics as a regular pointer.

Reference Counting :

Reference Counting is used to handle shared objects with proper release when the object is no longer needed. This is done by attaching a counter to the object, which holds the number of places that hold a pointer to the object. The user code is responsible for calling *AddRef()* and *Release( )* to increment and decrement the counter. When the counter reaches 0, the object is deleted.

Any one who has ever used this method knows that it can be error prone if the *AddRef( )* and *Release( )* functions are not called properly, resulting in either memory leaks or invalid pointers. Later in this article, a way to solve this problem will be presented.

To provide the actual reference counting, a base class which handles the counter is created. All this class provides is an *addRef( )* function which increments the counter, a *subRef( )* function which decrements the counter and returns *true* if the counter is 0.

All objects we want to include in this allocation scheme, must be instances of a class derived from this base class. It doesn't have to be a direct parent, just an ancestor, so that the object will have the reference counter available.

```
class RefCntObject
{
public:
virtual ~RefCntObject( ) {}
/** Default constructor.  Initial reference count is 0,
and will be incremented as soon as the object is
pointed to. */
```

```
RefCntObject() : m_refcnt(0) {}
/** Add 1 to the reference count. */
void addRef( ) { m_refcnt++; }
/** Subtract 1 from the reference count.
Returns true if the reference count has reached 0
and the object should be deleted. */
bool subRef( ) { return (--m_refcnt <= 0); }
private:
int m_refcnt;
};
```

Program for reference counting :

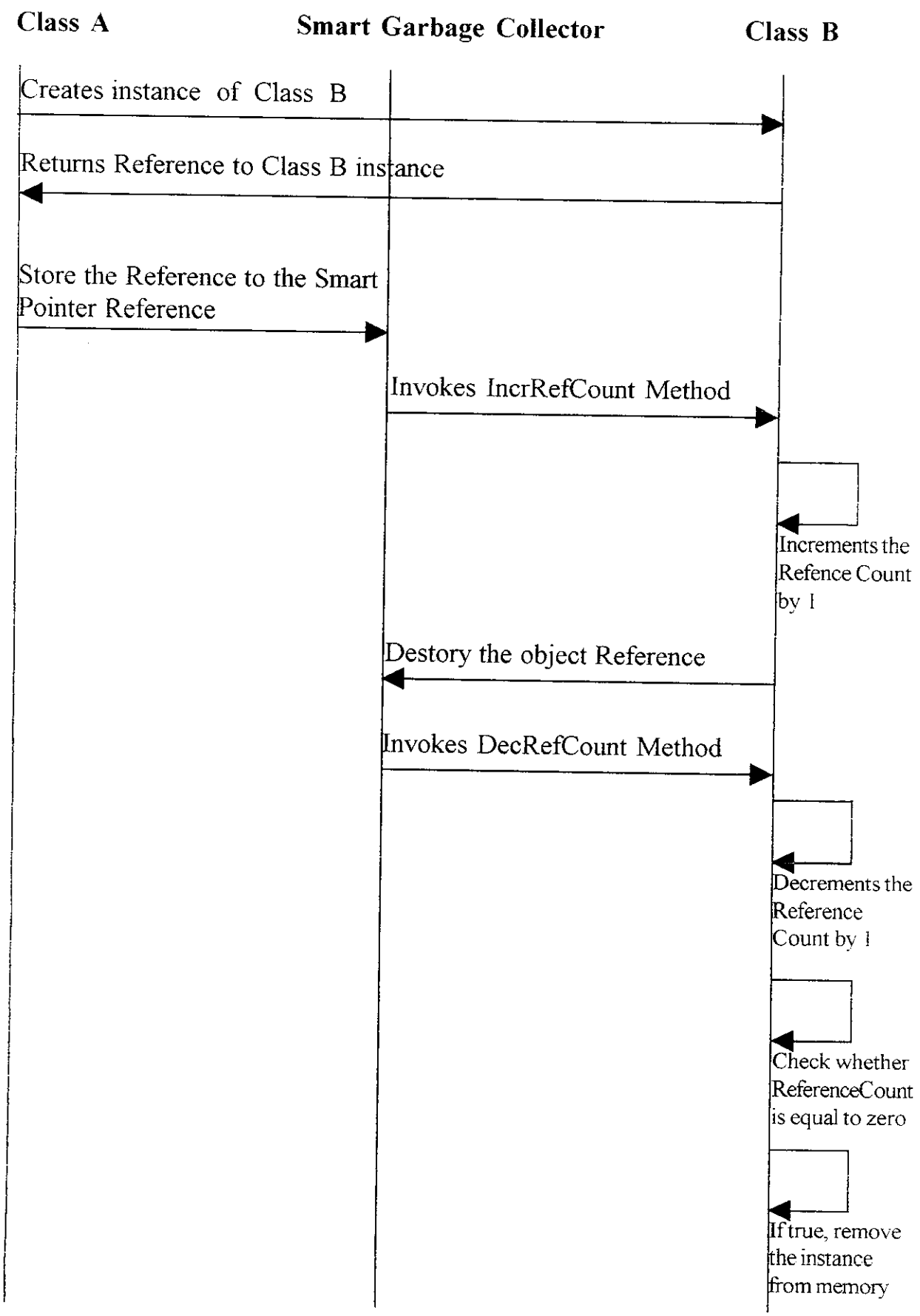
```
#ifndef SmartPointer_h
#define SmartPointer_h

template<class T> class SmartPointer
{
public:
        ///
        // Constructor.
        // This is default constructor pointing to NULL object.
        //
        SmartPointer() : m_Ptr(0)
        {
        }
        ///
        // Constructor.
        // This points to the given object.
        //
        SmartPointer(T*       t) : m_Ptr(t)
        {
                if (m_Ptr != (T*) 0)
                {
                        m_Ptr->IncrRefCount();
                }
        }
        SmartPointer(const SmartPointer<T> &rhs)
        {
        m_Ptr = (T *)rhs;
        if (m_Ptr != (T*) 0)
        {
        m_Ptr->IncrRefCount();
        }
        }
```

```
///
// Copy Constructor.
// This is declared as a template, so that SmartPointer<Derived>
// could be passed as an argument to a function that expects
// a SmartPointer<Base>
//
template<class T2>
SmartPointer(SmartPointer<T2>& rhs)
{
m_Ptr = (T2 *)rhs;
if (m_Ptr != (T2*) 0)
{
m_Ptr->IncrRefCount();
}
}
///
// Destructor
//
virtual ~SmartPointer(void)
{
if (m_Ptr != (T*) 0)
{
m_Ptr->DecrRefCount();
}
}
///
// Access to embedded object pointer
//
operator T* (void) const
{
return m_Ptr;
}
///
// Access to embedded object reference
//
T& operator* (void) const
{
return (*m_Ptr);
}
///
// Overloaded operator to make this object behave just like a regular
// c++ pointer
//
T* operator-> (void) const
{
return m_Ptr;
```

```
}
///
// Assignment operator . Assigning a new object on memory.
//
SmartPointer& operator = (T* _rhs)
{
if (_rhs == m_Ptr)
{
// The user is trying to assign the same object.
// Since we already have that refernce just say OK.
return *this;
}
// We do not need refernce to our object anymore.
if (m_Ptr != (T*) 0)
{
m_Ptr->DecrRefCount();
}
// This is our new object
m_Ptr = _rhs;
if (m_Ptr != (T*) 0)
{
// We are pointing to valid object in memory
m_Ptr->IncrRefCount();
}
return *this;
}
///
// Assignment operator. Assigning a new object using the reference
//
SmartPointer& operator = (const SmartPointer& rhs)
{
if (m_Ptr == (T*)rhs)
{
// The two reference are the same
return *this;
}
// We do not refernce to our object anymore.
if (m_Ptr != (T*) 0)
{
m_Ptr->DecrRefCount();
}
// This is our new object
m_Ptr = (T*)rhs;
if (m_Ptr != (T*) 0)
{
// We are pointing to valid object in memory
```

```
 m_Ptr->IncrRefCount();
 }
 return *this;
 }
 protected:
 ///
 // The pointer to the object that this object refers to on heap
 //
 T*       m_Ptr;
 };
 template <class T>
inline bool operator<(SmartPointer<T>&lhs,SmartPointer<T> &rhs)
 {
 return( (T*) lhs < (T*) rhs);
 }
 template <class T>
 inline bool operator== (const SmartPointer<T>& lhs,const
 SmartPointer<T>& rhs)
 {
 return  ( ((T*) lhs) == ((T*) rhs));
 }
#endif SmartPointer_h
```

