CS 7B - Spring 2018 - Quiz2 Solutions: Pointers

1. Consider the following complete program:

```
int main() {
    void* ptr = NULL;
}
```

   (a) Is this syntactically correct? Yes/No ANS: yes.

   (b) Is it good for anything? Yes/No ANS: Not really...it creates a null pointer to void.

   (c) Write two other equivalent expressions that could be used instead of "NULL".
   ANS: Alternatively, `nullptr` or `0`.

2. Consider this modification of the previous program:

```
#include <iostream>
int main() {
    int var = 8;
    void* ptr = &var;
    std::cout << "\nptr = " << ptr;
    std::cout << "\n&ptr = " << &ptr;
    std::cout << "\n*ptr = " << *reinterpret_cast<int*>(ptr);
}
```

   (a) This is syntactically correct. Describe what is stored in `ptr`.
   ANS: The variable `ptr` contains the address in memory of the first of the four bytes of the integer, `var`. In one instance (as shown below) this was `0x28ff0c`.

   (b) Describe what `*reinterpret_cast<int*>(ptr)` represents.
   ANS: This reinterprets (casts) `ptr` from a pointer to `void` to a pointer to `int` and then dereferences is to retrieve the integer value at that address. But the memory at that location was declared to be of type `int` in the first place, so the compiler correctly interprets this as 100000 when it's derefenced (as shown below.)

   (c) Describe what is printed to the console.
   ANS: As shown below, the first line gives the address of the the first byte of the `int` var.
   The second line prints the address of the `ptr` itself. Hey–everything has to be somewhere in memory, otherwise it ceases to be a thing.

```
ptr = 0x28ff0c
&ptr = 0x28ff08
*ptr = 8
```

   (d) Assume that memory is stored with the least significant byte first. What would the line
   `std::cout << "*ptr = " << *reinterpret_cast<short*>(ptr);`
   print to the screen, if it replaced the last line? Explain.
   ANS: We get, `*ptr = 8`: the same thing.

   (e) Would the answer to part (c) above be different if `var = 100000`?
   ANS: No, we get `*ptr = -31072`. Negative, since the 2's complement bit is set. The hexadecimal form of 100000 is 0x186A0 $= 16^4 + 8 \cdot 16^3 + 6 \cdot 16^2 + 10 \cdot 16 = 65536 + 8 \cdot 4096 + 6 \cdot 256 + 160$ (to be sure). But that requires a third byte, so when we `reinterpret_cast` that to a float we only get `86A0` Now 31072 is 7960 in hexadecimal, or `111 1001 0110 0000` in binary. Whereas the hex `86A0` is $100000 - 16^4$. What we've got then is $100000 - 16^4 - 2^15$? Hmmm...that's not it.
   Let's look at some more code.

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <stdio.h>
int main() {
```

```
int var = 100000;
    void* ptr = &var;
    std::cout << "\nptr = " << ptr;
    std::cout << "\n&ptr = " << &ptr;
    std::cout << "\n*ptr = " << *reinterpret_cast<int*>(ptr);
    std::cout << "\n*ptr = " << *reinterpret_cast<short*>(ptr);
    printf("\n%08X\n", *reinterpret_cast<int*>(ptr));
    printf("%08X\n", *reinterpret_cast<short*>(ptr));
}
```

produces this output:

```
ptr = 0x28ff0c
&ptr = 0x28ff08
*ptr = 100000
*ptr = -31072
000186A0
FFFF86A0
```

That's showing use a little more of what's going on...how about this:

```
int main() {
int var = 0b1111111111111111;
    void* ptr = &var;
    std::cout << "\nptr = " << ptr;
    std::cout << "\n&ptr = " << &ptr;
    std::cout << "\n*ptr = " << *reinterpret_cast<int*>(ptr);
    std::cout << "\n*ptr = " << *reinterpret_cast<short*>(ptr);
    printf("\n%08X\n", *reinterpret_cast<int*>(ptr));
    printf("%08X\n", *reinterpret_cast<short*>(ptr));
}
//output:
ptr = 0x28ff0c
&ptr = 0x28ff08
*ptr = 65535
*ptr = -1
0000FFFF
FFFFFFFF
```

3. Consider the following complete program:

```
#include <iostream>
#include <string>
void print(const int& value) {
  std::cout << value << std::endl;
}
void print(const float& value) {
  std::cout << value << std::endl;
}
void print(const string& value) {
  std::cout << value << std::endl;
}
int main() {
  print(42);
  print("forty three");
  print(44.0f);
}
```

(a) Write a template function that will replace all these `print()` functions with one.
ANS:

```
template<typename T>
void print(const T& t) {
    std::cout << t << std::endl;
}
```

(b) When do the function templates get created?
ANS: A function template is created at runtime, and then only if there is a call to the function in code.

4. Consider the following program:

```
#include <iostream>
#include <string>

template<int N>
class Array {
  int m_Array[N];
public:
  int getSize() const { return N; }
};
int main() { }
```

(a) What line of code would you write in `main()` that will create an `Array` of 7 `int`s?
ANS:

```
    Array<7> a7;
```

(b) How would you modify the class template so you could create an `Array` on `N` objects of any type, `T`?
ANS:

```
template<int N, typename T>
class Array {
    T m_Array[N];
public:
    int getSize() const { return N; }
    T operator[](int i) {
        return m_Array[i];
    }
    void setValue(int i, T x) {
        m_Array[i] = x;
    }
};
```

(c) How would you implement your modified class definition to create an `Array` of 12 `string`s?
ANS:

```
int main() {
    Array<12, std::string> aStr7;
    std::vector<std::string> days{"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
                                  "Friday", "Saturday", "Meday", "Youday", "Usday",
                                  "Theyday", "Weday"};
    for(int i = 0; i < 12; ++i)
        aStr7.setValue(i,days[i]);
    for(int i = 0; i < 12; ++i)
        std::cout << aStr7[i] << std::endl;

}
```