|  |
| --- |
| **What’s the difference between an inline function and a macro?** |

The major difference between inline functions and macros is the way they are handled. Inline functions are parsed by the compiler, whereas macros are expanded by the C++ preprocessor. This difference creates other differences, as best illustrated by examples.

The C++ preprocessor implements macros by using simple text replacement. Suppose we have the following macro:

|  |
| --- |
| #define SUM(a,b) (a+b) |

When the preprocessor comes across any occurrences of SUM(first, last) in the code, then that text will be replaced by (first + last). When would one want to use a macro? Usually when what you’re substituting for is very simple, and does justify the overhead of a function call. Remember that function calls do incur overhead.

Inline functions, as mentioned earlier, are parsed by the compiler directly instead of the preprocessor. Inline functions look very similar to regular functions. Here is what an inline function implementation of the SUM macro would look like:

|  |
| --- |
| // note the use of the 'inline' keyword  inline int sum(int a, int b)  {  return (a+b);  } |

The difference between an inline function and a regular function is that wherever the compiler finds a call to an inline function, it ***writes a copy of the compiled function definition***. However, with a regular function, a normal function call is generated

The reason C++ has inline functions and macros is to eliminate the overhead incurred by function calls. However, the tradeoff of this is the fact that the program size increases with both macros and inline functions. Remember that inline functions look like regular functions, but macros are implemented with text replacement.

The fact that macros use text replacement creates the potential for bugs. Suppose we have the following code:

|  |
| --- |
| #define DOUBLE(X) X\*X  int y = 3;  int j = DOUBLE(++y); |

If you’re expecting that j will be assigned a value of 4 squared (16), then you would be wrong. Because of the text replacement, what actually happens is that the DOUBLE(++y) expands to ++y \* ++y, which equals 5\*5, giving us 25. This problem would not occur if DOUBLE were implemented as an inline function. Inline functions only evaluate their arguments once, so any side effects of evaluation happens only once.

Another problem with macros occurs with binding. Suppose we have a macro with two statements, and then we try to use that macro with an if statement. If we decide not to use the curly brackets with our if statement, then we will have something that looks like this:

|  |
| --- |
| #define ADD\_TWO(x,y) x += 2; y +=2  bool flag = true;  int j = 5, k = 7;  if(flag)  ADD\_TWO(j,k); |

Then you’re probably thinking that the macro will expand to this:

|  |
| --- |
| if(flag)  {  j +=2;  k +=2;  } |

But what actually happens is that the if statement binds to the first expression in the macro. So this is what it really expands to:

|  |
| --- |
| if(flag)  {  j +=2;  }  k +=2; |

If we had used an inline function instead of a macro, the problem shown above would not have occurred. This is because an inline function is treated as a single statement, so the entire function would be bound to the if statement.

Because of all this, it’s generally considered a good idea to use inline functions over macros.

**Difference between endl and backslash n**

"\n" and "endl" both cause a new line to be printed.

The difference between two is

"endl" not only inserts a newline character, but it also flushes the output buffer,before sending the value which u actually wants to print. Hence "endl" is more reliable than "\n".

**difference between static and dynamic linking**

Static Versus Dynamic Library

A static library is linked into your executable code. It is part of the executable, thus creating a larger executable file. A larger executable file requires more RAM and disk space. On the other hand, a static library is faster than a DLL because it has already linked into the executable and loaded into memory when the executable is first run. Before it can be used, a programmer-defined DLL must be loaded into memory, if that has not already loaded.

Static library has to have a link into every executable that needs it. Consequently, if you have three different executable that need a static library, this static library must be linked into all three executable. And if all three executables run at the same time, you have three copies of the static library loaded into the memory space, which makes for inefficient use of memory space. No code sharing exists across applications, which is where dynamic link libraries come into picture. A dynamic link library is loaded only at runtime. And only one copy of the library needs to get loaded into the memory space. The library can be shared across applications if more than one application needs it.

The executable file contains references to the functions in the DLL. If you create ten different programs that use the same library, you need to create only one DLL, which saves memory and disk space. Primarily, a DLL works to reduce the load of an EXE, as well as share resources across multiple executables or instances of an executable. You can update these DLLs without re-linking because the executable contains references and not the actual code.

NOTE

Static libraries offer better performance. DLLs offer better memory management.

**Enhancements of c**

cin and cout iostream.h

>> extraction operator

<< insertion operator

variable declaration anywhere

scope resolution operator

new and delete operators

default arguments

function overloading

inline functions

reference

const keyword

**inline functions**

syntax

inline void disp()

{

//code

}

in case of normal functions, when function is called,

a) control goes to function body

b) execution of function body

c) return the control back to caller.

in case of inline functions, when function is called,

entire function body is placed at the place where it is called.

This is done at compile time. So the advantage is, execution is faster.

But again it depends upon compiler to compiler. If the code is large or contains some recursive statement/s, compiler may treat your function as normal function. Hence, keyword "inline" is just a request and not a command.

**inline vs macros**

1) inline replacement is done at compile time whereas macro replacement is at preprocessing time.

2) compiler may not treat your function as inline, if it contains large code or recursive statements. macros are always get replaced ,irrespective of their content.

**Pragma Pack info**

Packing concerns the binary layout of a structure or class. The items are always placed on a boundary of some kind.

#pragma pack instructs the compiler to pack structure members with particular alignment. Most compilers, when you declare a struct, will insert padding between members to ensure that they are aligned to appropriate addresses in memory. This is mainly for the performance benefit.

e.g

#include<iostream>

using namespace std;

struct foo

{

char a;

int d;

};

int main()

{

cout<<"welcome to c++"<<endl;

foo f;

cout<<"Size of f is"<<"\t"<<sizeof(f)<<endl;

}

/\*

How are the members aligned in memory?

According to 32-bit architecture:

< char >< pad >< pad >< pad > // 1 byte char, 3 bytes padding

< int > // 4 bytes

and thus sizeof(f) == 8

\*/

If the items follow directly onto the next byte, you're said to have byte alignment, forced by #pragma pack(1) in C/C++.

e.g

#include<iostream>

#pragma pack(1)

using namespace std;

struct foo

{

char a;

int d;

};

int main()

{

cout<<"welcome to c++"<<endl;

foo f;

cout<<"Size of f is"<<"\t"<<sizeof(f)<<endl;

}

Since there is no padding , size is 5.

**release build**

Understanding Release Builds

You commonly develop your program by using debug builds.

By the time you are ready to create a release build:

All your code should be in place,

Your code’s logic should be correct,algorithms are adequate etc.

By default, a release build uses optimizations. i.e the compiler will not produce symbolic debugging information. The absence of symbolic debugging information, along with the fact that code is not generated for TRACE and ASSERT calls, means that the size of your executable file is reduced and will therefore be faster.

Why OOPS ?

**To create low maintenance software**.

In the Procedure-Oriented approach, the problem is viewed

as a sequence of things to be done, such as reading,

calculating and printing. A number of functions are written

to accomplish this task. The primary focus is on functions.

E.g. add() ,modify() and delete() functions are called from

main() .

Procedure-oriented programming basically consists of

writing a list of instructions for the computer to follow,

and organizing these instructions into groups known as

functions. We normally use a flowchart or Algorithms to

organize these actions and represent the flow of control

from one action to another.

While we concentrate on the development of functions, very

little attention is given to the data that are being used

by various functions. What happens to the data? How are

they affected by the functions that work on them?

In a multi-function program, many important data items are

placed as global so that they may be accessed by all the

functions. Each function may have its own local data.

Global data can be modified from any function. In a large

program it is very difficult to identify what data is used

by which function. In case we need to revise an external

data structure, we should also revise all functions that

access the data.

Another serious drawback with the procedural approach is

that it does not model real world problems very well. This

is because functions are action-oriented and do not really

correspond to the elements of the problem (eg:

customer,account,transaction etc).

Some of the features are

* Emphasis is on doing things (algorithms)
* Large programs are divided into smaller programs known as functions.
* Most of the functions share global data.
* Data move openly around the system from function to function.
* Functions transforms data from one form to another. (eg: data can be changed by any function)
* Employs top-down approach in program design.