4.3 Sequential Flow Diagram



Description :

The Smart Pointer is nothing but normal pointer with intelligence. It helps the programmers to avoid all the problems faced by them while using pointers like dangling references and memory leakage. The smart pointer can be used just like normal pointers. There is no difference in their usage. The smart pointer keeps track of the reference to an object. Whenever a new reference created to a object, the reference count of that object is incremented. When the reference of the smart pointer object goes out of scope, the smart pointer takes care of decrementing the reference count of that object and also checks whether the reference count = 0. If it is true then the object is deleted. So this type of reference count algorithm make sure that the object is not deleted, when there is reference pointing to that object. Since the object deletion is taken care of the smart pointer, the programmers are relieved from deleting the objects which they have been created using new operator. This avoids memory leakage and dangling references, thereby providing effective memory management.

Dealing with STL

Introduction STL :

The Standard Template Library, or STL, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its component are heavily parameterized: almost every component in the STL is a template.

STL consists of five main components.

They are :

Algorithm : Computational procedure that is able to work on different containers

Container : Object that is able to keep and administer objects

Iterator : Abstraction of the algorithm-access to containers so that an algorithm is able to work on different containers Function

Object : A class that has the function-call operator (operator( )) defined.

Adaptor : Encapsulates a component to provide another interface (e.g. make a stack out of a list)

STL is designed to be generic and efficient (does not incur time overload compared to alternatives). To achieve these two design goals ,STL containers store their objects by value . This means that if we have an STL container that stores objects of class Base, it cannot store of objects of classes derived from Base.

```
Class Base{ /* ..... */ };
Class Derived: public Base { / * .... * / } ;
Base b;
Derived d;
Vector <Base> v;
v.push_ back (b);       //ok
v.push_ back (d);       //error
```

If we need a collection of objects from different classes The simplest solution is to have a collection of pointers :

```
Vector <Base*> v;
v.push_ back (new Base);      //ok
v.push_ back (new Derived); // ok too
// cleanup:
for (vector <Base*>:: iterator I = v.begin( ); I ! = v.end( );
++I )
delete *I ;
```

The problem with this solution is that after we have done with the container, we need to manually cleanup the objects stored in it . This is both error prone and not exception safe .

Smart pointers are a possible solution , as illustrated below .(Alternative solution is a smart container , like the one implemented in Pointainer.h)

```
Vector <linked_ptr<Base> > v;
v.push_ back (new Base);      //ok
v.push_ back (new Derived); // ok too
// cleanup is automatic
```

Since the smart pointer automatically cleans up after itself, there is no need to manually delete the pointed objects.

It is to be noted that STL containers may copy and delete their elements behind the scenes (for example when they resize themselves). Therefore, all copies of an element must be equivalent , or the wrong copy may be the one to survive all this copying and deleting. This means that

some smart pointers cannot be used within STL containers, specifically the standard auto _ ptr and any ownership – transferring pointer.

Program : Implementation of STL Using smart pointers :

```
#include <iostream>
#include <list>
Human  h ("Male");
Student s ("123", "PhD", "Female");
lh.push_back(h);  // Base class object
lh.push_back(s);  // Derived class object
list<Human>::iterator i = lh.begin( );
while (i != lh.end( ))
{
Human t_h = *i;
t_h.print( );
i++;
}
}
///
// Solved
//
#include <iostream >
#include <string>
#include <list>
using namespace std;
class Human
{
public :
Human(string a_Gender)
{
m_Gender = a_Gender;
}
virtual void print( )
{
cout << " Human: Gender = " << m_Gender << endl;
}
private :
string m_Gender;
};
class Student : public Human
{
public:
Student(string a_Regno, string a_Course, string a_Gender)
: Human(a_Gender)
```

```
{
m_Regno  = a_Regno;
m_Course = a_Course;
}
void print()
{
cout << "Student: Regno = " << m_Regno
<< " Course = " << m_Course;
Human::print();
}
private:
string m_ Regno, m_Course;
};
void main ()
{
list<Human*>lh;
Human  *h = new Human("Male");
Student *s = new Student("123", "PhD", "Female");
lh.push_back(h);  // Base class object
lh.push_back(s);  // Derived class object
list<Human>::iterator i = lh.begin();
while (i != lh.end())
{
Human *t_h = *i;
t_h->print();
i++;
}
}
```

Auto Pointer Problem:

The present c++ standard is providing a lot of problems while using auto pointers. Those problems can be visualised by using the compilers based on the latest c++ standard .Some of the compiler which supports latest c++ standard are:

1. SUN's C++
2. SGI's C++
3. Latest GNU Compiler

Draw Backs while using Auto Pointers:

1. Reference count is not maintained.
2. It can be a reference to only one object .
3. when you assign a object x which contains Auto pointer to another object y of same type the auto pointer reference of the object x is equated to zero, according to present C++ standard, when we access the auto pointer object it will crash the system .

This problem is due to the non maintenance of reference count in Auto pointers. In our Smart Pointer we are maintaining the Reference Count so the problem will not incur .Thus we prove Smart Pointer is better than normal and Auto pointer

Description :

1. In our System, the Application is the Root Cause of invoking or triggering the Smart Garbage Collector. Thereby the Smart garbage Collector is triggered only whenever there is a Reference to the Garbage Collector thereby Reducing System Resources & Cost involved.
2. When a ClassA creates an instance of ClassB, the Reference to the ClassB instance is Returned.
3. The Reference of ClassB Instance is stored in Smart Pointer Reference.
4. This Smart Pointer Reference invokes the Smart Garbage Collector.
5. Whenever there is a Reference to the instance of ClassB, the Smart Garbage Collector invokes the IncrRefCount Method which inturn increments the Reference Count of the Object by 1.
6. When the object Reference is destroyed, again the Garbage Collector is invoked / Triggered and in turn decrements the Reference count of the object by 1.
7. After Decrementing the Reference Count, Smart Garbage Collector checks whether Reference Count = 0. If it is a true it removes the instance from memory, since the object is no longer needed and the memory is reclaimed.
8. Thereby the Garbage Collector Rectifies / Alleviates the problem of memory leakage and Dangling References.

5. A Sample Application Implemented with Smart Garbage Collector

Machine Shop Simulation :

Description

A machine shop (or factory or plant) comprises m machines or work stations. The machine shop works on jobs, and each job comprises several tasks. Each machine can process one task of one job at any time, and different machines processing the task until the task completes.

