Const Correctness (Continued)

20. (Continued)

ii) (Continued)

IV) However when you apply the subscript operator to a ~~const~~ 'const MyFriedList' object, the compiler will call the 'const' subscript operator.
Since that return a 'Fried const &', you can inspect the corresponding Fried object, but you cannot mutate / change it!

VI) Example:

```
void f (MyFriedList const &a) { // <- the MyFriedList is const
  // call methods that do not change the Fried at a[3]: OK
  Fried x = a[3];
  a[3].inspect();

  // If you try to change the Fried at a[3]: error
  Fried y;
  a[3] = y;
  a[3].mutate();
}
```

compiler catch both errors at compile time

21. Using a const member function to make an "invisible" change to a data member

a) Use mutable (or as last resort, use const_cast)
must instead of const_cast

b) use mutable to a member data indicate the compiler that is ok to modify during a const member function

c) If for some reason you need to cast away the constness of 'this' via the const_cast keyword, eg: (in Set::lookup() const)

```
Set * self = const_cast<Set*>(this);
```

'self' will have the same bits as 'this' ('self' = 'this'), but 'self' is a 'Set*' rather than a 'Set const*'

Therefore, you can use 'self' to modify the object pointed to by 'this'

22. Why could we get an error converting 'Foo**' to 'Foo const**'?

a) Converting Foo** to Foo const** would be invalid and dangerous

b) Even so C++ allow the (safe) conversion of 'Foo*' to 'Foo const*', the compiler would give you an error if you try to implicit convert 'Foo**' to 'Foo const**'

23. A member function can be called on a const object only if it does not change the object.

24. C++ uses a keyword to indicate that the member function does not modify the object

25. Q: what happens in this case?

```
void A::func() const {
    member = 1234;
}
```

26. Q: which of these will compile

A)
```
void A::nonconst_func(){
    A *ap = this;
}
```

B)
```
void A::const_func() const {
    A *ap = this;
}
```

C)
```
void A::const_func2() const {
    const A *ap = this;
}
```

D)
```
void some_func(A *obj); // Changes obj.
void A::func() const {
    some_func(this);
}
```

○ A
○ B
○ C
○ D

Constness Correctness (Continued)

27. Q: Is this safe?

```
void A::func() const {
    A *ptr = const_cast<A *>(this);
    ptr->member = 1234;
}

const A a_const;
```

a) `a_const.func();` // ← Is this ok? yes No

```
A a_nonconst;
const A &const_ref(a_nonconst);
```

b) `const_ref.func()` // ← Is this ok? yes No

28. Rule says that the casting away constness is okay, but if you try to cast something that was defined const, the behavior is undefined.

29. If you cast away constness of something that is not defined const, but is being accessed through a const pointer or reference, then it is okay.

30. Remember: Constness does not mean that the object will not change. It means that we promised to not change it.

31. Is this possible?

```
void func(const A &a) {
    const int i1 = a.member;
    some_other_func();
    const int i2 = a.member;
    if (i1 != i2) {
        ...
    }
}
```

Is this possible? yes No

42 32. Constness can be physical or logical

a) Physical constness: the bits don't change.

b) Logical constness: A user of public interface of the object cannot detect any changes.

Pointers:

1. A pointer is a variable that stores a memory Address.

2. Pointers holds the memory address as its value and has the ability to "point" (hence pointer) to certain value within a memory, by use of its associated memory address.

3. To Retrieve a variable's memory address, we need to use address-of operator &

```
#include <iostream>
int main (){
    using namespace std;
    unsigned short int myInt = 98; // declare an integer and initialize it
                                   // with 98 value.

98 <-- cout << myInt << endl;

0xbff26312 <-- cout << &myInt << endl; // use address-of operator & to print out
                                       // a memory address of myInt

    return 0;
}
```

4. Assigning a variable's Memory Address to a pointer
   a. we need to declare a pointer that we can assign a memory address
   b. pointer type has to match w/ variable type it will point to.
      i) The pointer void is one exception, which can handle different types of variables it will point to

   c.
```
#include <iostream>
using namespace std;
int main(){
    unsigned short int *pPointer = NULL; Declare & initialize pointer
    unsigned short int value = 225;
    pPointer = &value;

    cout << "pPointer's memory address: " << &pPointer << endl;
    cout << "Integer value memory address: " << &value << endl;
    cout << "pPointer is pointing to memory address: " << pPointer << endl;
    cout << "pPointer access to value: " << *pPointer << endl;
    return 0;
}
```