(top-down means Function has given more importance than data.)

Let’s say we need to design Banking Application :

Using Procedure Oriented Programming e.g. C prog. , we may have following algorithm :

switch(open)

{

Saving

RD

Current

FD

}

switch(calculate)

{

Saving

RD

Current

FD

}

switch(print)

{

Saving

RD

Current

FD

}

The above algorithm works fine so far.

Now what if we introduce “Flexi-Deposit” after some time ……….

What the new programmer will do ?

He will have to modify all the above code.

That will impact everything. It can be error-prone also.

It will be difficult for a new person to understand the code which is written.

POP - makes u go back to someone’s code , understand and then make changes.

Using OOPS, we may go for the following hierarchy :

Account

calcInt() abstract method

Saving Current RD FD [ child classes ]

How does it make it low maintenance ?

We can derive “Flexi-Deposit” from FD , override “calcInt()”. Thus your maintenance is low. It will not trouble other existing classes.

Maintenance :- going to existing code and review it.

Example :

Let's say that you are working for a vehicle parts manufacturer that needs to update it's online inventory system. You need to design two similar but separate forms for a website, one form that processes information about cars and one that does the same for trucks.

For cars, you need to record the following information:

* Color
* Engine Size
* Transmission Type
* Number of doors

For trucks, the information will be similar, but slightly different. You need:

* Color
* Engine Size
* Transmission Type
* Cab Size
* Towing Capacity

In procedural programming, you would write the code first to process the car form and then the code for the truck form.

With object-oriented programming, you would write a base class called vehicle that would record the common characteristics what we need from both trucks and cars. In this case, the vehicle class will record:

* Color
* Engine Size
* Transmission Type

You'll make each one of those characteristics into a separate method. The color method, for example, could take the color of the vehicle as a parameter and do something with it, like storing it in a database.

Next, you will create two more classes: truck and car, both of which will inherit all of the methods of the vehicle class and extend it with methods that are unique to them.

The car class will have a method called numberOfDoors and the truck class will have the methods cabSize and towingCapacity.

So let's assume that you have a working example for both procedural and OO programming.

Now, let's run through a few scenarios that you would come across in a normal working environment.

Scenario 1

Suppose that you suddenly need to add a bus form, that records the following information:

* Color
* Engine Size
* Transmission Type
* Number of passengers

*Procedural:* You need to recreate the entire form, repeating the code for Color, Engine Size, and Transmission Type.

*OOP:* You simply derive bus class from the vehicle class and add the method, numberOfPassengers.

Scenario 2

Instead of storing color in a database like you previously did, for some reason your client wants the color emailed to him.

*Procedural:* You change three different forms: cars, trucks, and buses to email the color to the client rather than storing it in the database.

*OOP:* You change the color method in the vehicle class. Since the car, truck, and bus are all derived from the vehicle class, they are automatically updated.

Scenario 3

You want to move from a generic car to specific makes, for example: Nissan and Mazda.

*Procedural:* You create a new form for each make, repeating all of the code for generic car information and adding the code specific to each make.

*OOP:* You derive nissan class and mazda class from the car class and add methods for each set of unique information for these cars.

Scenario 4

Suppose there is a bug in the transmission type area of your form and you need to fix it.

*Procedural:* You open and update each form.

*OOP:* You fix the transmissionType method in the vehicle class and the changes will be carried out in every class that inherits from it.

Wrapping It Up

As you can see from the above scenarios, employing an OOP style has significant advantages over procedural programming, especially as your scale increases. Consider the savings we would receive from OOP in terms of repeated code, flexibility, and maintenance if we also had to add forms for boats, motorcycles, planes etc. Objects and methods are also far easier to test than procedural programming by using unit testing to test results.

**Advantages of OOP**

**Simplicity**: software objects model real world objects, so the complexity is reduced and the program structure is very clear.

**Modularity**: each object forms a separate entity whose internal workings are decoupled from other parts of the system.

**Modifiability**: it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.

**Extensibility**: adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.

**Maintainability**: objects can be maintained separately, making locating and fixing problems easier.

**Re-usability**: objects can be reused in different programs.

**Memory leak**

When we allocate memory dynamically but somehow lose the way to reach that memory then it is called as memory leak.

For example, suppose there is a function wherein we allocate memory dynamically and store its address in a pointer that is local to that function. When the control returns from the function, the local pointer dies losing the address of the memory it was pointing to. The dynamically allocated memory would continue to remain allocated, yet not accessible. This allocated memory is leaked memory since we have lost the way to access it.

e.g.

void disp()

{

int \*ptr=new int[3];

//do some stuff

}

void main()

{

disp();

}

In the above program,when the control returns from the disp,the memory gets leaked.

**dangling pointer**

Suppose we allocate a chunk of memory and store its address in a pointer. If this chunk of memory is freed and if the pointer continues to point to that location,the pointer is said to be a dangling pointer.

e.g.

void main()

{

int \*ptr=new int[3];

// make use of ptr

delete ptr;

// still some more statements to execute

........

......

}

In the above example," delete ptr", would free the allocated memory making "ptr" a dangling pointer.

Difference between static final and enum

You can also create your own constants by marking a variable **static const.** But sometimes you'll want to create a set of constant values to represent the *only* valid values for a variable. This set of valid values is commonly referred to as an ***enumeration****.*

Let's say that you 're creating a GUI Application where, u need to create ur component and set the font style ( plain or bold or italic).

The old way to fake an "enum":

public static const int PLAIN=0;

public static const int BOLD=1;

public static const int ITALIC=2;

later in the code we can say

void setStyle(int style)

{

switch(style)

{

case 0: cout<<"plain"<<endl;

break;

case 1: cout<<"bold"<<endl;

break;

case 2: cout<<"italic"<<endl;

break;

default: cout<<"unpredictable color"<<endl;

}

}

And we will call above function by

mc.setStyle(MyFont::BOLD);

mc.setStyle(MyFont::ITALIC);

mc.setStyle(1); // saves typing

The good news about this technique is that it DOES make the code easier to read. The other good news is that you can't ever change the value of the fake enums you've created; e.g. BOLD will always be 1. The bad news is that there's no easy or good way to make sure that the value of “style” will always be 0, 1, or 2. If some hard to find piece of code sets “style” equal to 6, it's possible that your code will break.. .

**release build**

Understanding Release Builds

You commonly develop your program by using debug builds.

By the time you are ready to create a release build:

All your code should be in place,

Your code’s logic should be correct, algorithms are adequate etc.

By default, a release build uses optimizations i.e the compiler will not produce symbolic debugging information. The absence of symbolic debugging information, along with the fact that code is not generated for TRACE and ASSERT calls, means that the size of your executable file is reduced and will therefore be faster.

**File Handling**

Required to store the data permanently, so that it can be used in future.

C Programming File Handling

C being a procedure oriented programming, has got various library functions to deal with file. e.g. fopen, fclose, fgetc,fputc etc.

C++ - File Handling

C++ uses object oriented approach for File Handling. i.e. there are various classes. e.g. class to read from file, class to write to the file etc.

Stream

Stream is a communication path between source and destination.

e.g. if ur program reads from file, u need to open input stream on file.

if ur program writes to the file, u need to open output stream on file.

C++ stream hierarchy

ios

istream ostream

iostream

ifstream ofstream

fstream

cin is an object of istream , cout is an object of ostream.

Difference Text and Binary

Text mode vs Binary mode

a) How newline is treated :-

In text mode , when u write a data inside file, "newline" is converted into "linefeed (\n)" and "carriage return (\r). Similarly when u read the data from file, "linefeed (\n)" and "carriage return (\r) " are converted back to "newline".

In Binary Mode, no such conversion take place.

b) How End Of File is represented :-

In text mode "End of File" is represented by a special character, having a value 26.

In Binary mode no such character present.

c) How numeric data is stored :-

In text mode, if u store 1234.56 , it will take 7 bytes. This is because each and every digit ( including "dot" ) is treated as a single character.

In binay mode, if u store 1234.56, it will take 4 bytes only. This is because here numeric data is stored, the way it is stored in computer's memory.