A Sheet metal plant might have one machine (or station) for each of the following tasks: design; cut the sheet metal to size; drill holes; cut holes; trim edges; shape the metal;

seal seams. Each of these machine begins /stations can work on one task at a time.

Each job includes several tasks. For example, to fabricate the heating and air-conditioning ducts for a new house, we would need to spend some time in the design phase, and then some time cutting the sheet metal stock to the right size pieces. We need to drill or cut the holes (depending on their size), shape the cut pieces into ducts, seal the seams, and trim any rough edges.

For each task of a job, there is a task time (i.e., how long does it take) and a machine on which it is to be performed in a specified order. So a job goes first to the machine for its first task,. When this first is complete, the job goes to the machine for its second task, and so on until its last task completes. When a job arrives at a machine, it may have to wait

because the machine might be busy. In fact, several jobs might already waiting for that machine.

Each a machine in our machine shop can be one of the three states: Active, idle and changeover in the active state the machine is working on a task of some job; in the idle state it is doing nothing; and in the changeover state the machine has completed a task and work needed to prepare it for a new task is in progress. In the changeover state, the machine operator might, for example, clean the machine, put away tools used for the last task, and take a break. The time each machine must spend in its changeover state varies from machine to machine.

When a machine becomes available for a new job, it will need to pick one of the waiting jobs to work on. In our machine shop each machine serves its waiting jobs in a F I F O manner, and so the waiting jobs at each machine form a (F I F O) queue . In other machine shops the next job may be selected by priority. Each job has a priority associated with it. When a machine becomes free, the waiting job with highest priority is selected.

The time at which a jobs last task completes is called its **Finished Time**. The length of a job is the some of its task times. If a job of length l arrives at the machine shop at time 0 and completes at time f , then it must have spend exactly f – l amount of time waiting in machine queues. To keep customers happy, it is desirable to minimise the time a job spends waiting in machine queues.

Machine shop performance can be improved if we know how much time job spend waiting and which machine are contributing most to this wait.

When simulating a machine shop, we follow the jobs from machine to machine without physically performing the tasks. We simulate time by using a simulating clock that is advanced each time it task completes or a new job enters the machine shop. As tasks completes , new tasks or scheduled. Each time a task completes or a new job enters the shop, we say that an event has accrued. In addition, a **Start Event** initiates the simulation . The following example illustrates how the simulation works when known new jobs enter the machine shop during the simulation.

Example : Consider a machine shop that has m=3 machines and n=6 jobs. We assume that all 6 are available at time 0 and that no new jobs become available during the simulation. The simulation will continues until all jobs have completed.

The 3 machines, M1, M2 and M3 have a changeover time of 2, 0, and 1, respectively. So when a task completes, the machine one must wait two time units before starting another, machine 2 can start the next task immediately, and machine 3 must wait one time unit. The figure(1) gives the characteristics of 6 jobs. Job one for example has 3 task. Each task is specified as a pair of the form (machine, time). The first task of job 1 is to be done on M1 and takes 2 time Units, the second is to be done on M2 and takes 4 time units, the 3rd is to be done on M1 and takes one time Unit.

The job lengths (i.e. is the sum of their task times) or 7, 6, 8 and 4, respectively

Figure(2) shows the machine shop simulation. Initially, the 4 jobs are placed into queues corresponding to their first tasks. The first task for jobs 1 and 3 are to be done on M1, so these jobs are placed on the queue for M1. The first tasks for jobs 2 and 4 are to be done on M3. Consequently, these jobs begin on the queue for M3. The queue for M2 is empty at the start all 3 machines are idle. We use the symbol I to indicate that the machines have no active job at this time. Since no machine is active, the time at which they will finish their current active tasks is undefined and denoted by the symbol L (large time) .

The simulation begins at time 0. I.e. the first event, the start event, occurs at time 0. At this time the first job in each machine queue is scheduled on the corresponding machine. Job 1s first task is scheduled on M1 , and job 2s first task on M3. The queue for M1 now contains job 3 only, while that for M3 contains job 4 only. The queue for M2 remains empty. The job 1 become the active job on M1, and job 2 the active job on M3. M2 remains idle. The finish time for M1 becomes 2 (current times of 0 + task time of 2) While the finish for M3 becomes 4.

The next event occurs at time 2. This time is determined by finding the minimum Of the machine finish times. At time 2 M1 completes its active task. This task is a job 1 task. Job 1 is moved to machine M2 for its next task. Since M2, is idle, the processing of job 1 is moved to machine

M2 for its next task. Since M2 is idle, the processing of job 1's second task begins immediately.

This task will complete at time 6(current time of 2, plus task time of 4).M1 goes into its change – over state and will remain in this state for two time units. Its active job is set to C(change over), and its finish time to 4. At time 4 both M1 and M3 complete their active tasks. As M1 completes a change – over Tasks , it begins a new job. From its queue, the first job, job3, is selected. The task length is 4. So the task will complete at time 8.

This time becomes the finish time for M1. M3's active job , job 2, needs to be serviced next on M1. M1 is busy, so the job is added to M1'S job queue .M3 moves into its change – over state and completes this change-over task at time 5. Remaining sequence of events follows as per the above explanation.

Jobs 2 & 4 finish at time 12, job 1 finishes at time 15, and job 3 , finishes at time 19, since the length of job 2 is 6 and its finished time 12, job 2 must have spent a total of 12-6=6 time units waiting in machine queues. Similarly job 4 must have spent 12-4=8 time units waiting in queues. The wait times for jobs 1 & 3 are 8 & 11 respectively.

The total wait time is 33. We may determine the distribution of these 33 units of wait time across the 3 machines. For example job4 joined the queue for M3 at time 0 and did not become active until time 5. So this job waited at M3 for 5 time units. No other job experience a wait at M3.

The total wait time at M3 was, therefore 5 times unit. Going through the figure 2 we can compute wait times for M1 & M2. The numbers 18 and 10 respectively. As expected, the sum of job wait times (33) equals the sum of machine wait times.

High Level Simulated Design

In designing our simulator, we shall assume that all jobs are available initially (i.e. no jobs enter the shops during the simulation). Further, we assume that the simulation to run until all jobs complete. Since the simulator is a fairly complex program , we break it into modules. The tasks to be performed by the simulator are input the data and put the jobs into the queues for their first tasks ; perform the start event (i.e. do the initial loading of jobs on to the machines); run through all the events (i.e. perform the actual simulation); and output the machine wait times. We shall have one C++ function for each task. Each function may through exceptions suggest NoMem and Bad Input (invalid in put data). As a result, our main function takes the form of the program 1.The **Catch Block** could be replaced by several catch blocks, one for each kind of exception that may be thrown and a different message output each.

The Class Task

Before we can develop the code for the 4 functions invoked by program 1, we must develop representation for the data objects that are needed. These objects include tasks, jobs, machines, and an event list. Each task has two components: **Machine** (The machine on which it is to be performed) and the **Time** (the time needed to complete the task).

Program 2 gives the class definition. Since the machines are assumed to be numbered 1 through M, machine of type int (type unsigned may be used instead). We shall assume that all times are integral. The data type of time is declared as long to permit very long simulation. The class **Task** has two friends; the **Class Job** and the function **MoveToNextMachine**. Task and job have been declared friends because they need access to the private members of task. We may avoid granting this access by defining public member functions of task to set and retrieve the values of the **Machine** and **Time** components.

