Dans ce TP nous allons étendre notre bibliothèque `libsocket` pour qu'elle offre des abstractions non seulement pour les serveurs, mais également pour les clients. Dans votre répertoire `TP5` :

- copiez `/pub/2A/ASR-Reseau/libsocket.hh-TP5`, puis renommer le `libsocket.hh`
- copiez `libsocket.cc` depuis le répertoire de votre TP4, puis ajoutez au bout le contenu du fichier `/pub/2A/ASR-Reseau/libsocket.cc-TP5`
- copiez `/pub/2A/ASR-Reseau/hello-client.cc`

Exercice 1. classe Socket

Nous allons commencer par modifier la classe `Socket` pour pour lui ajouter des méthodes utiles aux clients.

`connect(char* buf, int port)`
: demande à établir une connexion au serveur qui tourne sur la machine dont le nom est dans `buf`, et qui écoute sur le `port`.

ATTENTION : lorsqu'on appelle `gethostbyname`, s'il se produit une erreur, le code d'erreur se trouve dans `h_errno`, et la fonction pour obtenir un `char*` vers un message d'erreur approprié pour le code d'erreur n'est plus `strerror` mais `hstrerror`.

Il faudra donc pouvoir distinguer les erreurs qui s'interprètent par `strerror` et celles qui s'interprètent par `hstrerror`. Une manière très simple de distinguer ces deux cas, c'est de stocker le deuxième cas sous une forme négative : si le code est positif, alors il s'interprète par `strerror` s'il est négatif, alors il s'interprète par `hstrerror` (mais il faut bien sur enlever ce signe "moins" avant l'interprétation).

Il vous faudra donc aussi modifier, la méthode `errmsg` pour qu'elle teste si le code est positif ou négatif : dans le premier cas le message d'erreur s'obtient avec `strerror`, dans le second cas il s'obtient avec `hstrerror`. Le formattage reste le même.

`connect(string host, int port)`
: cette variante appelle la précédente en convertissant l'argument `host` de type `string` en `char*` grâce à la méthode `c_str()` des strings.

`read(char* buf, int n)`
: lire sur la socket exactement `n` octets que l'on placera dans la buffer pointé par `buf`. C'est ce qu'on a fait en TD avec la fonction `read_exact`.

`readline(char* buf, int n)`
: lire sur la socket jusqu'à la fin de la ligne (ou de fichier si le bout de la ligne correspond au bout du fichier) et placer les octets dans le buffer pointé par `buf`. Terminer cetrte séquence par `'\0'` pour qu'elle puisse être utilisée comme une chaîne C. On ne mettra pas le caractère `'\n'` de fin de ligne dans le buffer.

L'argument `n` indique la taille maximale du buffer : si la ligne est trop longue, on ne lira que $n-1$ octets, on placera `'\0'` dans le dernier octet, et on retournera la valeur 1 pour indiquer un succès intermédiaire : la lecture s'est bien passée, mais on n'a pas encore lu toute la ligne. Si on a tout lu sans encombre, on retourne 0 comme d'habitude.

`put_int8 (int n)`
`put_int16(int n)`
`put_int32(int n)`
: voici des méthodes pour écrire sur la socket les représentations binaires d'entiers respectivement de

8, 16, ou 32 bits. Si `n` est un entier, on peut obtenir l'octet le moins significatif en masquant tous les autres de la manière suivante : `n & 0xFF`. En hexadécimal, `F` est la valeur 15, c'est à dire 4 bits tous à 1. `0xFF` représente donc 8 bits tous à 1. Avec `(n & 0xFF)` on obtient donc les premiers 8 bits de `n` : les 8 bits les moins significatifs ; c'est à dire l'octet le moins significatif.

Pour obtenir l'octet suivant de `n`, on décale les bits de `n` d'un octet vers les moins significatifs (vers la gauche), et on prend alors l'octet le moins significatif du résultat : `(n>>8) && 0xFF`. Pour des entiers de 8, 16, ou 32 bits, on répète cette procédure pour écrire tous les octets dans l'ordre du moins significatif au plus significatif.

`get_int8 (int& n)`
`get_int16(int& n)`
`get_int32(int& n)`

voici des méthodes pour lire sur la socket les représentations binaires d'entiers respectivement de 8, 16, ou 32 bits. Ce sont les inverses des méthodes `put_int?(n)` précédentes. Elle vont donc devoir lire les octets de `n` du moins significatif au plus significatif.

Si on lit d'abord l'octet le moins significatif dans `b0`, puis l'octet suivant dans `b1`, comment reconstruire l'entier de 16 bits correspondant ? Il faut décaler `b1` d'un octet dans la direction des bits significatif (vers la droite) et il faut alors le concaténer avec `b0`. On peut faire cela de la manière suivante : `b0 | (b1<<8)`

Exercice 2. Classe Client Nous allons créer une classe `Client` offrant une interface simplifiée pour fabriquer un client et le connecter à un serveur en réseau. Cette classe hérite aussi de `Socket` pour intégrer la socket avec laquelle il va se connecter et communiquer avec le serveur. Elle a les méthodes suivantes :

`run(const char* host, int port)`

cette méthode utilise les méthodes de `Socket` pour se connecter au serveur sur la machine `host` et qui écoute sur le `port` donné. Puis elle invoque la méthode `interaction` du `Client` pour interagir avec le serveur.

`run(string host, int port)`

cette variante appelle la méthode précédente en convertissant l'argument `host` de type `string` en `char*` grâce à la méthode `c_str()` des `string`.

`interaction()`

méthode virtuelle (donc redéfinissable dans les classes dérivées) implémentant l'interaction avec le serveur. Cette méthode est "pure" virtuelle : elle *doit* donc être implémentée dans une classe dérivée.

Exercice 3. Compilation de la bibliothèque Vérifiez comme dans le TP précédent que la bibliothèque compile correctement.

Exercice 4. hello-client : Dans cet exercice, vous allez programmer le client que j'ai illustré en classe ; mais vous le ferez en utilisant la bibliothèque `libsocket` que vous venez d'implémenter et en complétant le code du fichier `hello-client.cc`. Le client doit se connecter à un serveur spécifié sur la ligne de commande, lire une ligne envoyée par ce serveur, l'afficher, puis fermer la connexion.

Puis vérifiez que le client compile :

```
c++ -o hello-client hello-client.cc libsocket.o
```

Dans un shell faite tourner le serveur par exemple sur le port 8080 :

```
../TD4/hello-serveur 8080
```

Dans un autre shell, invoquez votre client pour qu'il se connecte à votre serveur :

```
./hello-client localhost 8080
```