JUST FOR KNOWLEDGE

1) in text mode while writing, newline is converted to '\n' and '\r' combination. While reading, '\n' and '\r' combination is converted to newline.

in binary mode, no such conversion takes place.

2) in text mode end of file is represented by "EOF" having a value 26.

in binary mode, no such indication exists.

3) in text mode if u write a value 1234.567 , it will take 8 bytes. This is because, each and every digit is treated as a character.

in binary mode above mentioned number will be stored ,the way it is stored in computer's memory. That is it will take 4 bytes (float).

4) Also since you are using a fstream(), the default open mode which is ios::in | ios::out will only open an existing file, it will not create a new file. This is also true for the ios::ate open mode. The only time a file will be created is when you use either the ios::app or the ios::trunc open modes.

fstream ff("d:\\temp\\one.txt",ios::in | ios::app);

**file opening modes**

modes can be

ios::in - for reading

ios::out - for writing

ios::app - for appending.can be used with files capable of output

ios::ate - causes seek to the end of the file,when file is opened.Still i/o operations can occur anywhere within the file

ios::ate mode allows us to add data or change data anywhere in the file But In ios::app mode allows us to add data only to the end of file.

The difference between the two is that with "ate" if you change the file position you can write to the middle or beginning of the file. With "app" even if you change the file position you will write to the end of the file. If you aren't planning on changing the file position at all, just use "app".

ios::trunc - destroy existing file

ios::binary - open in binary mode.by def.it is text mode.

Default modes

ifstream - ios::in

ofstream - ios::out | ios::trunc

fstream - ios::in | ios::out

**Flush ()**

flush()

when output is performed,data is not immediately written to the physical device linked to the stream. instead information is stored in an internal buffer untill the buffer is full.Only then are the contents of that buffer written to disk.However you can force the information to be physically written to disk before the buffer is full by calling flush().

closing a file or terminating a program also flushes all buffers.

### Get and put pointers

.

The exact behavior of the get and put pointers for a stream depends on the type of stream. For example, for fstream objects, the get and put pointers are tied together. That is, any operation that moves one always moves the other. For  strstream objects, the pointers are independent. That is, either pointer can be moved without affecting the other.

**NEW AND DELETE**

You can overload the new and delete operators from within the scope of a class declaration. This feature allows a class to have its own custom new and delete operators. You typically use this feature to gain a performance benefit.

Global heap operations often rely on operating system functions to allocate and free memory. These operations can be inefficient, particularly in a program that frequently allocates and frees many small blocks of memory in tight iterations.

Overloaded new and delete functions within a class definition always are static and have no this pointer associated with the object being created or deleted. This is because the compiler calls the new function before it calls the class's constructor function and it calls the delete function after it calls the destructor.

The new function executes before the class's constructor function. The new function cannot access any of the class's members, because no memory exists for them untill new allocates it and because the constructor function has not performed any other class-specific initializations yet. The delete operator executes after the destructor function. Consequently, the delete operator cannot have access to the class members.

**C++ delete operator**

C++The delete operator destroys the object created with new by deallocating the memory associated with the object.

The delete operator has a void return type. It has the syntax:

Read syntax diagram[Skip visual syntax diagram](http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/topic/com.ibm.vacpp7a.doc/language/ref/clrc05cplr202.htm#skipsyn-61)>>-+----+--delete--object\_pointer------------------------------><

'-::-'

The operand of delete must be a pointer returned by new, and cannot be a pointer to constant. Deleting a null pointer has no effect.

The delete[] operator frees storage allocated for array objects created with new[]. The delete operator frees storage allocated for individual objects created with new.

It has the syntax:

Read syntax diagram[Skip visual syntax diagram](http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/topic/com.ibm.vacpp7a.doc/language/ref/clrc05cplr202.htm#skipsyn-62)>>-+----+--delete--[--]--array---------------------------------><

'-::-'

The result of deleting an array object with delete is undefined, as is deleting an individual object with delete[]. The array dimensions do not need to be specified with delete[].

The result of any attempt to access a deleted object or array is undefined.

If a destructor has been defined for a class, delete invokes that destructor. Whether a destructor exists or not, delete frees the storage pointed to by calling the function operator delete() of the class if one exists.

The global ::operator delete() is used if:

* The class has no operator delete().
* The object is of a nonclass type.
* The object is deleted with the ::delete expression.

The global ::operator delete[]() is used if:

* The class has no operator delete[]()
* The object is of a nonclass type
* The object is deleted with the ::delete[] expression.

The default global operator delete() only frees storage allocated by the default global operator new(). The default global operator delete[]() only frees storage allocated for arrays by the default global operator new[]().

C++ new Operator

**C++ new Operator**

C++The new operator provides dynamic storage allocation. The syntax for an allocation expression containing the new operator is:

Read syntax diagram[Skip visual syntax diagram](http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/topic/com.ibm.vacpp7a.doc/language/ref/clrc05cplr199.htm#skipsyn-60)>>-+----+--new--+---------------------+--+-(--type--)-+--------->

'-::-' '-(--argument\_list--)-' '-new\_type---'

>--+-------------------------+---------------------------------><

'-(--+---------------+--)-'

'-initial\_value-'

If you prefix new with the scope resolution operator (::), the global operator new() is used. If you specify an *argument\_list*, the overloaded new operator that corresponds to that *argument\_list* is used. The *type* is an existing built-in or user-defined type. A *new\_type* is a type that has not already been defined and can include type specifiers and declarators.

An allocation expression containing the new operator is used to find storage in free store for the object being created. The new expression returns a pointer to the object created and can be used to initialize the object. If the object is an array, a pointer to the initial element is returned.

You can use set\_new\_handler() only to specify what new does when it fails.

You cannot use the new operator to allocate function types, void, or incomplete class types because these are not object types. However, you can allocate pointers to functions with the new operator. You cannot create a reference with the new operator.

When the object being created is an array, only the first dimension can be a general expression. All subsequent dimensions must be constant integral expressions. The first dimension can be a general expression even when an existing *type* is used. You can create an array with zero bounds with the new operator. For example:

char \* c = new char[0];

In this case, a pointer to a unique object is returned.

An object created with operator new() or operator new[]() exists until the operator delete() or operator delete[]() is called to deallocate the object's memory. A delete operator or a destructor will not be implicitly called for an object created with a new that has not been explicitly deallocated before the end of the program.

If parentheses are used within a new type, parentheses should also surround the new type to prevent syntax errors.

In the following example, storage is allocated for an array of pointers to functions:

void f();

void g();

int main(void)

{

void (\*\*p)(), (\*\*q)();

// declare p and q as pointers to pointers to void functions

p = new (void (\*[3])());

// p now points to an array of pointers to functions

q = new void(\*[3])(); // error

// error - bound as 'q = (new void) (\*[3])();'

p[0] = f; // p[0] to point to function f

q[2] = g; // q[2] to point to function g

p[0](); // call f()

q[2](); // call g()

return (0);

}

However, the second use of new causes an erroneous binding of q = (new void) (\*[3])().

The type of the object being created cannot contain class declarations, enumeration declarations, or const or volatile types. It can contain pointers to const or volatile objects.

For example, const char\* is allowed, but char\* const is not.

Placement Syntax

Arguments specifying an allocated storage location can be supplied to new by using the *argument\_list*, also called the placement syntax. If placement arguments are used, a declaration of operator new() or operator new[]() with these arguments must exist. For example:

#include <new>

using namespace std;

class X

{

public:

void\* operator new(size\_t,int, int){ /\* ... \*/ }

};

// ...

int main ()

{

X\* ptr = new(1,2) X;

}

The placement syntax is commonly used to invoke the global placement new function. The global placement new function initializes an object or objects at the location specified by the placement argument in the placement new expression. This location must address storage that has previously been allocated by some other means, because the global placement new function does not itself allocate memory. In the following example, no new memory is allocated by the calls new(whole) X(8);, new(seg2) X(9);, or new(seg3) X(10); Instead, the constructors X(8), X(9), and X(10) are called to reinitialize the memory allocated to the buffer whole.

Because placement new does not allocate memory, you should not use delete to deallocate objects created with the placement syntax. You can only delete the entire memory pool (delete whole). In the example, you can keep the memory buffer but destroy the object stored in it by explicitly calling a destructor.