The Class Job :

Each job has a list of associated task that are performed in list order. Consequently , the task list may be represented as a queue Task Q. To determine the total wait time experience by a job, we need to know its length and finish time. The finish time determine the event clock, while the length is the sum of task times. To determine a job length, we associate a private data member **Length** with it.

The program gives the class Job. **Task Q** has been defined as a linked queue of a tasks. since the number of task associated with a job is known at time of input we could have defined Task Q to be a pointer to a dynamically constructed formula based queue just large enough to hold the desired number of tasks. This definition would work quite well for our limited simulator, and we assume no jobs enter the shop after the simulation begins. In a more general simulator, jobs are permitted to enter the job during simulation. When linked queues are used, task queue notes may be freed as soon as a task completes. These free nodes may be reused to construct the task queue for new jobs. When formula based queues are used, the task queues space may be freed only when the entire job has completed. Consequently the simulation may fail for lack of sufficient space, even though very few unfinished task remain in this shop.

The private member **ArriveTime** records a time at which the job enters its current machine queue and determines the time the job waits in this queue. The job identifier

Is stores in ID and is used only when outputting the total wait time encounted by this job.

The public member **AddTask** Adds a task to the jobs task queue. The task is to be performed on a machine p and takes t time. This function is used only during data input. The public member **DeleteTask** is used when a job is moved from a machine queue to active status.

At this time its first task is deleted from the task queue (the task queue maintains a list of task yet to be schedule on machines), the job length is incremented by the task time, and the task time returned. Length becomes equal to the job length when we schedule the last task for the job.

The Class Machine:

Each machine has a changeover time, an active job, and a queue of waiting jobs. Since a job is a rather large object (it has a task queue and other data members), it is more efficient to define the queue of waiting jobs as a queue of pointers to jobs. Therefore, each machine queue operation deals with pointers (which are small objects) rather than with jobs.

Since each job can be in almost one machine queue at any time, the total space needed for all queues is bounded by the number of jobs. However, the distribution of jobs over the machine queues changes as a simulation proceeds. It is possible to have a few very long queues at one time. These queues might become very short later, and some other queues become long. If we use formula based queues, we must declare the queue size for each machine at its maximum possible value. (Alternatively, we must dynamically adjust the size of the array that holds the queue). As a result, we need a space for m*(n+1) job pointers (m is the number of machines and n number of jobs). When we use linked queues, we needs space for n job pointers and linked fields, so we use linked queues.

Program 3 use the **class Machine.** The private members **JobQ, ChangeTime, TotalWait, NumTask, and Active**, respectively, denote queue of pointers to waiting jobs, the change over time of the machine, the total time jobs have spent waiting at this machine the number of task processed by the machine and a pointer to the currently active job. The active job pointer is zero whenever the machine is idle or in its changeover state.

The public member **IsEmpty** returns to if the job queue is empty: The public member **AddJob** adds a job to the queue of waiting jobs; **SetChange** sets the value of **ChangeTime** (changeover time) during input; and **Stats** returns the total wait time experienced at this machine as well as the number of task processed.

We store the finish times of all machines in an event list. To go from one event to the next we need to determine the minimum of the machine finish times. Our simulator also needs an operation that sets the finish time for a particular machine. This operation has to be done each time a new job is scheduled on a machine. When a machine becomes idle, its finish time is set to some large number.

The class EventList :

The program gives the **class EventList** that implements event list as a one dimensional array **FinishTime**, with Finish Time[p] being the finish time of the machine p. When the machine shop is initialised all machine

are idle and their finish time is set to Big T. For correct operation, the simulation should complete before time Big T. The number of machines in the shop is recorded in **NumMachines.**

The function **NextEvent** returns, in p, the machine that completes its active task next, and in t , the time at which the task completes for an machine shop, it takes O (m) time to find the minimum of the finish times, so the complexity of Next Event is O(m). The function to set the finish time of machine of a machine, **SetFinishTime**, runs in O(1)time. When we use 2 data structures- heaps and left ist trees the complexity of both Next Event and SetFinishTime becomes O(logm). If the total number of task across the jobs is T, then our simulator will invoke Next Event and SetFinishTimeO(T) times each. Using the event list implementation of program 4, these invocation takes O(Tm) time; using one of these data structures , the invocations take O(Tlogm) time. Eventhough the data structures are more complex, they result in a faster simulation when m suitably large.

Global Variables

Our code for the 4 functions of the program 1 uses the global variables defined in program file. The significance of most of these variables is self evident. **Now** is the simulated clock and records the current time. Each time an event occurs, it is updated to the event time . **LargeTime** is a time that exceeds the finish time of the last job and denotes the finish time of an idle machine.

The Function Input Data

The code for the function **InputData** begins by inputting the number of machines and jobs in the shop. Next, we create the initial event list *EL,with finish times equal to **LargeTime** for each machine, and the array M of machines. Then , we input the change-over times for the machines. Next , we input the jobs one by one. For each job we first input the number of task it has, and then we input the task as pairs of the form (machine, time). The machine for the first task of the job is recorded in the variable p. When all tasks of a job have been input, the job (actually a pointer to the job) is added to the queue for its first tasks machine.

The Functions StartShop and ChangeState.

To start the simulation, we need to move the first job from each machines job queue to the machine and commence processing. Since each machine is initialized in its idle state, we perform the initial loading in

the same way as we change a machine from its idle state, which may happen during simulation, to an active state. Function **Change State (i)** performs this changeover for machine i. The function start the shop, program 7 needs merely invoke Change State for each machine.

Program 8 gives the code for **Change State.** If machine p is idle or in its change over states, Change State returns 0. Otherwise it returns a pointer to the job that it has been working on. Additionally Change State (p) changes the state of machine p. If machine p was previously idle or in its change over state, then it begins to process the next job on its queue. If its queue is empty, the machines new state is "idle". If machine p was previously processing a job, machine p moves into its changeover state.

If **M[p]. Active** is 0, then machine p is either in its idle or changeover state; The job pointer, **Last Job**, to return is 0. If the job queue is empty the machine moves in to its idle state and its finish time is set to **Large Time.** If its job queue is not empty, the first job is deleted and becomes machines p's active job. The time this job has spent waiting in machine p 's queue is added to the total wait time for this machine, and the number of task processes by the machine incremented by one. Next the task that this machine is going to work on is deleted from the jobs task list, and the finish of the machine is set to the time at which the new task will complete.

If **M[p] . active** is non zero, the machine has been working on a job whose task has just completed. A pointed to this job is to be returned,

so the pointer is saved in Last Job. The machine should now move in to its change over state and remain in that state for **Change Time** time units.

The Functions Simulate and MoveToNextMachine

The function **Simulate,** program 9, cycles through all shop events until the last job completes . n is the number of incomplete jobs , so the while loop of the program 9 terminates when no incomplete jobs remain. In each iteration of the while loop, the time for the next event determined and the clock time Now updated to this event time. A change job operation is performed on machine p on which the event occurred. If this machine has just finished a task of job (J is not zero), job * J moves to the machine on which its next task is to be performed. The function **MoveToNextMachine** performs this move. If there is no next task for job * J, the job has completed, functions MoveToNextMachine returns false and n is decremented by 1. Function Move To Next Machine first checks to see whether any unprocessed tasks remain for the job, *J . If not , the job has completed and its finish time and wait time are output .The function returns false to Indicate there was no next machine for this job.