5. Pointers and Arrays in C++ Language.

a. In C++, an array pointer is a constant pointer to its first element.

b.
```
#include <iostream>
int main(){
    using namespace std;
    int Marks [10] = {1,2,3,4,5,6,7,8,9,0}; //Declare array w/ 10 elmts
0xbf8... ← cout << Marks << endl; // Print address of array name  # Print out the memory address of the first element of an array
    cout << &Marks << endl;                                      // Print
0xbf8... ← cout << &Mark[0] << endl; // Print 1st elmnt of address array

1 ← cout << *Mark << endl;  // Print 1st elmnt value by dereferencing
                            // an array name
    return 0;
}
```

c. An array name is indeed a pointer to its 1°st elmnt; therefore we can also access the array elements by a const pointer.

d. Dereferencing an array name will access a value of the first element of a given array eg:

```
#include <iostream>
int main(){
    using namespace std;
    int Marks [10] = {1,2,3,4,5,6,7,8,9,0};
← const int *pMarks = Marks; // Constant pointer to Mars array
6 ← cout << *(pMarks+5) << endl; // Accest 6th elmnt by pMark pointer
    cout << *(Marks+5) << endl; // Accets 6th elmnt by dereferencing arr name
    cout << Mark(5) << endl;
    return 0;
}
```

e. The '+' sign tell the compiler to move 5 objects that are integers. In this case integers are 4 bytes each so we access the pointer to point address 20 bytes behind the address reserved by the first array example

Pointers (continued)

5. e. (continued):

```
#include <iostream>
int main(){
  using namespace std;
  int Marks[10] = {1,2,3,4,5,6,7,8,9,0};
  const int *pMarks = Marks;
  for (int i=0, bytes = 0; i<10; ++i, bytes += 4){
    cout << "Element " << i << ":" << pMarks << " +";
    cout << bytes << "bytes=" << (pMarks + i) << endl;
  }
  return 0;
}
```

output:

Element 0 : 0xbfa5ce0c +0 bytes = 0xbfa5ce0c
1 : +4 = 10
2 : +8 = 14
3 : +12 = 18
4 : +16 = 1c
5 : +20 = 20
6 : +24 = 24
7 : +28 = 28
8 : +32 = 2c
9 : +36 = 30

Adding 20 bytes (0x16)

6. Array of pointers:

a.

```
#include <iostream>
int main(){
  using namespace std;
  const char *linuxDist[8] = {"Debian", "Ubuntu", ... };
  for(int i=0; i<8; ++i){
    cout << linuxDist[i] << endl;
  }
  return 0;
}
```

output:

Debian
Ubuntu

46 7. Pointers vs. Arrays:

a) "In C++, pointers and arrays are closely related.
The name of an array can be used as a pointer to its initial element.
Taking a pointer to the element one beyond the end of an array is
guaranteed to work.
This is important for many algorithms. However, since such a pointer does not
in fact point to an element of the array, it may not be used for reading and writing.
The result of taking the address of the element before the initial element is
undefined and should be avoided." Bjarne Stroustrup

~~8. Why do we need pointers?~~
~~Convenient indeed~~

8. Allocating a memory from the heap:

a. variables declared and used locally inside a function are destroyed,
once a return value is passed back to a calling statement

b. once solution are global variables but they lead to decreased
code readability, efficiency & bugs. They should be avoided.

c. use new, malloc, calloc:

```
#include <iostream>
int main(){
  using namespace std;
  unsigned short *pPointer
  pPointer = new unsigned short
  *pPointer = 31;
  cout << *pPointer << endl;
  delete pPointer;
  cout << *pPointer << endl;
  pPointer = NULL;
  Return 0
}
```

d. It is important to initialize pointers to prevent a stray Pointer
which can lead to unpredictable results

Pointers (Continued)

9. Data type class pointers
   a. Declare a data type class pointer is not different from declare a pointer to any other data type.

```
#include <iostream>
using namespace std;
class Heater {
  public:
    Heater (int itsTemperature);
    ~Heather();
      int getTemperature() const;
  private:
    int temperature;
}
Heater::Heather(int itsTemperature) { temperature = itsTemperature }
Heather::~Heather(){ }
INT Heather::getTemperature() const { return temperature }

int main(){
  Heater *modelXYZ = new Heater(8);
  cout << modelXYZ->getTemperature() << endl;
  delete model XYZ;
  return 0;
}
```

   b. The (.) operator is used to access class member functions cannot be used when an object is declared as a pointer. When a class object is declared as a pointer an arrow (->) operator must be used instead.