#include <new>

class X

{

public:

X(int n): id(n){ }

~X(){ }

private:

int id;

// ...

};

int main()

{

char\* whole = new char[ 3 \* sizeof(X) ]; // a 3-part buffer

X \* p1 = new(whole) X(8); // fill the front

char\* seg2 = &whole[ sizeof(X) ]; // mark second segment

X \* p2 = new(seg2) X(9); // fill second segment

char\* seg3 = &whole[ 2 \* sizeof(X) ]; // mark third segment

X \* p3 = new(seg3) X(10); // fill third segment

p2->~X(); // clear only middle segment, but keep the buffer

// ...

return 0;

}

The placement new syntax can also be used for passing parameters to an allocation routine rather than to a constructor.

Related References

Imp about extra 4 bytes in case of new[] \_1

int\* num = new int[5];

the OS can allocate 4 extra bytes, store the size of the allocation in the first 4 bytes of the allocated memory and return an offset pointer (ie, it allocates memory spaces 1000 to 1024 but the pointer returned points to 1004, with locations 1000-1003 storing the size of the allocation). Then, when delete is called, it can look at 4 bytes before the pointer passed to it to find the size of the allocation.

I am sure that there are other ways of tracking the size of an allocation, but that's one option.

Imp about extra 4 bytes in case of new[]

|  |  |  |  |
| --- | --- | --- | --- |
| have such code  #include <cstdlib>  class Foo  {  int m\_nData;  public :  Foo() : m\_nData(0) { }  /\*~Foo()  {  }\*/  static void\* operator new[](const size\_t size)  {  return malloc(size);  }  static void operator delete[](void\* pvData)  {  free(pvData);  }  };  int main()  {  Foo\* pObjs = new Foo[5];  delete [] pObjs;  }  In this case I receive value of size in operator new overloading as 20 bytes as I wanted (sizeof(int) \* 5). But if I uncomment the destructor I get size as 24 bytes. Yeah, I now that these extra bytes is used to store the size of allocated memory and equals to sizeof(size\_t). I can't understand why I get them only if I implement destructor explicitly. If I don't do it, the compiler should the exact same thing or I missing something?  I've tried that on MSVS 2010 and 2012. Compiled for Win32.  [c++](http://stackoverflow.com/questions/tagged/c%2b%2b) [operator-overloading](http://stackoverflow.com/questions/tagged/operator-overloading) [new-operator](http://stackoverflow.com/questions/tagged/new-operator) [destructor](http://stackoverflow.com/questions/tagged/destructor)   |  |  |  | | --- | --- | --- | | [share](http://stackoverflow.com/q/13746517)|[improve this question](http://stackoverflow.com/posts/13746517/edit) | [edited Jan 17 at 18:52](http://stackoverflow.com/posts/13746517/revisions) | asked Dec 6 '12 at 15:08  [[https://www.gravatar.com/avatar/8a4c1a07939f3ae21da3e23868c9b25f?s=32&d=identicon&r=PG](http://stackoverflow.com/users/1102204/pavel-dubsky)](http://stackoverflow.com/users/1102204/pavel-dubsky)  [Pavel Dubsky](http://stackoverflow.com/users/1102204/pavel-dubsky) 494 | |
|  | |  |  | | --- | --- | |  | Why do you think the compiler "should do the exact same thing"? – [AndreyT](http://stackoverflow.com/users/187690/andreyt) [Dec 6 '12 at 15:29](http://stackoverflow.com/questions/13746517/operator-new-does-not-receive-extra-bytes#comment18890838_13746517) | |  | I thought that if I do not implement destructor at all the compiler will make it for me and it will look like the one I've commented in the example above - empty. – [Pavel Dubsky](http://stackoverflow.com/users/1102204/pavel-dubsky) [Dec 6 '12 at 17:33](http://stackoverflow.com/questions/13746517/operator-new-does-not-receive-extra-bytes#comment18895043_13746517) | | |  |  | | --- | --- | | 1 |  | | Conceptually, this is the case. However, when a compiler implicitly implements a destructor that does absolutely nothing, such destructor is called *trivial*. Such destructor does not really play any role and the compiler does not have to call it. Actually, even if you explicitly implement an empty destructor, the compiler doesn't have to call it, but your compiler apparently wanted to keep the call. – [AndreyT](http://stackoverflow.com/users/187690/andreyt) [Dec 6 '12 at 19:03](http://stackoverflow.com/questions/13746517/operator-new-does-not-receive-extra-bytes#comment18897478_13746517) | |  | Thanks for the answer. I really didn't know that – [Pavel Dubsky](http://stackoverflow.com/users/1102204/pavel-dubsky) [Dec 6 '12 at 19:11](http://stackoverflow.com/questions/13746517/operator-new-does-not-receive-extra-bytes#comment18897689_13746517) | |

**2 Answers**

[active](http://stackoverflow.com/questions/13746517/operator-new-does-not-receive-extra-bytes?answertab=active#tab-top) [oldest](http://stackoverflow.com/questions/13746517/operator-new-does-not-receive-extra-bytes?answertab=oldest#tab-top) [votes](http://stackoverflow.com/questions/13746517/operator-new-does-not-receive-extra-bytes?answertab=votes#tab-top)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| up vote 5 down vote accepted | "Extra bytes" requested by new[] from operator new[] are not used to "store the size of allocated memory", as you seem to believe. They are used to store the *number of elements* in the array, so that the delete[] will know how many destructors to call. In your example destructors are trivial. There's no need to call them. So, there's no need to allocate these extra bytes and store the element count.  The "size of allocated memory" (i.e. the size of the block *in bytes*) is a completely different story. It is stored and retrieved independently by a lower-level allocator - the malloc/free in your example.  In other words, in general case a memory block allocated by new[] has two sets of extra bytes in front of the actual data: the block size in bytes (introduced by malloc) and the element count (introduced by new[]). The second one is optional, as your example demonstrates. The first one is typically always present, as it is unconditionally allocated by malloc. I.e. your malloc call will physically allocate more than 20 bytes even if you request only 20. These extra bytes will be used by malloc to store the block size in bytes.  The latter happens in your example as well. You simply don't see it since it happens inside malloc.   |  |  |  | | --- | --- | --- | | [share](http://stackoverflow.com/a/13746628)|[improve this answer](http://stackoverflow.com/posts/13746628/edit) | [edited Dec 6 '12 at 16:30](http://stackoverflow.com/posts/13746628/revisions) | answered Dec 6 '12 at 15:13  [[https://www.gravatar.com/avatar/89a4da7af5bf1558b7d2704309f1f118?s=32&d=identicon&r=PG](http://stackoverflow.com/users/187690/andreyt)](http://stackoverflow.com/users/187690/andreyt)  [AndreyT](http://stackoverflow.com/users/187690/andreyt) 119k12153337 | |
|  | |  |  | | --- | --- | |  | Thank you for the clarity! – [Pavel Dubsky](http://stackoverflow.com/users/1102204/pavel-dubsky) [Dec 6 '12 at 16:12](http://stackoverflow.com/questions/13746517/operator-new-does-not-receive-extra-bytes#comment18892355_13746628) | |

[](http://engine.adzerk.net/r?e=eyJhdiI6NDE0LCJhdCI6NCwiY20iOjI5MTEsImNoIjoxMTc4LCJjciI6NTkxNywiZGkiOiJhMTFiNTc2NTBkMjQ0ZmE2YjFmYTVmNWM2MWJjZjVmNSIsImRtIjoxLCJmYyI6MTAxODIsImZsIjo3NDcyLCJrdyI6ImMrKyxvcGVyYXRvci1vdmVybG9hZGluZyxuZXctb3BlcmF0b3IsZGVzdHJ1Y3RvciIsIm53IjoyMiwicmYiOiJodHRwOi8vd3d3Lmdvb2dsZS5jby5pbi91cmw_c2E9dCZyY3Q9aiZxPSZlc3JjPXMmc291cmNlPXdlYiZjZD0yJnNxaT0yJnZlZD0wQ0RFUUZqQUImdXJsPWh0dHA6Ly9zdGFja292ZXJmbG93LmNvbS9xdWVzdGlvbnMvMTM3NDY1MTcvb3BlcmF0b3ItbmV3LWRvZXMtbm90LXJlY2VpdmUtZXh0cmEtYnl0ZXMmZWk9WTZsU1VwMmdKY0xfclFlc3g0SG9EdyZ1c2c9QUZRakNORjNOb3FsRUVKekphWkwyd1VKZnZsUjJrUERyQSZidm09YnYuNTM1MzcxMDAsZC5ibWsiLCJydiI6MCwicHIiOjE2MDQsInN0Ijo4Mjc3LCJ6biI6NDQsInVyIjoiaHR0cDovL2NhcmVlcnMuc3RhY2tvdmVyZmxvdy5jb20vIn0&s=ywDr8dUaiNTJIXXtY0KYCh8LDpU)

|  |  |
| --- | --- |
| up vote 5 down vote | If the compiler doesn't need to call the destructor, it doesn't need to remember how many destructors to call. You would not observe this behaviour if you had a data member that needed destructing, like std::string, as the compiler would always need to destruct it. |

