**Item 1: Distinguish between pointers and references**

A reference must *always* refer to some object

char \*pc = 0; // set pointer to null

char& rc = \*pc; // make reference refer to dereferenced null pointer

Well, this is evil, pure and simple. The results are undefined.

Because a reference must refer to an object, C++ requires that references be initialized.

string& rs; // error! References must be initialized

string s("xyzzy");

string& rs = s; // okay, rs refers to s

Pointers are subject to no such restriction:

string \*ps; // uninitialized pointer valid but risky.

Nullability check is always issue for pointer but not for reference.

void printDouble(const double& rd)

{

cout << rd; // no need to test rd; it must refer to a //double

}

Pointers, on the other hand, should generally be tested against null:

void printDouble(const double \*pd)

{

if (pd) { // check for null pointer

cout << \*pd;

}

}

Another important difference between pointers and references is that pointers may be reassigned to refer to different objects. ***A reference, however, always refers to the object with which it is initialized.***

string s1("Nancy");

string s2("Clancy");

string& rs = s1; // rs refers to s1

string \*ps = &s1; // ps points to s1

rs = s2; // rs still refers to s1, but s1's

// value is now "Clancy"

ps = &s2; // ps now points to s2; s1 is unchanged

The most common example is operator[]. This operator typically needs to return something that can be used as the target of an assignment.

Int arr[10]={1,2,3,..}

Arr[3]=5;

Operator[] always returns the reference.

References are the feature of choice when you *know; you* have something to refer and will be neverwanted to refer to anything else. In all other cases, stick with pointers.

**Item 4: Avoid gratuitous default constructors**

Default constructors initialize objects without any information from the place where the object is being created. Sometimes it makes perfect sense. For Ex-

Like numbers, may be initialized to zero or to undefined values. Objects that act like pointers may reasonably be initialized to null. Data structures like linked lists, hash tables, maps, and the like may reasonably be initialized to empty containers.

But in real world example, things are little bit different. For example- Some xyz companie, all equipment must be tagged with a corporate ID number, without ID equipment in such

companies are nonsensical.

Consider a class for company equipment in which the corporate ID number of the equipment is a mandatory constructor argument:

class EquipmentPiece {

public:

EquipmentPiece(int IDNumber);

...

};

Because EquipmentPiece lacks a default constructor, its use may be problematic in below contexts.

**In creation of array.**

EquipmentPiece bestPieces[10] // error! No way to call

// EquipmentPiece ctors

EquipmentPiece \*bestPieces = new EquipmentPiece[10]; // error! same problem

A solution for non-heap arrays is to provide the necessary arguments at the point where the array is defined:

int ID1, ID2, ID3, ..., ID10; // variables to hold equipment ID numbers

...

EquipmentPiece bestPieces[] = { // fine, ctor arguments are provided

EquipmentPiece(ID1),

EquipmentPiece(ID2),

EquipmentPiece(ID3),

...

EquipmentPiece(ID10)

};

For heap-based construction, one way is to create array of pointer instead array of object like:

EquipmentPiece \*\* bestPieces= new operator\*[10];

For(int i=0;i<10;i++)

bestPieces[i]=new EquipmentPiece(ID);

Disadvantage:

* Clean memory to avoid the memory leak.
* Consume more memory to hold the pointer variable.

Another and best way is first allocating the raw memory for the array, then use "placement new" to construct the EquipmentPiece objects in the memory.

Allocate enough raw memory for an array of 10 EquipmentPiece objects with the help of

operator new[] function.

void \*rawMemory = operator new[](10\*sizeof(EquipmentPiece));

Make bestPieces point to it so it can be treated as an EquipmentPiece array

EquipmentPiece \*bestPieces = static\_cast<EquipmentPiece\*>(rawMemory);

Construct the EquipmentPiece objects in the memory using "placement new"

for (int i = 0; i < 10; ++i)

new (&bestPieces[i]) EquipmentPiece( *ID Number* );

This technique allows us to create arrays of objects when a class lacks a default constructor and gives the guarantee that objects are initialized.

But make sure placement new required placement delete. Normal delete operator will not work here.

// Destruct the objects in bestPieces in the inverse order in which they were constructed

for (int i = 9; i >= 0; --i)

bestPieces[i].~EquipmentPiece();

operator delete[](rawMemory); // Deallocate the raw memory

While operator delete[] leads undefine behavior.

delete [] bestPieces; // undefined! bestPieces didn't come from the new operator

**Item 2: Prefer C++-style casts.**

C++ offers, cast between object that changes a pointer-to-const-object into a pointer-to-non-const-object (i.e., a cast that changes only the const-ness of an object)

Another cast that changes a pointer-to-base-class-object into a pointer-to-derived-class-object (i.e., a cast that completely changes an object's type). Traditional C-style casts unable to do the same.

A second problem with tradition casts is that they are hard to find.

(type) expression //Tradition cast

static\_cast<type>(expression) //New CPP cast.

C++ offers four type of casting:

* **static\_cast**

*static\_cast* has basically the same power and meaning as the general-purpose C-style cast. It also has the same kind of restrictions. For example, we can't cast a struct into an int or a double.

int firstNumber, secondNumber;

...

double result = ((double)firstNumber)/secondNumber;

With the new casts, you'd write it this way:

double result = static\_cast<double>(firstNumber)/secondNumber;

* **const\_cast**

The other new C++ casts are used for more restricted purposes. const\_cast is used to cast away the constness or volatileness of an expression. By using a const\_cast, we emphasize that the only thing you want to change through the cast is the constness or volatileness of something.

class Widget { ... };

class SpecialWidget: public Widget { ... };

void update(SpecialWidget \*psw);

SpecialWidget sw; // sw is a non-const object, but

const SpecialWidget& csw = sw; // csw is a reference to const object

update(&csw); // error! can't pass a const

// SpecialWidget\* to a function

// taking a SpecialWidget\*

update(const\_cast<SpecialWidget\*>(&csw));

// fine, the constness of &csw is

// explicitly cast away (and

// csw — and sw — may now be

// changed inside update)

update((SpecialWidget\*)&csw);

// same as above, but using a

// harder-to-recognize C-style cast

Widget \*pw = new SpecialWidget;

update(pw); // error! pw's type is Widget\*, but

// update takes a SpecialWidget\*

update(const\_cast<SpecialWidget\*>(pw));

// error! const\_cast can be used only

// to affect constness or volatileness,

// never to cast down the inheritance

// hierarch

* **dynamic\_cast**

It is used to perform *safe casts* down or across an inheritance hierarchy. That is, you use dynamic\_cast to cast pointers or references to base class objects into pointers or references to derived or sibling base class objects in such a way that you can determine whether the casts

succeeded. Failed casts are indicated by a null pointer (when casting pointers) or an exception (when casting references).

Widget \*pw= new SpacialWidget();

...

update(dynamic\_cast<SpecialWidget\*>(pw));

// fine, passes to update a pointer

// to the SpecialWidget pw points to

// if pw really points to one,

// otherwise passes the null pointer

void updateViaRef(SpecialWidget& rsw);

updateViaRef(dynamic\_cast<SpecialWidget&>(\*pw));

// fine, passes to updateViaRef the

// SpecialWidget pw points to if pw

// really points to one, otherwise

// throws an exception

*dynamic\_casts are restricted to helping you navigate inheritance hierarchies. They cannot be applied to types lacking virtual functions, nor can they cast away const-ness.*