10. Passing to a function by reference using pointers.
    a. In addition to passing by value, and reference, passing arguments to a function can be done using pointers

```
#include <iostream>
void addone(int *a, int *b) { ++*a; ++*b; }
int main(){
  int a = 1, b = 4;
a1, b4 ← cout << "a:" << a << ", b:" << b << endl;
  addone(&a, &b);
a2, b5   cout << "a:" << a << ", b:" << b << endl;
  return 0;
}
```

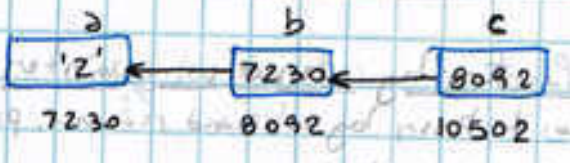
11. Memory leak caused by pointer reassignment
   a. A memory leak takes place when a memory allocated for the heap is no longer needed and was not released by delete operator

12. Memory leak caused by misuse of local variables
   a. when a function returns a value or executes its last statement, all local variables declared w/in the function definition are destroyed and no longer accessible from the stack segment
   b. Returning a pointer to any of these variables can cause memory leak
   c. using a heap inside a function and not using delete when leaving can produce memory leak

13. Pointers with increase (++) and decrease (--) operators
   a. Both the increase (++) and decrease (--) operators have greater operator precedent than the dereference operator (*); However both have a special behavior when using as suffix (the x.
      i) the expression is evaluated w/ the value it had before being incresing
   b. *p++;
      since ++ has greater precedence than *, this expression is equivalent to *(p++).
      Therefore, it increase the value of the pointer and not the value pointed at. to do so we need to change to (*p)++;
   c. *p++ = *q++; is equivalent to do:
      *p = *q;
      p++;
      q++;

14. Pointers to pointers

```
a. char a;
   char *b;
   char **c;
   a = 'z'
   b = &a;
   c = &b;
```



- c has type char** and a value of 8092
- *c has type char* and a value of 7230
- **c has type char and a value of 'z'

Pointers (continued)

15. Void pointers
   a. In C++, void represents the absence of type, therefore void pointers are pointers that point to a value that has no type and undetermine length and undetermined dereference ~~pointer~~ properties
   b. One of its uses is to pass generic parameters to a function.

```
#include <iostream>
using namespace std;
void increase (void *data, int psize){
  if (psize == sizeof(char)){
    char *pchar;
    pchar = (char*) data;
    ++ (*pchar);
  }else if (psize == sizeof(int)){
    int *pint;
    pint = (int*) data;
    ++ (*pint);
  }
}
int main (){
  char a= 'x';
  int b= 1682
  increase (&a, sizeof(a));
  increase (&b, sizeof(b));
  cout << a << ", " << b << endl; → y, 1683
  return 0;
}
```

   i) sizeof is an integrated operator in the C++ language that returns the size in bytes of its parameter
   ii) For non-dynamic data types, this value is a constant. eg sizeof(char) is 1 byte long

16. NULL pointer
   a. Do not confuse null pointers w/ void pointers
   b. A null pointer is a value that any pointer may have to represent that it is pointing to "nowhere"
   c. A void pointer is a special type of pointer that can point to somewhere w/out a specific type. ~~One refer to the value~~
   d. One refers to the value stored in the pointer itself and the other to the type data it points to.

17. Pointers to functions

a. one use is for passing a function as an argument to another function, since functions cannot be passed by dereferenced.

b. In order to declare a pointer to a function, we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses and an asterisk (*) is inserted before the name

i) `int (*minus) (int, int)`

c. ...

```
int operation (int x, int y, int (*functocall) (int, int)){
  int g = (*functocall)(x,y);
  return g;
}
...
int (*minus)(int,int) = substraction;
...
m = operation (7,5, addition);
...
```

18. Pointers to Members

a. Allows to refer to nonstatic members of class object.

b. The use of a pointer to member to point a static class member is not allowed because the address of a static member is not associated w/ any particular object.

c. To point to a static class member, you must use a normal pointer.

