問1　セキュアプログラミングに関する次の記述を読んで，設問1～4に答えよ。

A社は，従業員数100名のソフトウェア開発会社である。A社では，PC向けソフトウェア製品として，“青少年が安全に安心してインターネットを利用できる環境の整備等に関する法律”に基づいた青少年有害情報フィルタリングソフトウェア（以下，Bフィルタという）を開発することとなった。

A社では，Bフィルタの開発に当たって，F主任をリーダとした開発チームを編成した。この開発チーム内の体制は，設計を行うDグループ，作成を行うPグループ及び試験を担当するQグループの三つのグループとした。また，プログラム開発に用いる言語は，JIS X 3014“プログラム言語C++”（以下，C++という）とした。

次は，この開発におけるセキュリティの脆弱性の発見から修正及び再発防止に至る経緯である。

〔URLマッチングの方式設計〕

Dグループでは，Bフィルタの概略機能を図1のようにまとめた。その後，要件定義を経て，URLマッチングによる有害の度合いを決定する方式設計を行い，図2の内容のとおり決定した。

(1) Bフィルタの提供形態は，特定のブラウザ（以下，Xブラウザという）へのプラグインとする。
(2) XブラウザからWebサイトへのアクセスを監視し，有害情報と判定されるアクセスをブロックする。
(3) 次の要素を総合して，有害情報であるか否かを判定する。
　(a) URLマッチング
　　アクセスしようとしたURLと，有害情報のURLリストとのマッチングによって有害の度合いを決定する。有害情報のURLリストはA社が逐次更新し，自動ダウンロード機能によってBフィルタに取り込まれる。
　(b) キーワードマッチング
　　表示するコンテンツの内容と，有害情報であるか否かを判定するためのキーワードリストとのマッチングによって有害の度合いを決定する。キーワードリストは，青少年の保護者が設定する。
(4) 保護者による設定で，動画へのアクセスをすべてブロックすることが可能である。
（以下，省略）

図1　Bフィルタの概略機能

(1) Xブラウザのプラグイン用インタフェースを利用し，Xブラウザから Web サイトへのアクセスをトリガにして URL マッチング処理を開始する。
　なお，Xブラウザからプラグインには，利用者がアクセスしようとした URL がそのまま渡される。
(2) マッチングに先行して，Xブラウザから渡される URL を次のとおり正規化する。
・文字コード：UTF-8 とする。
・URL エンコード：RFC 3986 に準拠する（パーセントエンコード）。
・正規化後の文字列長：128 バイト以下とする。128 バイトを超える部分は切り捨てる。
（以下，省略）

図2　URL マッチングの方式設計結果

〔詳細設計と作成〕

D グループでは，図 2 の方式設計結果に基づいて設計作業を進め，URL をパーセントエンコードする機能を一つの関数として作成することを決定し，その関数の仕様を図3のとおりとした。

```
void urlPercentEncode(char *dst, char *src, int n)
返却値：なし。
  効果：src が指す文字列（終端ナル文字を含む）から dst が指す文字列に文字をすべて転記する。その
        際，エンコードすべき文字を検出した場合は，当該文字をパーセントエンコードした上で dst が指
        す文字列に転記する。
        なお，n の値は dst が指す文字列に書き込める最大バイト数とする。
        dst が指す文字列への転記に当たって，バイト数が n を超える場合は，n バイトを書き込んだ時点
        で処理を打ち切って終了する。この場合，dst が指す文字列がナル文字で終了することは保証され
        ない。
```

図3　作成するパーセントエンコード関数の仕様

その後，P グループでは，パーセントエンコード関数の作成に取り掛かり，図 4 に示すソースコードを作成した。

```
1  void urlPercentEncode(char *dst, char *src, int n){
2     int srcPos = 0;
3     int dstPos = 0;
4     char x = 0;
5
6     while(src[srcPos] != '\0' && dstPos < n){
7        /* エンコードすべきか否かを判定 */
8        if(src[srcPos] == '-' || src[srcPos] == '.' ||
9                 src[srcPos] == '_' || src[srcPos] == '~' ||
10                ((src[srcPos] >= '0') && (src[srcPos] <= '9')) ||
11                ((src[srcPos] >= 'A') && (src[srcPos] <= 'Z')) ||
12                ((src[srcPos] >= 'a') && (src[srcPos] <= 'z'))){
13       /* エンコードしない場合 */
14          dst[dstPos++] = src[srcPos++];
15       }
16       else{
17       /* エンコードする場合 */
18          dst[dstPos++] = '%';
19          x = (src[srcPos] & 0x00f0) >> 4;
20          dst[dstPos++] = x <= 9 ? x + '0' : x - 10 + 'A';
21          x = src[srcPos++] & 0x000f;
22          dst[dstPos++] = x <= 9 ? x + '0' : x - 10 + 'A';
23       }
24    }
25    if(dstPos < n){
26       dst[dstPos] = '\0';
27    }
28    return;
29 }
```

図 4　最初に作成したソースコード

〔脆弱性の発見〕

開発計画に基づき，URL マッチング機能だけがおおよそ完成した時点のものをプロトタイプ 1 として，Q グループによる試験を実施した。その試験の結果，ある特定の条件下で URL マッチング機能が正常に動作しない不具合が発見された。その結果を受けてプロトタイプ 1 の動作について詳細に確認したところ，パーセントエンコード関数が実行された直後に，ほかの関数でフラグとして使用している変数（以下，フラグ変数という）が想定外の値に書き換わり，URL マッチング機能が正常に動作しないことが判明した。