When the job * J to be moved has a next task , the machine p for this task is determined and the job added to this machines queue of waiting jobs.In case machine p is idle ,**ChangeState** is invoked to change its state .

The Function Output Stats :

Since the time at which a job finishes as well as the time a job spends waiting in machines queues is output by **MoveToNextMachine, OutputStatus** needs to output only the time at which the machine shop completes all jobs (this time is also the time at which the last job completed and has been output by MoveToNextMachine) and the statistics(total wait time and number of tasks processed) for each machine .program1 gives the code.

Job Characteristics

Job#	#Tasks	Tasks
1	3	(1,2)(2,4)(1,1)
2	2	(3,4)(1,2)
3	2	(1,4)(2,4)
4	2	(3,1)(2,3)

Simulation

Time	Machine Queues			Active Jobs			Finish Times		
	M1	M2	M3	M1	M2	M3	M1	M2	M3
Init	1,3	-	2,4	I	I	I	L	L	L
0	3	-	4	I	I	2	2	L	4
2	3	-	4	C	I	2	4	6	4
4	2	-	4	3	I	C	8	6	5
5	2	-	-	3	I	4	8	6	6
6	2,1	4	-	3	C	C	8	9	7
7	2,1	4	-	3	C	I	8	9	L
8	2,1	4,3	-	C	C	I	10	9	L
9	2,1	3	-	C	4	I	10	12	L
10	1	3	-	2	4	I	12	12	L
12	1	3	-	C	C	I	14	15	L
14	-	3	-	I	C	I	15	15	L
15	-	-	-	C	3	I	17	16	L
16	-	-	-	C	C	I	19	19	L

6. Conclusion

The Project work namely " **Smart Garbage Collector** " has been successfully dealt with creation of Garbage Collector in C++ with Smart Pointers using Reference Count Technique. It has resolved the problems in explicit deallocation of memory, which has been the hindrance in the application of modern programming languages for the growing market of real time systems.

It also includes an application, namely Machine Shop Simulation , in which the Smart Garbage Collector is plugged in , to visualise the effectiveness of the project.The simulation results are found to agree with the analytical results to an appreciable degree.

The salient features of the Smart Garbage collector are :

1. Memory Management is easier and effective
2. Reduces the development time involved in explicit deallocation of memory.
3. A simple system can be easily integrated into existing software i.e is it provides pluggable and portable Technology.
4. For interactive software development environments , it allows dynamic interaction with a running program, since neither the program nor the developer need to keep track of every block of memory.

Future Enhancements

Eventhough the project has successfully completed all the claimed requirements have been met. However the possibilities for renovation are infinite and the scope for development is innumerable

The current working system is very flexible and feasible in nature, Hence the system can be enhanced by incorporating new specification or facility as per the requirements of the end user. By implementing special or more effective techniques of Garbage Collection in this system, it is possible to bring out more effective Garbage Collector applicable for complex real time systems.

Bibliography

Books :

[1.] Paul R. Wilson, "Uniprocessor Garbage Collection Techniques", 1992

[2.] David R.Musser, 'C++ Programming with STL" Addison Wesley Publication.

[3.] Ayrton Senna, "Memory Management in C++".

[4.] Micheal Spertus, "Garbage Collection in Oops", Geodesic Systems.

[5.] STL Tutorial and Reference Guide by Atul Saini.

URLS

[1.] http://www.goodosic.com/GreatCircle/index.html.

Describes the Automatic Memory Management System in Geodesic Systems

[2.] ftp://cs.colorado.edu/pub/techreports/zorn/CU-CS-665-93-ps.Z.

Describes the Memory Allocation Costs in Large C and C++ Programs by Detlefs, Dosser, and Zorn,

[3.] ftp://ftp.netcom.com/pub/hb/hbaker/NoMotionGC.html

Describes the Treadmill: Real-Time Garbage Collection without Motion Sickness by Henry Baker.

Appendix

Source Code

1.Reference Counting

Program 1

/*/// This program explains the real world problem in the conventional dynamic allocation of the memory while using pointers (i.e explicit deallocation of memory for objects while using pointer variables).

```
*/
#include <Student.h>
///
// Pointer to Student object
//
Student *s2;
void main(void)
{
//  Student s(123, "Ravi", 28);
Student *s = new Student(123, "Ravi", 28);
///
//        Make s2 to point to Student object which is pointed by s.
//
s2 = s;
///
// Assume that some where, we are deleting the student object
// using s.
//
delete s;
///
// The system will crash, when the program tries to access
// the data members of that deleted student object using
// the pointer s2.
//
cout << "Name = " << s2->getName( ) << endl;
}
/* This  program is the header file for the student problem, this file
is imported in the problem file named- problem.cpp */
#ifndef Student_h
#define Student_h
```

```
#include <string>
include <iostream>
sing namespace std;
lass Student
{
public:
///
// Constructor - intializer
//
Student(int a_nRollno, string a_Name, int a_nAge)
: m_nRollno(a_nRollno), m_Name(a_Name), m_nAge(a_nAge)
{
cout << "Constructor invoked" << endl;
}
///
// Destructor
//
~Student( )
{
cout << "Destructor invoked" << endl;
}
///
// Returns the roll number of the student
//
int getRollno( ) const
{
return m_nRollno;
}
///
// Returns the student name
//
string getName( ) const
{
return m_Name;
}
///
// Returns the age of the student
//
int getAge( ) const
{
return m_nAge;
}
private:
///
// Roll Number
//
```

```
int            m_nRollno;
///
// Student Name
//
string  m_Name;
///
// Age of the Student
//
int            m_nAge;
};
#endif Student_h
```

Program 2

FILENAME: RefCount.cpp

Provides the definition for the members of the class RefCount.

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
#include <RefCount.h>
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name   : RefCount
Description     : Constructor with initializer.
This object may be initialised with an existing reference count.
Input Arguments : int - Reference Count starting value.
Ouput Arguments : None
Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
RefCount::RefCount(int _RefCount) : m_nRefs(_RefCount)
{
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
// Function Name   : ~RefCount
// Description     : Destructor.
//                  This is virtual because DecRefCount() calls
//                  delete the operator when the reference count
//                  falls to zero.
// Input Arguments : None
// Ouput Arguments : None
// Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
RefCount::~RefCount(void)
{
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
// Function Name   : IncrRefCount
// Description     : Increment the number of references to this object
```

```
// Input Arguments : None
// Ouput Arguments : None
// Return Type     : None
c
void RefCount::IncrRefCount(void)
{
++m_nRefs;
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
// Function Name  : DecrRefCount
// Description    : Decrement the number of references to this
object.
// If count falls to zero the object is deleted.
// Input Arguments : None
// Ouput Arguments : None
// Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
void RefCount::DecrRefCount(void)
{
// The reference count is not expected to be less than zero
// However this check is provided to allow for RefCount being
// constructed with a negative reference count. Maybe a smart
// hacker might some use for this feature. All we do is safeguard
// the state of this object.
if (m_nRefs > 0)
{
        --m_nRefs;
}

if (m_nRefs == 0)
{
// There are no more references to this object. It is safe
// to delete now. The application is not expected to access
// this object after calling this method.
delete this;
}
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name  : GetRefCount
Description    : Returns reference count of this object
Input Arguments : None
Ouput Arguments : None
Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
int RefCount::GetRefCount()
{
```

```
return m_nRefs;
}
#ifndef RefCount_h
#define RefCount_h
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
WARNING
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
```

Do not access any reference counted object after decrementing the reference count. It is not guaranteed the object may still be present.

It is safe to create objects of class derived from RefCount on the stack. However do not perform any reference counting on them.
(i.e calling IncrRefCount() and DecrRefCount() methods)

RefCount should be inherited virtual. This will help to maintain just one copy of the RefCount object in case of multiple inheritance and when more than one class happens to inherit from RefCount.

DO NOT use this on pointer to arrays. Use an array class and the use a smart poninter to the array class.

There is a potential source of problem in constructors.

If 'this' pointer needs to be passed to a function called within the constructors use SmartPointer instead of regular C++ pointers.

