Object Reference and Memory Allocation

Questions:
1. What is the difference between the following declarations?

```cpp
const T* p;

T* const p = new T( ..constructor args..);
```
2. Is the following C++ syntax legal?

```cpp
T* const p;
p = new T( ...constructor arguments ....);
```
3. Is the following C++ syntax legal?

```cpp
const T* p;
p = new T( ...constructor args....  );
*p = T( .... constructor args ....  );
```
4. Is the following C++ syntax legal?

```cpp
const T* p;
const T* q;
p = new T( ...constructor args.... );
q = p;
p = new T( .... constructor args .... );
```
5. Is it true or false that a Java array behaves like any other class type object when it comes to polymorphism?
6. If you absolutely must use a macro in C++, what must you do immediately afterwards?
Object Reference in C++

SPECIAL NOTE:

If we had to choose one word on which we could bestow the honor of being the source of the greatest confusion between C++ and Java, that word would be “reference”.

What it means in Java is not the same thing as what it means in C++. 
An object reference in C++ is merely an alternative name for an object or a variable. In C++, the notation

\[ T& \]

means a reference to an object of type \( T \). For example, I could say

\[
\text{int } i = 2; \\
\text{int}& \ r = i;
\]

With the second declaration, the variable \( r \) is simply another name for the variable \( i \). So if we say

\[
r = 3;
\]

we are also causing the value of \( i \) to become 3. You might think of \( r \) as serving as a sort of alias for \( i \).
If you apply the address operator \& to \texttt{r}, you will get the address of \texttt{i}:

\begin{verbatim}
int* q = &r;
\end{verbatim}

will cause \texttt{q} to point to \texttt{i}. That is, if we were to declare

\begin{verbatim}
int* p = &i;
\end{verbatim}

the pointers \texttt{p} and \texttt{q} would point to exactly the same location in the memory.
If we incremented \( r \) by

\[
 r++; 
\]

we would actually be incrementing \( i \) and achieving \( i++ \).
The following declaration is an error:

```
int& r = 100; /* WRONG */
```

That’s because the initializer for a `T&` type must be an object of type `T` whose address can be ascertained. The constant 100 has no address associated with it.

This constraint does not apply to a reference of `const T&` type. For example, it would be legal to say

```
const int& r = 100;
```
You *cannot* change the object of a reference. Here is an example:

```c
int i = 3;
int j = 100;
int& r = i;

r++; // i is now 4
r = j; // i is now 100
r++; // i is now 101
```

With these declarations, `r` remains a reference to `i` no matter what happens subsequently.
You can even have a reference to a pointer type, as the following example illustrates:

```cpp
int i = 3;

int* p = &i;

int*& q = p; // q serves as a reference to the int* variable

cout << *q; // will output 3

*q = 100; // i is now 100

cout << *p; // will output 100

cout << i; // will output 100
```
#include <iostream>
#include <string>

class User {
public:
    string name;
    int age;
    User(string nam, int a) { name = nam; age = a; }
};

int main()
{
    User u1( "Melinda", 87 );
    User* u2 = new User( "Belinda", 129 );
    User& u3 = u1;
    const User& u4 = User( "Tralinda", 187 );

    cout << "u1’s name is " << u1.name << endl;    // Melinda
    cout << "u2’s name is " << u2->name << endl;    // Belinda
    cout << "u3’s name is " << u3.name << endl;     // Melinda
    cout << "u4’s name is " << u4.name << endl;     // Tralinda

    User* p = &u1;
    User* q = &u3;
    cout << "Pointer p point to User "
         << p->name << endl;    // Melinda
    cout << "Pointer q point to User "
         << q->name << endl;    // Melinda
}
Object Reference in Java

While a class-type object in C++ can be accessed directly, or via a pointer, or via a reference, in Java there is only one mode of accessing non-primitive objects – by reference.

But the concept “reference” in Java does not have the same meaning as “reference” in C++.