d. Pointers to member functions can be use in the same way as pointers to functions

e. Note: A member function does not have the same type as a non member function that has the same number and type of arguments and the same return type.

f.
```
class X {
  public:
    int a;
    void f(int b) { cout << "b: " << b << endl;}
};
int main(){
  int X::*ptiptr = &X::a; //declare pointer to data member
  void (X::*ptfptr)(int) = &X::f; //Declare a pointer to member function
  X xobject; // Create an object of class type X
  xobject.*ptiptr = 10; // initialize data member
  cout << "a: " << xobject.*ptiptr << endl;
  (xobject.*ptfptr)(20); // call member function
  return 0
}
```

output: a: 10
b: 20

Pointers (Continued)

18. (Continued)

g. To reduce syntax complexity, we can declare a typedef to be a pointer to a member.

```
typedef int X::*my_pointer_to_member;
typedef (X::*my_pointer_to_function)(int);
int main(){
    my_pointer_to_member ptiptr = &X::a;
    my_pointer_to_function ptfptr = &X::f;
    X xobject;
    xobject.*ptiptr = 10;
    cout << "a:" << xobject.*ptiptr << endl;
    (xobject.*ptfptr)(20);
}
```

h. The pointer to member operators .* and ->* are used to bind a pointer to a member of a specific class object.
Because the precedent of () (function call operator) is higher than .* and ->*, you must use parentheses to call the function pointed to by ptf.

i. Pointer-to-member conversion can occur when pointers to members are initialized, assigned, or compared.

j. Note: pointer-to-a member is not the same as a pointer to an object or a pointer to a function

19. Pointer to member operator .* and ->*.

a. The .* operator is used to dereference pointers to class members
i) The first operant must be of class type.
If the type of the first operand is class type T, or is a class that has been derived from class type T, the second operand must be a pointer to a member of a class type T.

b. The ->* operator is also used to dereference pointers to class member.
The first operant must be a pointer to a class type.
If the first type of the first operand is a pointer to class type T, or is a pointer to a class a class derived from class type T, the second operand must be a pointer to a member of class type T.

c. The .* and ->* operators bind the second operand to the first, resulting in an object or function of the type specified by the second operand.

d. If the result of .* or ->* is a function, you may use the result as the operator for the ( ) (function call) operator.
If the second operand is an lvalue, the result of .* or ->* is an lvalue

20. Is the type of "pointer-to-member function" different from "pointer-to-function"
a. yes
b. Example: let's assume we have the following function int f(char a, float b)
i) The type of this function is different depending on whether it is an ordinary function or a non-static member function of some class:
I) Its type is "int (*)(char, float)" if an ordinary function
II) Its type is "int (Fred::*)(char, float)" if a non-static member function of class Fred
ii) Note: If it's a static member function of class Fred, its type is the same as if it were an ordinary function: "int (*)(char, float)".

21. How can I avoid syntax errors when casting pointers to members?
a. By using typedef.
b. Example:

```
class Fred {
  public:
    int f(char x, float y);
    int g(char x, float y);
    int h(char x, float y);
    int i(char x, float y);
    ...
};
```

i) typedef int (Fred::*FredMemFn)(char x, float y);
I) FredMemFn is the type name, and a pointer of that type points to any member of Fred that takes (char, float), such as f, g, h, and i
ii) to declare a member-function pointer:

```
int main(){
  FredMemFn p = &Fred::f;
  ...
}
```

iii) to declare functions that receive member-function pointers:

```
void userCode(FredMemFn p){ ... }
```

Pointer (Continued)

21. (continued)

b.

iv. To declare functions that return member-function pointers:

```
Fred MemFn userCode() { ... }
```

22. How can I avoid syntax errors when calling a member function using a pointer-to-member function?

a. Using a #define macro is the easier way. They help to increase readability and writability

b.
```
#define CALL_MEMBER_FN(object, ptrToMember) ((object).*(ptrToMember))
```

c. example:

```
class Fred {
  public:
    int f(char x, float y);
    ...
};
...
typedef int (Fred::*FredMemFn)(char x, float y);
...
void userCode(Fred& fred, FredMemFn p){
  int ans = CALL_MEMBER_FN(fred, p)('x', 3.14);
```

d. Array of pointers-to-member functions:

```
FredMemFn a[] = { &Fred::f, &Fred::g, ... };
...
void userCode(Fred& fred, int memFnNum){
  CALL_MEMBER_FN(fred, a[memFnNum])('x', 3.14);
  ...
}
```

e. Declaring a pointer-to-member function that points to a const member function

```
class Fred {
  public:
    int f(int i) const;
    ...
};
typedef int (Fred::*FredMemFn)(int) const;
```