**Very Very Imp for new[] and 4 bytes extra**

#include<iostream>

using namespace std;

class Special

{

public:

Special()

{

cout<<"no-arg constructor"<<endl;

}

void\* operator new[](size\_t k)

{

cout<<"k is\t"<<k<<endl; // 8

void \*ptr=malloc(k);

return ptr;

}

~Special()

{

cout<<"in dest"<<endl;

}

};

void main()

{

Special \*s=new Special[4];

}

So far, we’ve only dealt with the non-array versions of **new** and **delete**.

**new[] / delete[]**

Ah, this is where the fun begins! Most people don’t realize it, but in something so fundamental such as **new[]** and **delete[]**, there’s already compiler magic involved. The C++ standard just mandates *what* **new[]** and **delete[]** should do, but not *how*.

Let’s start with a simple example:

|  |  |
| --- | --- |
| 1 | int\* i = new int [3]; |

The above allocates storage for 3 ints by calling **operator new[]**, and since int is an integral type, there’s no constructors to call. Like with **operator new**, we can overload **operator new[]** and use placement-syntax as well:

|  |  |
| --- | --- |
| 1  2  3  4  5 | // our own version of operator new[]  void\* operator new[](size\_t bytes, const char\* file, int line);    // calls the above operator new[]  int\* i = new (\_\_FILE\_\_, \_\_LINE\_\_) int [3]; |

The behaviour of **delete[]** and **operator delete[]** is the same as with **delete** and **operator delete**. We can call **operator delete[]** directly, but must make sure to call the destructors manually.

But what happens with non-POD types?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | struct Test  {    Test(void)    {      // do something    }      ~Test(void)    {      // do something    }      int a;  };    Test\* i = new (\_\_FILE\_\_, \_\_LINE\_\_) Test [3]; |

Even though sizeof(Test) == 4, our version of **operator new[]** will get called with an argument of 16 bytes. Why? Think about how the array needs to be deleted:

|  |  |
| --- | --- |
| 1 | delete[] i; |

The compiler must somehow know how many instances of type Test are to be deleted – otherwise it can’t call the instances’ destructors. So what almost every compiler does upon a call to **new[]** is the following:

* For N instances of type T, request an allocation for **sizeof(T)\*N + 4** bytes from **operator new[].**
* Store N in the first 4 bytes.
* Construct N instances using **placement new**, starting at ptr + 4
* Return ptr + 4 to the user.

The last bullet point is especially important: If your overload of **operator new[]** returns the memory address 0×100, the instance **Test\* i** will point to 0×104! The memory layout of the 16 bytes would then be:

0×100: 03 00 00 00    -> number of instances stored by the compiler-generated code  
0×104: ?? ?? ?? ??    -> i[0], Test\* i  
0×108: ?? ?? ?? ??    -> i[1]  
0x10c: ?? ?? ?? ??    -> i[2]

When **delete[]** is used later on, the compiler inserts code which reads the number of instances N by going back 4 bytes from the given pointer, and calls the destructors in reverse order – if the type to be deleted is non-POD. Otherwise, there’s no 4 byte overhead added because no destructors need to be called (like in the **new int[3]** example above).

Unfortunately, this compiler-defined behaviour causes problems when using our own overloads for **operator new**, **operator new[]**, **operator delete**, and **operator delete[]**. Even though we can call **operator delete[]** directly, we somehow need to figure out how many destructors to call (if any).

Which we can’t.

The reason is that we can never be sure whether the compiler inserted some extra 4 bytes in the allocation or not. This is totally compiler-dependent. It might work, but it could also horribly break with some user-defined types. And other compilers could do it differently altogether.

This is also the reason why using **delete** on instances allocated with **new[]** will most likely crash your code, and vice versa. The compiler-generated code simply tries to access memory which doesn’t belong to him (using **delete[]** for allocations via **new**), or not all instances of an array are correctly destructed (using **delete** for allocations via **new[]**), or else.

However, with the knowledge of what happens behind the scenes with calls to **new**, **new[]**, **delete** and **delete[]**, we can build our own allocation functions which correctly handle simple and array allocations for all types, can use our custom allocators, provide additional information like file name and line number, and more. The next post in this series will show how.

In the meantime, make sure to read the following articles as well, which also explain the concept of **global operator new** and **class operator new**:

**Coupling and Cohesion**

**Coupling**

Coupling is the degree to which one class knows about another class. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be **loosely coupled**... That is a good thing. If on the other hand, class A relies on parts of class B that are not part of class B's interface, then the coupling between the classes is tighter... not a good thing. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

If class A knows non-API stuff about class B, and class B knows non-API stuff about class A... this is Really Bad. If either class is ever changed, there's a chance that the other class will break.

Following is the example of **tight coupling**, which has been enabled by *poor encapsulation*.

class DoTaxes

{

float rate;

float doColorado()

{

SalesTaxRates str=new SalesTaxRates();

rate=str.salesRate; // bad

// instead of this , there should be a method call.

}

}

class SalesTaxRates

{

public float salesRate;//should be prv

public float adjustedSalesRate;

public float getSalesRate(String region)

{

salesRate=new DoTaxes().doColorado();

return adjustedSalesRate;

}

}

**Cohesion**

while coupling has to do with how classes interact with each other, cohesion is all about how a single class is designed. The term *"cohesion" is used to indicate the degree to which a class has a single, well-focused purpose*. Cohesion is a subjective concept. **The more focused a class is, the higher its cohesiveness - a good thing.** The key benefit of high cohesion is that such classes are typically much **easier to maintain** (and less frequently changed) than classes with low cohesion. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more **reusable** than other classes.

e.g.

class BudgetReport

{

void connectToRdbms()

{

}

void generateBudgetReport()

{

}

void saveToFile()

{

}

void print()

{

}

}

Now imagine your manager comes along and says, "Do u know that accounting application we're working on? The clients just decided that they're also going to want to generate a revenue projection report, and they want to do some inventory report also. They do like our reporting features, however, so make sure that all of these reports will let them choose a database, choose a printer and save generated reports to data files.

Rather than putting all the printing code into one report class, we probably would have been better off with the following design right from the start:

class BudgetReport

{

Options getReportingOptions()

{

}

void generateBudgetReport(Options o)

{

}

}

class ConnectToRDBMS

{

DBconnection getRDBMS()

{

}

}

class PrintStuff

{

PrintOptions getPrintOptions()

{

}

}

class FileSaver

{

SaveOptions getFileSaveOptions()

{

}

}

*This design is much more* ***cohesive****. Instead of one class that does everything, we've broken the system into four main classes, each with a very specific, or cohesive, role. Because we've built these specialized, reusable classes, it'll be* much easier to write a new report, since we've already got the database connection class, the printing class, and the file saver class, and that means they can be reused by other classes that might want to print a report.

**Message Passing:**

1   . Objects can communicate with each others by passing message same as people passing message with each other.

2 . Objects can send or receive message or information.

3 .  Message passing involves name of object, name of function (message) and information to be send.

4  .    For example, student.mark(number). Here student is object, mark is message, name is information.

User sends message to MailBox.