ソースコードの調査を行ったところ，パーセントエンコード関数にバッファオーバフロー脆弱性があり，隣接したメモリ領域に置かれたフラグ変数が想定外の値に書き換えられることが分かった。

〔関数の仕様変更〕

B フィルタの開発チームは，B フィルタで同様の脆弱性を作り込まないようにするために，標準 C++ライブラリの文字列クラスを使用してパーセントエンコード関数を作り直すとともに，ほかの関数においても作り直しを行った。パーセントエンコード関数の作り直しに当たって，引数及び返却値を文字列型に変更するのに伴い，D グループでは関数の仕様を図 5 に示すように変更した。その後，P グループは図 5 の仕様に基づいた関数の作成に取り掛かり，図 6 のソースコードを作成した。

なお，図 3 の仕様に基づく関数 urlPercentEncode()を呼び出していた部分は，すべて図 5 の仕様に基づく関数 urlPercentEncodeString()を呼び出すように修正した。

```
string urlPercentEncodeString(string src)
返却値：処理済の文字列。
  効果：src 文字列から，返却値に文字をすべて転記する。その際，エンコードすべき文字を検出した場合
        は，当該文字をパーセントエンコードした上で返却値に転記する。
```

図 5　変更したパーセントエンコード関数の仕様

```
#include <string>
#include <iterator>

using namespace std;

string urlPercentEncodeString(string src){
   string dst("");
   string::iterator readPoint = src.[ a ]();
   string::iterator atEOL = src.[ b ]();
   char x = 0;

   while(readPoint != atEOL){
      /* エンコードすべきか否かを判定 */
      if(*readPoint == '-' || *readPoint == '.' ||
               *readPoint == '_' || *readPoint == '~' ||
               ((*readPoint >= '0') && (*readPoint <= '9')) ||
               ((*readPoint >= 'A') && (*readPoint <= 'Z')) ||
               ((*readPoint >= 'a') && (*readPoint <= 'z'))){
      /* エンコードしない場合 */
         dst.push_back(*(readPoint++));
      }
      else{
      /* エンコードする場合 */
         dst.push_back('%');
         x = (*readPoint & 0x00f0) >> 4;
         x = x <= 9 ? x + '0' : x - 10 + 'A';
         dst.push_back(x);
```

```
            x = *readPoint++ & 0x000f;
            x = x <= 9 ? x + '0' : x - 10 + 'A';
            dst.push_back(x);
        }
    }
    return dst;
}
```

図６　作り直したソースコード

図 4 のソースコードでは文字列を配列として取り扱っていたので領域の境界をプログラムで管理する必要があったが，図６のソースコードでは［　c　］ので，バッファオーバフローが発生することはなくなった。

〔修正後の確認と再発防止〕

その後，Q グループでは，作り直したプロトタイプ 1 を試験して，バッファオーバフローが発生しないことを確認した。最後に，F 主任は再発防止策として，従来の A 社におけるC++開発標準ルールを図７に示すように改訂した。

なお，図７中の下線部は，この改訂によって追加した項目である。

1. 識別子
   （省略）
2. 型・定数
   - 共用体の使用を禁止する。構造体の使用は，最小限にとどめる。
   - オブジェクトを生成する場合，できる限り標準 C++ライブラリで定義されたクラスを使用する。
   （省略）
3. 宣言・定義・初期化
   - 直接アクセス可能なグローバル変数は使用しない。必要な場合は，専用のアクセサを使用する。
   - すべてのオブジェクトは，明示的に初期化する。
   （省略）
4. ポインタ・配列
   - ポインタ変数に対する演算操作は禁止する。
   - 文字列処理を行う場合，文字列を格納するための［　d　］の使用を禁止する。
   - 配列又はポインタに添字を指定して書込みを行う場合，［　e　］を毎回実施する。
   （省略）
5. 関数
   - 関数の返却値及び引数に関して，明示的に型を宣言する。
   - 関数内で値を変更する場合を除き，関数の引数であるポインタは const へのポインタとする。
   - 引数としてポインタを受け取った場合，添字指定での［　f　］は禁止する。
   - 文字列処理を行う関数では，処理が成功したときは，原則として処理結果の文字列を返却値として返す。

（以下，省略）

図７　A社におけるC++開発標準ルール（改訂版）

この改訂によって，A 社におけるプログラム開発では本件と類似した問題点を作り込むことがなくなり，より安全なソフトウェアを提供することができるようになった。

設問 1　図6中の［ a ］，［ b ］に入れる適切なメンバ関数名（メソッド名）を解答群の中から選び，記号で答えよ。

解答群

ア　begin	イ　clear	ウ　end	エ　get
オ　set	カ　size	キ　start	ク　tail

設問 2　図4のソースコードで発見された脆弱性について，(1)，(2)に答えよ。

(1)　バッファオーバフローが発生するメモリ領域はどこか。図 4 中の変数名を用いて 20 字以内で述べよ。

(2)　図4の6行目から始まる while 文の副文（ループ本体）の実行において，バッファオーバフローが発生するための条件を，図 4 中の変数名を用いて 60 字以内で述べよ。

設問 3　本文中の［ c ］では，図4で存在したバッファオーバフロー脆弱性が，図6では存在しない技術的理由を述べている。［ c ］に入れる適切な字句を 60 字以内で述べよ。

設問 4　図 7 について，文字列の取扱いで本件と類似した問題点の作り込みを未然に防ぐのに有効となるように［ d ］～［ f ］に入れる適切な字句を，それぞれ8字以内で答えよ。