23. Example of difference between .* and ->* operator

a.
```
class Fred { ... }
typedef int (Fred::*FredMemFn)(int i, double);
...
void sample(Fred x, Fred& y, Fred *z, FredMemFn func){
    x.*func(42, 3.14);
    y.*func(42, 3.14);
    z->*func(42, 3.14);
}
```

b. Use .* when the left-hand argument is a reference to an object

c. Use ->* when the left-hand argument is a pointer to an object

d. using a macro (as previous page) it would look like follow:

```
#define CALL_MEMBER_FN (fred, ptrToMember) ((object).*(ptrToMember))
...
void sample(Fred x, Fred& y, Fred *z, FredMemFn func){
    CALL_MEMBER_FN(x, func)(42, 3.14);
    CALL_MEMBER_FN(y, func)(42, 3.14);
    CALL_MEMBER_FN(*z, func)(42, 3.14);
```

24. Can I convert a pointer-to-member-function to void?

a. No.

b. Pointers to member functions and pointers to data are not represented in the same way

A pointer to a member function might be a data structure rather than a single pointer.

If it's pointed at a virtual function, it might not actually be pointed at a statically resolvable piece of code so it might not even be a raw address - it might be a different data structure of some sort.

c. Example:

```
class Fred{
  public:
    int f(char x, float y); ...};
#define CALL_MEMBER_FN (obj, ptrToMember) ((obj).*(ptrToMember))
int callit(Fred &o, FredMemP, char x, char y){
  return CALL_MEMBER_FN(o, p)(x, y); }
int main(){
  FredMemFn p = &Fred::f;
  void *p2 = (void*) p;   // Illegal!!!
  Fred o;
  callit(o, p, 'x', 3.14f);   // OK
  callit(o, FredMemFn(42), 'x', 3.14);   // might fail
}
```

Maps

1. A map stores its keys in a way that make easier to access, if the key is given. Therefore, facilitating the search task.

2.
```
int main(){
    map<string, int> words;
    strings;
    while (cin>>s) ++words[s];

    typedef map<string, int>::const_iterator Iter;
    for (Iter p = words.begin(); p!= words.end(); ++p){
        cout << p->first << ":" << p->second << "\n";
    }
}
```

3. If we apply a code such as:
   `words["sultan"];`
   a. If "sultan" was not seen before by words, then words will insert "sultan" with the default value for an int (0).
   b. entry would be ("sultan", 0)
   c. if we apply `++words["sultan"]` then "sultan" will be associate with the int value 1 : ("sultan", 1)
   d. the steps would be:
      i. map discover that "sultan" wasn't found
      ii. insert a ("sultan", 0) pair
      iii. ++ will increment int value yielding 1.
      iv. result: ("sultan", 1).

4.
```
map<string, double> dow_price;
...
dow_price["AAA"] = 34.69;
```

5.
```
map<string, double> dow_weight;
do
dow_weight.insert(make_pair("BBB", 5.95));
```

6. `double triple_AAA_price = dow_price["AAA"];`

7.
```
if (dow_price.find("INTC") != dow_price.end()){
    cout << "Intel price is in dow_price" << endl;
}
```

8. Number: The key is called 'first' and the value is called 'second'.

9. Iterator example:

typedef map<string, double>::const_iterator Dow_iterator;

```
for( Dow_iterator p = dow_price.begin();
     p != dow_price.end();
     ++p) {
    const string & symbol = p->first;
    cout << symbol << '\t'
         << p->symbol << '\t'
         << dow_name[symbol] << '\n';
}
```

10. Example of computation using map:

```
double weighted_value( const pair<string, double> & a,
                       const pair<string, double> & b){
    return a.second * b.second;
}
```

11.
```
#include <map>
Template <class Key,
          class T,
          class compare = std::less<Key>,
          class Allocator = std::allocator<std::Pair<const Key, T>>
         > class map;
```

Map (cont. no 2)

copy map:

```
std::copy ( mapIn.begin(),
            mapIn.end(),
            std::inserter(mapOut, mapOut.end()));
```