User will say

MailBox m;

m.send(“hello how are u?”);

here

m is object

send is message

hello how are u is information

another example

account.deposit(50000);

**smart pointers**

Implementing a simple smart pointer in C++

Introduction

What are smart pointers?

a smart pointer is a pointer which is smart. What does that mean? Actually, smart pointers are objects which behave like pointers but do more than a pointer. These objects are flexible as pointers and have the advantage of being an object (like constructor and destructors called automatically). A smart pointer is designed to handle the problems caused by using normal pointers (hence called smart).

**Problems with pointers**

What are the common problems we face in C++ programs while using pointers? The answer is memory management. E.g.

char\* pName = new char[1024];

…

SetName(pName);

…

…

if(null != pName)

{

delete[] pName;

}

How many times have we found a bug which was caused because we forgot to delete pName. It would be great if someone could take care of releasing the memory when the pointer is not useful (we are not talking about the garbage collector here). What if the pointer itself takes care of that? Yes, that’s exactly what **smart pointers** are intended to do. Let us write a smart pointer and see how we can handle a pointer better.

Let’s say we have a class called Person which is defined as below.

#include<iostream>

#include<string.h>

using namespace std;

class Person

{

int age;

char\* pName;

public:

Person()

{

pName=NULL;

age=0;

}

Person(char\* pName, int age)

{

this->pName=new char[strlen(pName)+1];

strcpy(this->pName,pName);

this->age=age;

}

~Person()

{

delete pName;

}

void display()

{

cout<<pName<<"\t"<<age<<endl;

}

void shout()

{

cout<<"aaaaaaaaaaaoooooooooo"<<endl;

}

};

Now we shall write the client code to use Person.

void main()

{

Person\* pPerson = new Person("Scott", 25);

pPerson->display();

delete pPerson;

}

Now look at this code, every time we create a pointer, we need to take care of deleting it. This is exactly what we want to avoid. We need some automatic mechanism which deletes the pointer. One thing which strikes to us is a destructor. But pointers do not have destructors, so what? Our smart pointer can have one.

So we will create a class called SP which can hold a pointer to the Person class and will delete the pointer when its destructor is called. Hence our client code will change to something like this:

void main()

{

SP p(new Person("Scott", 25));

p->display();

// Dont need to delete Person pointer..

}

Note the following things:

* We have created an object of class SP which holds our Person class pointer. Since the destructor of the SP class will be called when this object goes out of scope, it will delete the Person class pointer (as its main responsibility); hence we don’t have the pain of deleting the pointer.
* One more thing of major importance is that we should be able to call the display method using the SP class object the way we used to call using the Person class pointer, i.e., the class should behave exactly like a pointer.

**Interface for a smart pointer**

Since the smart pointer should behave like a pointer, it should support the same interface as pointers do; i.e., they should support the following operations.

* Dereferencing (operator \*)
* Indirection (operator ->)

Let us see the SP class and Person class

#include<iostream>

#include<string.h>

using namespace std;

class Person

{

int age;

char\* pName;

public:

Person(): pName(0),age(0)

{

}

Person(char\* pName, int age)

{

this->pName=new char[strlen(pName)+1];

strcpy(this->pName,pName);

this->age=age;

}

~Person()

{

cout<<"inside dest"<<endl;

delete pName;

}

void display()

{

cout<<pName<<"\t"<<age<<endl;

}

void shout()

{

cout<<"aaaaaaaaaaaoooooooooo"<<endl;

}

};

class SP

{

private:

Person\* pData; // pointer to person class

public:

SP(Person\* pValue)

{

pData=new Person(\*pValue);

}

~SP()

{

// pointer no longer requried

delete pData;

}

Person& operator\* ()

{

return \*pData;

}

Person\* operator-> ()

{

return pData;

}

};

This class is our smart pointer class. The main responsibility of this class is to hold a pointer to the Person class and then delete it when its destructor is called.

**IMP TABLE**

|  |  |  |
| --- | --- | --- |
| **Inheritance Mode** | **Parent member** | **In child class** |
| Private | Private | Not Accessible |
|  | Protected | Private |
|  | Public | Private |
|  |  |  |
| Protected | Private | Not Accessible |
|  | Protected | Protected |
|  | Public | Protected |
|  |  |  |
| Public | Private | Not Accessible |
|  | Protected | Protected |
|  | Public | Public |

**JUST FOR KNOWLEDGE**

1)istream and ostream classes , their constructor is protected. It means u can not instantiate them , but u can instantiate their child classes e.g. ifstream and ofstream.

2) When two or more objects(parent1 and parent2) are derived from

a common base class (Grandparent) ,you can prevent multiple copies

of the base class(Grandparent) from being present in an object (child)

derived from those objects(parent1 and parent2) by declaring

the base class(Grandparent) as virtual,when it is inherited

3) When two or more objects(parent1 and parent2) are derived from

a common base class (Grandparent) ,you can prevent multiple copies

of the base class(Grandparent) from being present in an object (child)

derived from those objects(parent1 and parent2) by declaring

the base class(Grandparent) as virtual,when it is inherited

3) when accessed normally ,virtual functions behave just like any other type of class member function.However what makes

virtual functions important and capable of supporting runtime polymorphism is how they behave when accessed via a base class

pointer.

4) when a derived class that has inherited a virtual function is itself used as a base class for another derived class ,the

virtual function can still be overridden. i.e. no matter how many times a virtual function is inherited,it remains virtual.\*/

5) virtual functions are hierarchical. i.e. when function is declared as virtual by a base class,it may be overridden by a

derived class. However,the function does not need to be overridden. When a derived class fails to override a virtual

function,then when an object of that derived class accesses that function,the function defined by the base class is used\*/

6) when a derived class fails to override a virtual function,the

first redefinition found in reverse order of derivation is used\*/

7) when accessed normally ,virtual functions behave just like any other type of class member function.However what makes

virtual functions important and capable of supporting runtime polymorphism is how they behave when accessed via a base class

pointer. \*/

8) when a derived class that has inherited a virtual function is itself used as a base class for another derived class ,the

virtual function can still be overridden. i.e. no matter how many times a virtual function is inherited,it remains virtual.\*/

9) \*virtual functions are hierarchical. i.e. when function is declared as virtual by a base class,it may be overridden by a

derived class. However,the function does not need to be overridden. When a derived class fails to override a virtual

function,then when an object of that derived class accesses that function,the function defined by the base class is used\*/

10) /\*when a derived class fails to override a virtual function,the

first redefinition found in reverse order of derivation is used\*/

abstract class

in c++ the moment u declare at least one pure virtual function inside the class, it becomes abstract class.

abstract class can not be instantiated, but a pointer or reference of abstract class can be created ( for late binding purpose)

abstract class can contain pure virtual , virtual or non-virtual functions.

child class of abstract class has to define pure virtual function which is declared in the parent class.

Why abstract class

Abstract classes let you define some behaviors; they force your subclasses to provide others.   
  
For example, if you have an application framework, an abstract class may provide default services such as event and message handling. Those services allow your application to plug in to your application framework. However, there is some application-specific functionality that only your application can perform.   
Such functionality might include startup and shutdown tasks, which are often application-dependent. So instead of trying to define that behavior itself, the abstract base class can declare abstract shutdown and startup methods. The base class knows that it needs those methods, but it doesn't know how to perform those actions; it only knows that it must initiate the actions.

Some more examples for abstract classes :

abstract class Account

abstract double calcInt();

CurrentAccount, SavingsAccount,FD,RD all these child classes , they have to provide implementation for “calcInt()” as calculation of interest is must for every account type.

abstract class Employee  
{

// some other non-abstract methods  
     public double calculateSalary(double amt);  
}  
  
 class PermanentEmp:public Employee  
{  
  
public double calculateSalary(double amt){  
//salary calculation varies for permanent employee  
}  
  
class ContractEmp:public Employee  
{  
  
public double calculateSalary(double amt){  
//salary calculation varies for contract employee  
}

Well you can't have incomplete implementation with a simple class. In a simple class you have to provide all the concrete implementations. Where as in abstract class, you can have some concrete and some incomplete implementations.