```
Class:Class()
{
```

```
SmartPointer<Class> thisptr = this;
Init(thisptr);
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
```

CLASS:

* RefCount

DESCRIPTION:

* This class can be used to maintain number of references to a given object.

* When the reference count falls to zero the given object is deleted.

USAGE:

* Derive any class from RefCount. Increment the refernce count by calling the

* method IncrRefCount() whenever a reference is needed and call DecrRefCount()

* when the reference is no more needed.

RELATED CLASSES:

* None

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
class RefCount
{
public:
//
// Constructor.
// This object may be initialised with an
// existing reference count.
// Note: The initial reference count could be
// less than zero.
// This is allowed in case this feature is
// needed for some reason.
//
RefCount(int _RefCount = 0);
///
// Destructor.
```

```
// This is virtual because DecRefCount()
//calls delete the operator
// when the reference count falls to zero.
//
virtual ~RefCount(void);
///
// Increment the number of references to this
//   object
//
void IncrRefCount(void);
///
// Decrement the number of references to this
//object.
// If count falls to zero the object is
//deleted.
//
void DecrRefCount(void);
///
// Returns the current reference count to
//this object.
//
int GetRefCount();
protected:
///
// Number of references to the object
//
int m_nRefs;
};
#endif RefCount_h
```

Program 3

```
//Refcount.cpp
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
FILENAME: RefCount.cpp
Provides the definition for the members of the class RefCount.
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
#include <RefCount.h>
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name  : RefCount
Description    : Constructor with initializer.
This object may be initialised with an existing
reference count.
Input Arguments : int - Reference Count starting value.
Ouput Arguments : None
eturn Type     : None
*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
RefCount::RefCount(int _RefCount) : m_nRefs(_RefCount)
{
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
```

```
// Function Name  : ~RefCount
// Description    : Destructor.
// This is virtual because DecRefCount() calls
// delete the operator when the reference count
// falls to zero.
// Input Arguments : None
// Ouput Arguments : None
// Return Type    : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
RefCount::~RefCount(void)
{
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
// Function Name  : IncrRefCount
// Description    : Increment the number of references to this object
// Input Arguments : None
// Ouput Arguments : None
// Return Type    : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
void RefCount::IncrRefCount(void)
{
++m_nRefs;
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
// Function Name  : DecrRefCount
// Description    : Decrement the number of references to this
object.
// If count falls to zero the object is deleted.
// Input Arguments : None
// Ouput Arguments : None
// Return Type    : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
void RefCount::DecrRefCount(void)
{
// The reference count is not expected to be less than zero
// However this check is provided to allow for RefCount being
// constructed with a negative reference count. Maybe a smart
// hacker might some use for this feature. All we do is safeguard
// the state of this object.
if (m_nRefs > 0)
{
--m_nRefs;
}
if (m_nRefs == 0)
{
// There are no more references to this object. It is safe
```

```
// to delete now. The application is not expected to access
// this object after calling this method.
delete this;
}
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name  : GetRefCount
Description    : Returns reference count of this object
Input Arguments : None
Ouput Arguments : None
Return Type    : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
int RefCount::GetRefCount()
{
return m_nRefs;
}
```

Program 4

```
//Problem.cpp
include <Student.h>
//
// Pointer to Student object
//
Student *s2;
void main(void)
{
//  Student s(123, "Ravi", 28);
Student *s = new Student(123, "Ravi", 28);
///
// Make s2 to point to Student object which is pointed by s.
//
s2 = s;
///
// Assume that some where, we are deleting the student object
// using s.
//
delete s;
///
// The system will crash, when the program tries to access
// the data members of that deleted student object using
// the pointer s2.
//
cout << "Name = " << s2->getName() << endl;
}
```

2. Smart Pointer Class

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
FILENAME: SmartPointer.h
Provides the declaration for the members of SmartPointer class.
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
#ifndef SmartPointer_h
#define SmartPointer_h
```

DESCRIPTION:

* This class relieves the developer from the responsibility of deleting

* objects created on heap. All the developer needs to do is to pass

* around references to objects across various functions. The smart

* pointer keeps track of the number of references to the object it

* points to and garbage collects them when all references are deleted.

* This class helps in avoiding all the potential pointer problems.

USAGE:

* While creating a class extend the class from RefCount class.

Then use The SmartPointer template to keep track of the reference count to the object and the object will be deleted when the reference count equals Zero.

RELATED CLASSES:

* None

```
template<class T> class SmartPointer
{
public:
	///
	// Constructor.
	// This is default constructor pointing to NULL object.
	//
	SmartPointer() : m_Ptr(0)
	{
```

```
}
///
// Constructor.
// This points to the given object.
//
SmartPointer(T*        t) : m_Ptr(t)
{
        if (m_Ptr != (T*) 0)
        {
                m_Ptr->IncrRefCount();
        }
}
SmartPointer(const SmartPointer<T> &rhs)
{
m_Ptr = (T *)rhs;
if (m_Ptr != (T*) 0)
{
m_Ptr->IncrRefCount();
}
}
///
// Copy Constructor.
// This is declared as a template, so that SmartPointer<Derived>
// could be passed as an argument to a function that expects
// a SmartPointer<Base>
//
template<class T2>
SmartPointer(SmartPointer<T2>& rhs)
{
m_Ptr = (T2 *)rhs;
if (m_Ptr != (T2*) 0)
{
m_Ptr->IncrRefCount();
}
}
///
// Destructor
//
virtual ~SmartPointer(void)
{
if (m_Ptr != (T*) 0)
{
m_Ptr->DecrRefCount();
}
}
///
```

```
// Access to embedded object pointer
//
operator T* (void) const
{
return m_Ptr;
}
///
// Access to embedded object reference
//
T& operator* (void) const
{
return (*m_Ptr);
}
///
// Overloaded operator to make this object behave just like a reguiar
// c++ pointer
//
T* operator-> (void) const
{
return m_Ptr;
}
///
// Assignment operator . Assigning a new object on memory.
//
SmartPointer& operator = (T* _rhs)
{
if (_rhs == m_Ptr)
{
// The user is trying to assign the same object.
// Since we already have that refernce just say OK.
return *this;
}
// We do not need refernce to our object anymore.
if (m_Ptr != (T*) 0)
{
m_Ptr->DecrRefCount();
}
// This is our new object
m_Ptr = _rhs;
if (m_Ptr != (T*) 0)
{
// We are pointing to valid object in memory
m_Ptr->IncrRefCount();
}
return *this;
}
```