While a C++ reference is simply another name for an object that was created previously, a Java reference is really more like a disguised pointer in C++.
class User {
    String name;
    int age;
    User( String nam, int a ) { name = nam; age = a; }
}

User u = new User( "Orpheus", 109 );

The C++ analogy here would be:

    User* p = new User( "Orpheus", 109 );

which would cause \texttt{p} to hold a pointer to an object of type \texttt{User}. 
Memory Allocation in C++

The operator `new` allocates memory on the heap in C++:

```cpp
string* str = new string( "hello" );

int* p = new int[500];
```
It is interesting to know that the operator “new” actually invokes the following special memory allocation functions:

```c
void* operator new( size_t size );
void* operator new[]( size_t size );
```

For the case of appropriating memory for a single object, the operator `new` does the following three things:

1. The operator `new` first figures out the size of the object for supplying it as the argument to the function `operator new()`.
2. The operator `new` then invokes the function `operator new()`.
3. And, lastly, the operator `new` casts the `void*` returned by the function `operator new()` to `string*`. 
The operator `new[]` goes through the same three steps for the case of arrays, except that now the operator must figure out the memory needed by the entire array before invoking the memory allocation function.
Memory is deallocated by using the `delete` operator for the case of a single object and the `delete[]` operator for the case of arrays, as in the following examples

```c++
string* str = new string(buffer);
delete str;

int* p = new int[1000];
delete[] p;
```

It is critical to bear in mind that after such memory deallocation, the pointer, `str` in the first case and `p` in the second case, is still pointing to the original location in memory.

A pointer left in this state if dereferenced inadvertently later in the program would lead to a program crash.
The operators `delete` and `delete[]` actually invoke the following memory deallocation functions:


```c
void operator delete( void* );
void operator delete[]( void* );
```
In the absence of any automatic garbage collection, for every call to `new` there must exist somewhere in the code a call to `delete` and for every call to `new[]` there must exist somewhere later a call to `delete[]`.

Unlike Java, C++ does not come with any guaranteed way for cleaning out unreferenced objects.
Memory Allocation in Java

All objects, including arrays, in Java are created by using the operator `new`.

This operator constructs an object of a given class and returns a reference to it.
If we first define a `User` class by

```java
class User {
    String name;
    int age;
    User(String nam, int a) { name = nam; age = a; }
}
```

we could then create an object of type `User` by

```java
User u = new User("Zygot", 38); // (A)
```
An array of, say, 100 Users would be created by

```java
User[] uAarr = new User[100];  // (B)
```

and an array of a primitive type, say ints, would be created by

```java
int[] arr = new int[100];  // (C)
```
As was the case with C++, memory allocated for new objects created in this manner comes from a part of the system memory known as the heap.
Because of automatic garbage collection, Java does not need a `delete` like operator to free up the memory that belongs to objects that are no longer referenced.
Structures in C++

Simple structures in C++ look very much like structures in C.

Example:

```cpp
struct User {
    string name;
    int age;
};
```

In C, the identifier `User` is called a structure tag that can be used to declare a variable, say, `u`, to be of type `User` by

```cpp
struct User u;
```

However, in C++, `User` is a new type directly. And we can say

```cpp
User u;
```
A structure in C++ can be thought of as a light-weight C++ class. We can use the dot or the \(- >\) notation to access the members of a structure, as in

```cpp
void f() {
    User u;
    u.name = "Orpheus";
    u.age = 89;
}
```

This function creates a new user, \(u\), whose \texttt{name} is \texttt{Orpheus} and whose \texttt{age} is 89, the rest of the members remaining uninitialized.
One can even embed functions inside the definition of a structure.

```cpp
struct User {
    string name;
    int age;
    void print() { cout << "User \\
        << name << " is of age \\
        << age << endl;
    }
};

u.print();
```
We can even engage in data hiding by declaring private some of the members of a structure.

The default access privilege of a structure member is public, which is opposite of what it is for a class.
struct User {
private:
    string name;
    int age;
public:
    User( string s1, int yy ) {
        name = s1;
        age = yy;
    }

    void print() {
        cout << "User " << name << " is of age "
             << age << endl;
    }
};

User u( "Zaphod", 27 );

User* p = new User( "Zaphod", 27 );
Quick initialization of an array of structures
```cpp
#include <iostream>
using namespace std;

struct User {
    char* name;
    int age;
    short rank;
};

void print( User*, int );

int main()
{
    User usr_list[] = {
        "Bigshot, I. R. ", 39, 1,
        "Allears, U. B. ", 29, 100,
        "Moonstruck, H. I.", 58, 45
    };

    int size = sizeof( usr_list ) / sizeof( usr_list[0] );
    print( usr_list, size );

    return 0;
}

void print( User* up, int n ) {
    for ( int i=0; i < n; i++ )
        cout << up[i].name << '\t'
             << up[i].age << '\t' << up[i].rank << '\n';
}