For eg.

class base // abstract class

{

public:

virtual void disp1()=0;

virtual void disp2(){}

void disp3(){}

};

class sub:public base

{

public:

void disp1(){}

};

int main()

{

sub s;

s.disp1(); s.disp2(); s.disp3();

}

Eg2

class Shape // abstract

{

public:

virtual void draw()=0; // pure virtual function or Do-Nothing function

};

class Triangle:public Shape

{

public:

void draw()

{

triangle draw

}

};

class Rect:public Shape

{

public:

void draw()

{

Rect draw

}

};

class Polygon:public Shape

{

public:

void draw()

{

Poly draw

}

};

class Arc:public Shape

{

public:

void draw()

{

Arc draw

}

}

void perform(Shape \*s)

{

s->draw();

}

or

void perform(Shape &s)

{

s.draw();

}

int main()

{

perform(new Rect);

perform(new Triangle);

perform(new Polygon);

perform(new Arc);

or

Rect r; Triangle t; Polygon p; Arc a;

perform(r);

perform(t);

perform(p);

perform(a);

}

Early binding vs. Late binding

Early binding refers to events that occur at compile time. e.g. normal function calls,overloading of functions and operator.

The main advantage to early binding is efficiency.Because all information necessary to call a function is determined at compile time,these types of function calls are very fast.

late binding refers to function calls that are not resolved untill runtime.virtual functions are used to achieve late binding.The object and the function are not linked untill runtime. The main advantage of late binding is flexibility.

However because a function call is not resolved untill runtime,late binding can make for somewhat slower execution times.

Late binding

Connecting a function call to a function body is called binding.

When binding is performed before the program is run (by the compiler and linker), it is called by early binding.

When binding is performed during program execution (runtime) , it is called a Late Binding.

To cause a late binding to occur for a particular function, C++ requires that u use the "virtual" keyword when defining the function in the base class.

Late binding occurs only with virtual functions and only when u r using pointer to base class or reference to base class.

If a function is defined as "virtual" in the base class, it is "virtual" in all the derived class. The redefinition of a "virtual" function in a derived class is called as "overriding".

The keyword "virtual" tells the compiler it should not perform early binding. Instead, it should automatically install all the mechanisms necessary to perform late binding. This means that if u call "draw()" function for a "Triangle" instance through "pointer of Shape" or "reference of shape", u will get proper function invocation.

To accomplish this, the compiler creates a single table (called VTABLE) for each class that contains "virtual" functions. The compiler places the addresses of the virtual functions for that particular class in the VTABLE. In each class with virtual functions, it secretly places a pointer, called the vptr (virtual pointer i.e. VPTR), which points to the VTABLE for that object.

When u make a virtual function call through a base class pointer or reference ( that is when u make polymorphic call ), the compiler quietly inserts code to fetch the VPTR and look up the function address in the VTABLE, thus calling correct function and causing late binding to take place.

All of this - setting up the VTABLE for each class, initializing the VPTR, inserting the code for the virtual function call - happens automatically, so u don't have to worry about it. With virtual functions, the proper function gets called for an object, even if the compiler cannot know the specific type of the object.

compiler's instrctions:-

when compiler encounters any polymorphic call, it generates code like

a) see the content of base class pointer

b) access the object

c) get the vptr

d) call vptr+index function

( if there are 2 virtual functions defined in the order "disp1" and "disp2" respectively , then disp1 will be at index 0 and disp2 will be at index 1)

so if u call disp2, instruction (d) will be, call vptr+1 function.

Just for knowledge

A derived class destructor calls its base class destructor

after the derived class destructor does everything else.

If the base class needs no custom destruction,you still must

provide a virtual destructor (with an empty block)to permit the

proper destructor calls for dynamically allocated objects. \*/

Class Hierarchy

exception - top-level base class from which all exceptions are derived.

following are the child classes of exception

a) logic\_error

b) bad\_cast

c) bad\_typeid

d) bad\_alloc

e) runtime\_error

logic\_error is parent for the following classes

1) domain\_error

2) invalid\_argument

3) length\_error

4) out\_of\_range

runtime\_error is parent for following classes

1) range\_error

2) overflow\_error

3) underflow\_error

exception

exception is an abnormal condition which disrupts the normal programming flow.i.e when a particular exception gets raised in the program,program aborts.

In simple words exception is a runtime error.

exception handling allows you to manage runtime errors in an orderly fashion.

i.e if exception is handled properly,the program will not abort.

there are three terms used in exception

try,catch and throw

try- program statements that you want to monitor for exceptions are contained in a try block.

catch- the exception is caught using catch and processed.

catch block will execute only when the exception is raised.

throw- generates the exception specified by exception

if you throw an exception,for which there is no applicable catch statement,an abnormal program termination occur.Throwing an unhandled exception causes the standard library function terminate() to be invoked.terminate() calls abort() to stop your program.

syntax for exception handling

try

{

// possible statements

}

catch(type)

{

//statements to be processed in case of exception

}

here type can be any type i.e. built in types or user defined types.

In real world programs,most exceptions are class types rather than built in types.

The most common reason for this is that you can define a class type for an exception,is to create an object that describes the error that occured.this information can be used by the exception handler to help it process the error.

Just for know

1)using private variables isn't a 100% reliable way to enforce encapsulation.

The private data members and such access specifier only enforce the access permissions on compile time, and not on run time. In run time the memory address space is not protected by these constructs, and if you get the address location right, then you can access it directly (by the same process).

To access the private member of a class directly from outside of the class in C++, we need to know the definition of the class. The idea is to get the address of the private data member location and directly access the memory location through pointers.

2) istream and ostream classes , their constructor is protected. It means u can not instantiate them , but u can instantiate their child classes e.g. ifstream and ofstream.

\*/

Difference between typeid and dynamic\_cast

typeid does not check complete inheritance hierarchy i.e it does not use virtual table. Hence if u are using typeid , it is not compulsory to use polymorphic type.

dynamic\_cast can check complete inheritance hierarchy i.e it uses virtual table for this purpose. Hence if u are using dynamic\_cast , it you must have polymorphic type.

Imp info about typeid

class base{ void disp(){}}

class sub:public base{}

base \*ptr=new sub;

typeid(\*ptr).name() - base // type

vs

class base{ virtual void disp(){}}

class sub:public base{}

base \*ptr=new sub;

typeid(\*ptr).name() - sub // content

RTTI

Runtime type identification

reqd. in polymorphic language like C++,java, C# etc.

not reqd.in non-polymorphic language like c,because the type of each object is known at compile time.

e.g

Shape class pointer can point to any child class instance, i.e. Triangle,Rectangle,Circle or Polygon.

At a given time if u want to check the existence of a particular child in parent class pointer,rtti is used.

rtti can be achieved through typeid operator or dynamic\_cast.

To obtain an object's type use "typeid "

include header file <typeinfo>

typeid(object)

object is the object whose type you will be obtaining.It may be any type, i.e. userdefined or built in.

typeid() returns a ref.of type "type\_info" class.

type\_info Class

The type\_info class describes type information generated within the program by the compiler. Objects of this class effectively store a pointer to a name for the type.

type\_info class has following methods

bool operator==(const type\_info&)

bool operator!=(const type\_info&)

const char\* name()

dynamic\_cast

dynamic\_cast<targettype>(expr)

type and the expression being cast must evaluate to a pointer or reference.

dynamic\_cast is used to cast one type of pointer into other or one type of reference into other.It is used to perform casts on polymorphic types.

imp.note:-

Type information is generated for polymorphic classes only if the ("Enable Run-Time Type Information") compiler option is specified.

Just for know

static\_cast does not check information stored in virtual table. Henceu should not use it for downcast.In this example we have "print" function virtual.

in case of static\_cast, if b is pointing to base and if we call "print" , it gives runtime error.

if we remove virtual from "print", it won't give error.

2) static\_cast does not check information stored in virtual table. Hence

u should not use it for downcast.

3) static\_cast

performs a non-polymorphic cast.it can be used for any standard

conversion. no routine checks are performed.

static\_cast<type>(expr)

type specifies target type of the cast and expr is the expr. Being cast into a new type.

**JUST FOR KNOW DATE 25 APRIL 2014**

1)The compiler can not generate any code by just looking at function

template,because it does not know which data type ,the function is

going to work on.