```
///
// Assignment operator. Assigning a new object using the reference
//
SmartPointer& operator = (const SmartPointer& rhs)
{
if (m_Ptr == (T*)rhs)
{
// The two reference are the same
return *this;
}
// We do not refernce to our object anymore.
if (m_Ptr != (T*) 0)
{
m_Ptr->DecrRefCount();
}
// This is our new object
m_Ptr = (T*)rhs;
if (m_Ptr != (T*) 0)
{
// We are pointing to valid object in memory
m_Ptr->IncrRefCount();
}
return *this;
}
protected:
///
// The pointer to the object that this object refers to on heap
//
T*      m_Ptr;
};
template <class T>
inline bool operator<(SmartPointer<T>&lhs,SmartPointer<T> &rhs)
{
return( (T*) lhs < (T*) rhs);
}
template <class T>
inline bool operator== (const SmartPointer<T>& lhs,const
SmartPointer<T>& rhs)
{
return  ( ((T*) lhs) == ((T*) rhs));
}
#endif SmartPointer_h
```

3. Standard Template library Implement with reference Count Method

Program 1

```
//human.cpp
#include <Human.h>
#include <string>
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name   : Human
Description     : Constructor with initializer.
Input Arguments : string - Gender of the Human.
Ouput Arguments : None
Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Human:Human(string a_Gender)
{
m_Gender = a_Gender;
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name   : ~Human
Description     : Destructor.
Input Arguments : None
Ouput Arguments : None
Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Human::~Human()
{
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name   : print
Description     : Outputs the member values to the Standard Output
Device.
Input Arguments : None
Ouput Arguments : None
Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
void Human::print()
{
cout << "Human: Gender = " << m_Gender << endl;
}
//Main.cpp
#include <Human.h>
#include <Student.h>
#include <list>
```

```
void main()
{
list<SPHuman> lh;
SPHuman   h = new Human("Male");
SPStudent s = new Student("123", "PhD", "Female");
lh.push_back(h);  // Base class object
lh.push_back(s);  // Derived class object
list<SPHuman>::iterator i = lh.begin();
while (i != lh.end())
{
SPHuman t_h = *i;
t_h->print();
i++;
}
}
```

Program 2

```
//Student.cpp
#include <Student.h>
#include <string>
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name   : Student
Description     : Constructor with initializer.
Input Arguments : string - Gender of the Student.
string - Course of the Student.
string - Register number of the Student.
Ouput Arguments : None
Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Student::Student(string a_Regno, string a_Course, string
a_Gender): Human(a_Gender)
{
m_Regno  = a_Regno;
m_Course = a_Course;
}
]
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name   : ~Student
Description     : Destructor.
Input Arguments : None
Ouput Arguments : None
Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Student::~Student()
{
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
Function Name   : print
Description     : Outputs the member values to the Standard Output
Device.
Input Arguments : None
Ouput Arguments : None
Return Type     : None
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
void Student::print()
{
cout << "Student: Regno = " << m_Regno
<< " Course = " << m_Course;
Human::print();
}
```

Program 3

```
//Test.cpp
#include <Student.h>
#include <conio.h>
#include <iostream>
using namespace std;
void main(void)
{
while (true)
{
cout << "\n\n\n\n1. Normal Object Creation and Destruction" <<
endl;
cout << "2. Object Creation using Pointers" << endl;
cout << "3. Object Creation using Smart Pointers" << endl;
cout << "4. Exit" << endl;
int choice = 0;
cout << "\n\nEnter you choice (1-4): ";
cin >> choice;
if (choice == 1)
{
Student s1(123, "Ravi", 28);
}
else if (choice == 2)
{
Student *s2  = new Student(123, "Ravi", 28);
}
else if (choice == 3)
{
SPStudent s3 = new Student(123, "Ravi", 28);
}
else if (choice == 4)
{
break;
}
}

}
```

4. Source Code for Application

Program 1 : Main

```
#include <Task.h>
#include <Job.h>
#include <Machine.h>
#include <EventList.h>
#include <vector>
#include <list>
#include <iostream>
using namespace std;
///
// Global Variables.
//
long Now = 0; //current time
int  m; //number of ,machines
long n; //number of jobs
long LargeTime=100000; //finish before this time
SPEventList spEventList = 0; //pointer to event list
vector<SPMachine> vecMachines; //array of machines
///
// Function: InputData
//
void InputData()
{
  ///
  // Input machine shop data.
  //
        cout<< "Enter number of machines and jobs"<< endl;
        cin >> m  >> n;
  if(m < 1 || n < 1)
  {
                throw BadInput();
  }
  ///
  // create eventlist
  //
  spEventList = new EventList(m,LargeTime);
  ///
  // Input the machine wait times.
  //
  cout<< "Enter change-over times for machines" << endl;

  for(int j = 1; j <= m;j++ )
        {
```

```
            long ct; //change-over time
            cin >> ct;
  if (ct < 0)
  {
                    throw BadInput();
  }
  SPMachine spMachine = new Machine();
  spMachine->SetChangeOverTime(ct);
  vecMachines.push_back(spMachine);
      }

///
// Input the n jobs
//
for(int i = 1; i <= n; i++)
      {
  SPJob spJob = 0;
  cout << "Enter number of tasks for jobs " << i << endl;
  int nNumberOfTasks;  //number of tasks;
  int nMachineIDForJobsFirstTask;  //machine for first task of jaob
  cin >> nNumberOfTasks;
  if (nNumberOfTasks<1)
  {
    throw BadInput();
  }
  spJob = new Job(i);

  cout << "Enter the tasks (machine,time)"
      << "in process order" <<endl;
  for(int j = 1; j <= nNumberOfTasks;j++)
            { //get tasks
    int nMachineID;
    long nTaskTime;
    cin >> nMachineID >> nTaskTime;
    if (nMachineID < 1 || nMachineID > m || nTaskTime < 1)
    {
      throw BadInput();
    }
    if (j == 1)
    {
      nMachineIDForJobsFirstTask = nMachineID;
    }
    spJob->AddTask( nMachineID, nTaskTime);
            }
  vecMachines[nMachineIDForJobsFirstTask]->AddJob(spJob);
      }
```

```
}
///
// Function: StartShop
//
void StartShop()
{
  ///
        // Load first jobs onto each machine.
  //
  for(int i = 1; i <= m; i++)
  {
    ChangeState(i);
  }
}
///
// Function: ChangeState
//
SPJob ChangeState(int a_nMachineID)
{
  SPJob spLastJob;
  if (vecMachine[a_nMachineID]->GetActiveJob() != 0)
  {    // In idle or change over state.
    spLastJob = 0;
    ///
    // Wait over, ready for new job.
    //
    if (vecMachines[a_nMachineID]->IsEmpty())
    {    // no waiting job
      spEventList->SetFinishTime( a_nMachineID, LargeTime);
    }
    else
    { // Task job off Q and work on it.
      SPMachine spMachine = vecMachines[a_nMachineID];
      spMachine->LoadNextJob();
      SPJob spActiveJob = spMachine->GetActiveJob();
      spMachine->SetTotalWait(spActiveJob->GetArriveTime());
      spMachine->IncrNumberOfProcessedTasks();
      long nTaskTime = spActiveTask->DeleteTask();
      spEventList->SetFinishTime(a_nMachineID, Now + nTaskTime);
    }
  }
  else
  {
    spLastJob = vecMachine[a_nMachineID]->GetActiveJob();
    vecMachines[a_nMachineID]->SetActiveJob(0);
    spEventList->SetFinishTime(i, Now +
```

```
    vecMachines[a_nMachineID]->GetMachineChangeTime());
   }

  return spLastJob;
}
void Simulate()
{
  int nMachineID;
  int nTaskTime;
  while (n)
  {
    spEventList->NextEvent(nMachineID, nTaskTime);
    Now = nTaskTime;
    SPJob spJob = ChangeState(nMachineID);
    if (spJob != 0 && !MoveToNextMachine(spJob))
    {
      n--;
    }
  }
}
bool MoveToNextMachine(SPJob a_spJob)
{
  if (a_spJob->IsEmpty())
  {
    cout << "Job " << a_spJob->GetJobID() << " has Completed at "
        << Now << " Total Wait was " << (Now - a_spJob->GetLength())
<< endl;
    return false;
  }
  else
  {
    SPTask spTask = a_spJob->GetNextTask();
    int nMachineID = spTask->GetMachineID();
    vecMachines[nMachineID]->AddJob(a_spJob);
    a_spJob->SetArriveTime(Now);
    if (spEventList->NextEvent(nMachine) == LargeTime)
    {
      ChangeState(nMachineID);
    }
    return true;
  }
}
void OutputStats()
{
  cout << "Finish time = " << Now << endl;
  long TotalWait, NumTask;
```

```
 for (int i = 1; i <= m; i++)
 {
   vecMachine[i]->GetMachineStatus(TotalWait,NumTask);
   cout << "Machines " << i << " completed " << NumTask << " tasks"
   << endl;
   cout << "The total wait time was " << TotalWait;
   cout << endl << endl;
 }
}
void main(void)
{
        //Machine shop simulation.
        try{
                InputData( );  //get machine and job data
                StartShop( );  //initial machine loading
                Simulate( );   //run all jobs through shop
                OutputStats( );
        }//output machine wait times
        catch(...)
        {
                cout <<"An exception has occurred" << endl;
        }
}
```

Program 2 : EventList

```
#ifndef _EventList_h
#define _EventList_h
#include <SmartPointer.h>
#include <RefCount.h>
#include <vector>
using namespace std;

class EventList  : public virtual RefCount
{
  public:
    EventList(int a_nNumberOfMachines,long a_nBigTime)
    {
      ///
      // Initialize finish times for m machines.
      //
      if(a_nNumberOfMachines < 1)
      {
        throw BadInitializers();
      }
```

```
    m_nNumberOfMachines = a_nNumberOfMachines;
    ///
    // all machines idle,initialize with
    // large finish time.
    //
    for(int i = 1; i <= m_nNumberOfMachines;i++)
    {
      m_vecFinishTime.push_back(a_nBigTime);
    }
  }
  void NextEvent(int& a_nMachineID,long& a_nTaskCompletedTime)
  {
    ///
    // Return machine and time for next event.
    // find first machine to finish,this is
    // machine with smallest finish time.
    //
    a_nMachineID = 1;
    a_nTaskCompletedTime = m_vecFinishTime[1];
    for(int i = 2; i <= m_nNumberOfMachines; i++)
    {
      if(m_vecFinishTime[i] < a_nTaskCompletedTime)
      {
        //I finishes earlier
        a_nMachineID = i;
        a_nTaskCompletedTime = m_vecFinishTime[i];
      }
    }
  }

  long NextEvent(int a_nMachineID)
  {
    return m_vecFinishTime[a_nMachineID];
  }
  void SetFinishTime(int a_nMachineID,long a_nFinishTime)
  {
    m_vecFinishTime[a_nMachineID] = a_nFinishTime;
  }

private:
  Vector<int> m_vecFinishTime;
  int m_nNumberOfMachines; //number of machines inshop
};
typedef SmartPointer<EventList> SPEventList;
#endif _EventList_h
```

Program 3 : Job

```
#ifndef _Job_h
#define _Job_h
#include <RefCount.h>
#include <SmartPointer.h>
#include <deque>
using namespace std;

class Job  : public virtual RefCount
{
public:
  Job(long a_nJobID)
{
    m_nJobID = a_nJobID
    m_nLength = m_nArriverTime = 0;
}
    void AddTask(int a_nMachineID,long a_nTaskTime)
        {
    SPTask spTask = new Task(a_nMachineID, a_nTaskTime);
    m_TaskQ.push_back(spTask);
        }
        long DeleteTask()
        {
                //delete next task
    SPTask spTask = 0;
    ///
    // Get the task from the Q, This action removes
    // the task from Q and sets the task pointer in
    // the spTask.
    //
    TaskQ::iterator tqIterator = m_TaskQ.begin();
    if (tqIterator != m_TaskQ.end())
    {
      spTask = *tqIterator;
      m_TaskQ.pop_front();
    }
   m_nLength += spTask->GetTaskTime();
    Return spTask->GetTaskTime();
        }
 void SetArriveTime(long a_nArriveTime)
 {
   m_nArriveTime = a_nArriveTime;
 }
```

```
  long GetArriveTime() const
  {
    return m_nArriveTime;
  }
  SPTask GetNextTask()
  {
    SPTask spTask = 0;
    TaskQ::iterator tqIterator = m_TaskQ.begin();
    if (tqIterator != m_TaskQ.end())
    {
      spTask = *tqIterator;
      m_TaskQ.pop_front();
    }
    return spTask;
  }
  bool IsEmpty()
  {
    bool ret = false;
    TaskQ::iterator tqIterator = m_TaskQ.begin();
    if (tqIterator != m_TaskQ.end())
    {
      ret = true;
    }
    return ret;
  }
private:
  typedef deque<SPTask> TaskQ;
  TaskQ m_TaskQ;
  long m_nLength; //sum of scheduled task times
  long m_nArriveTime; //arrival time at current queue
  long m_nJobID; //job identifier
};
typedef SmartPointer<Job> SPJob;
#endif _Job_h
```

Program 4 : Machine

```
#ifndef _Machine_h
#define _Machine_h
#include <SmartPointer.h>
#include <RefCount.h>
#include <Job.h>
#include <deque>
using namespace std;
```

```
class Machine : public virtual RefCount
{
  public:
    Machine()
    {
      m_nTotalWait = m_nNumberOfProcessedTasks = 0;
      m_spActiveJob = 0;
    }
    bool IsEmpty()
    {
      bool ret = false;
      JobQ::iterator jqIterator = m_JobQ.begin();
      if (jqIterator != m_JobQ.end())
      {
        ret = true;
      }
     return ret;
    }
   void AddJob(SPJob a_spJob)
    {
      m_JobQ.push_back(a_spJob);
    }
    void SetMachineChangeOverTime(long a_nMachineChangeOverTime)
    {
      m_nMachineChangeOverTime = a_nMachineChangeOverTime;
    }
    void GetMachineStatus(long& a_nTotalWait,long&
     a_nNumberOfProcessedTasks)
    {
      m_nTotalWait = a_nTotalWait;
      m_nNumberOfProcessedTasks = a_nNumberOfProcessedTasks;
    }
    void IncrNumberOfProcessedTasks()
    {
      m_nNumberOfProcessedTasks++;
    }
    void SetActiveJob(SPActiveJob a_spActiveJob)
    {
      m_spActiveJob = a_spActiveJob;
    }
    SPActiveJob GetActiveJob() const
    {
      return m_spActiveJob;
    }
```



```
//#include <samrtpointer.h>

class task : public virtual Refcount
```

```
    int GetTaskTime() const
    {
      return m_nTaskTime;
    }
  private:
    int m_nMachineID;
    long m_nTaskTime;
};
typedef SmartPointer<Task> SPTask;
#endif _Task_h
```