In the above program when compiler encounter the function call,

i.e. add(20,40) ,it knows the type to use is int,because we pass two

integers. Now it generates a specific version on the add() for

type int,replacing every "type" with an int.This process is known

as instantiating the function template.The compiler also generates

a call to the newly instantiated function,and inserts it into the

code where add(20,40) is.

Similarly, the expression add('A','B') causes the compiler to generate a version of add() that operates on type char and a call to this function and add(34.30,45.89) call generates a function that works on type double.

More Importantly,The compiler generates only one version of add()

for each data type irrespective of the number of calls that have been

made for that type.

Do templates help us save memory ?

Not really, because even when we use templates the four functions (for int,char and double) do get generated.

The advantage is we are not required to type them out.The compiler

creates them from the generic version that we pass on to it.

This makes the program shorter and easier to understand.

Another advantage is ,if we are to modify the function we need to make the changes at only one place in the program instead of three places.

2)/\* Example of class template

Class templates are usually used for data storage(container) classes.

Stack ,Linked Lists are the examples of container classes.

i.e using one single class you can have any data type(int,char,float etc.), Stack or Linked List implementation.

\*/

**Namespaces**

Namespaces are used in the C++ programming language to create a separate region for a group of variables, functions and classes etc. Namespaces are needed because there can be many functions, variables for classes in one program and they can conflict with the existing names of variables, functions and classes. C++ uses namespace to avoid the conflicts.

The following will list the primary reasons due to which a conflict can occur:

* Between User defined variables/functions/classes and built in Library variables/functions/classes (e.g. sqrt(), abs() etc.)
* Between the separately included library and built in library
* Between the variables/functions/classes of the one separately included library and the same of the other separately included library

**Creating Namespaces**

The C++ language include the keyword “namespace” for creating namespaces.

namespace

{

members of namespace;

}

Take a look at an example:

#include<iostream>

using namespace std;

namespace myconstants {

const double pi = 3.141592;

}

namespace myshapes {

double area;

double perimeter;

void AreaOfCircle(double radius)

{

area = myconstants::pi \* radius \* radius;

}

void PerimeterOfCircle(double radius)

{

perimeter = 2 \* myconstants::pi \* radius;

}

}

int main(int argc, char \* argv[])

{

double r;

cout << endl << "Enter Radius:";

cin >> r;

myshapes::AreaOfCircle(r);

cout << endl << "Area of the Circle is :" << myshapes::area;

myshapes::PerimeterOfCircle(r);

cout << endl << "Perimeter of the Circle is :" << myshapes::perimeter;

}

## Using Namespaces

Namespaces are used with the ‘using’ keyword, which makes all the members of the namespace available in the current program and the members can be used directly, without taking reference of the namespace.

Namespaces are also usable by taking their name reference with scope resolution operator. This method allows distinguishing between the members of the same name in two different namespaces.

Take a look at an example:

#include<iostream>

using namespace std;

namespace mynamespace

{

int i,j;

}

using namespace mynamespace;

int main(int argc, char \*argv[])

{

cout << endl << i << ", " << j;

}

Or without the ‘using’ keyword:

#include<iostream>

using namespace std;

namespace mynamespace

{

int i,j;

}

int main(int argc, char \*argv[])

{

std::cout << endl << mynamespace::i << ", " << mynamespace::j;

}

## Namespace std

C++ has a default namespace named std, which contains all the default library of the C++ included using #include directive. All the files in the C++ standard library declare all of its entities within the std namespace. For that reason we include the ‘using namespace std;’ statement in most programs that is using any entity defined in iostream.

purpose is to localize the names of identifiers to avoid name

conflicts. c++ library has default namespace "std"

**JUST FOR KNOW**

we can not distribute function templates and class templates in object libraries.

Reason:-

we can compile a function template or class template into object code(.obj file). The code that contains a call to the function template or the code that creates an object from a class template can get compiled. This is because the compiler merely checks whether the call matches the declaration (in case of function template) and whether the object definition matches class declaration (in case of class template). Since the function template and the class template definitions are not found, the compiler leaves it to the linker to resolve this. However, during linking, linker doesn't find the matching definitions for the function call or a matching definition for the object creation. In short the expanded versions of templates are not found in the object library. Hence the linker reports error.

**TEMPLATE**

Template is a mechanism that make it possible to use one function or class to handle many different data types.By using templates,we can design a single class/function for each type. When used with functions they are known as function templates,whereas when used with classes they are called class templates.

Templates are used to write generic functions and generic classes function overloading -writing same operation on different types.

Drawbacks of Function Overloading

1)Rewriting the same function body over and over for different type is time consuming.

2)The program consumes more disk space

3)If we locate any error in one such function,we need to remember to correct it in each function body.

With template you can write one generic function,which can serve any data type we can also write class for generic purpose i.e. a class which can create any type of array e.g. int ,char ,double etc. syntax for generic function is

template<typename type>ret.type function(arg/s)

{

}

template<class type>ret.type function(arg/s)

{

}

when you create a template function,you are in essence ,allowing the compiler to generate as many different versions of that function as are necessary for handling the various ways that your program calls the function.

Template vs Macro

Templates are typesafe whereas macros are not

**JUST FOR KNOW**

#include<iostream.h>

template <class type>

void add(type a,type b) //Function Template

{

cout<<endl<<a+b<<endl;

}

// "type" is known as a template argument

void main()

{

add(20,40);

add('A','b');

add(34.30,45.89);

add(100,200);

}

/\* The compiler can not generate any code by just looking at function

template,because it does not know which data type ,the function is

going to work on.

In the above program when compiler encounter the function call,

i.e. add(20,40) ,it knows the type to use is int,because we pass two

integers. Now it generates a specific version on the add() for

type int,replacing every "type" with an int.This process is known

as instantiating the function template.The compiler also generates

a call to the newly instantiated function,and inserts it into the

code where add(20,40) is.

Similarly, the expression add('A','B') causes the compiler to generate a version of add() that operates on type char and a call to this function and add(34.30,45.89) call generates a function that works on type double.

More Importantly,The compiler generates only one version of add()

for each data type irrespective of the number of calls that have been

made for that type.

Do templates help us save memory ?

Not really, because even when we use templates the four functions (for int,char and double) do get generated.

The advantage is we are not required to type them out.The compiler

creates them from the generic version that we pass on to it.

This makes the program shorter and easier to understand.

Another advantage is ,if we are to modify the function we need to make the changes at only one place in the program instead of three places.

\*/

**Object Slicing**

if an object of a derived class is assigned to a base class object,the compiler accepts it,but it copies only the base portion of the object.It slices off the derived portion of the object.

Object slicing happens because, Base copy function (assignment) and constructor does not know anything about Derived. So, only Base part of Derived will be copied. To avoid slicing problem, we should have pure virtual function.

**STL** Standard Template Library

It is a library of various containers such as vector, list, stack etc.,iterators and algorithms.

Container: - it allows u to store elements in a dynamic way. i.e. u can increase or decrease the size of container during runtime.

Containers can be stored effectively inside the file system.

STL supports several container types, some of them are

a) vector - is a sequence that you can access at random.

You append entries to and remove entries from, the end of the vector without undue overhead. Insertion and deletion at the beginning or in the middle of the vector takes more time because they involve shifting the remaining entries to make room or to close the deleted object space. A vector is an array of contiguous objects with an instance counter or pointer that indicates the end of the container. Random access is a matter of using a subscript operation.

b) list -is a sequence that you access bidirectionally.

it enables you to perform insertions and deletions anywhere without undue performance penalties. Random access is simulated by forward or backward iteration to the target object. A list consists of noncontiguous objects linked with forward and backward pointers.

c) Stack :- It is a data structure that exhibits pushdown, pop-up behavior. The most recently inserted(pushed) element is the only one that can be extracted.

d) Queue :- It is a data structure wherein you insert elements at the end and extract elements from the beginning.

e) set :- container which holds unique values.

f) map :- container holds objects that are key values and associate each key object with another parameterized type object. does not permit duplicate keys.

Iterators- they provide a common method of traversing through to the containers.

algorithms-they are some of the commonly performed tasks.

e.g sort, count , max etc.