Pour réaliser des applications client/serveur en C++, il est utile de commencer par se construire une bibliothèque C++ qu'on pourra réutiliser dans chaque projet. C'est le but de ce TP. La bibliothèque comprend deux fichiers :

- le fichier `libsocket.hh` dont on fera l'include dans chaque application
- le fichier `libsocket.cc` contenant l'implémentation correspondante : on ajoutera `libsocket.o` comme bibliothèque quand on fera l'édition de liens de l'application.

Pour vous aider à démarrer, les fichiers `libsocket.hh` et `libsocket.cc` sont disponibles dans `/pub/2A/ASR-Reseau`. Copiez les dans votre répertoire de travail `TP4`. Il vous faudra compléter le fichier `libsocket.cc` : c'est le but des exercices suivants.

Exercice 1. classe ErrInfo

Que faire quand il se produit une erreur ? Différentes actions peuvent être envisagées :

- ne rien faire
- recommencer l'opération, essayer autrement, ou faire autre chose
- afficher un message sur un terminal (par exemple avec `perror`)
- ajouter un message à un fichier de log
- afficher un message dans une fenêtre graphique
- etc ...

cette décision dépend de l'application et varie d'une application à une autre. Si nous voulons une bibliothèque qui soit utilisable par n'importe quelle application, il faut qu'elle se contente de donner à l'application les informations sur une erreur, et c'est l'application elle-même qui choisira quoi faire de ces informations.

La classe `ErrInfo` est un conteneur destiné à prendre note des informations relatives à une erreur : le nom de la procédure dans laquelle l'erreur s'est produite et le code de cette erreur.

`errset(int code, string proc)`
: prend note, dans l'objet de classe `ErrInfo`, d'une erreur s'étant produite dans la procédure nommée `proc` et caractérisée par le code d'erreur `code`

`errmsg(char* buf, int n)`
: écrit dans le buffer `buf` de taille maximale `n` un message d'erreur approprié à l'erreur dont les informations sont enregistrées ici. Vous vous servirez des fonctions `snprintf` et `strerror` pour formatter ce message d'erreur. On voudrait un message ayant la forme de ceux produit par `perror`, par exemple :

  ```
  bind: permission refusée
  ```

Exercice 2. classe Socket

Nous allons créer une classe `Socket` offrant des méthodes simplifiées pour faire les opérations réseau. Cette classe hérite aussi de `ErrInfo` pour nous permettre de prendre note facilement des informations relatives à une erreur survenant durant une de ces opérations. La classe `Socket` contient un champ `_fd` dans lequel on mettra le descripteur correspondant à la socket. Les méthodes ci-dessous retournent 0 en cas de succès. En cas d'échec elles retournent -1 après avoir utilisé `errset` (voir exercice précédent) pour noter les détails de l'erreur.

socket()
: invoque l'appel système `socket(2)`[1] pour créer une socket dans la domaine de l'internet orientée "flux d'octets," et met son descripteur dans le champ `_fd`. Pour indiquer à C++ que vous voulez appeler la fonction globale `socket` plutôt que la méthode `socket` de la classe, il faut écrire `::socket`

close()
: invoque l'appel système `close(2)` pour fermer la socket et libérer les resources qui lui sont associées.

bind(int port)
: invoque `bind(2)` pour lier la socket au `port` sur toutes les interfaces. Cette méthode remplit une adresse de socket `struct sockaddr_in` (voir `ip(7)`), puis l'utilise dans l'appel à `::bind`

listen(int backlog = 10)
: invoque `listen(2)` pour indiquer qu'on va "écouter" sur cette socket, c'est à dire qu'on va y attendre des demandes de connexion de clients avec `accept`. L'argument est optionel et prend la valeur 10 par defaut quand il est omis.

accept(Socket& client)
: invoque `accept(2)` pour attendre et accepter une connexion d'un client. Il met alors le descripteur de la connexion dans le champ `_fd` de la socket `client` passée en argument.

Nous allons aussi définir des méthodes d'écriture :

write(const char* buf, int n)
: écrire sur la socket exactement `n` octets pointés par `buf`. C'est ce qu'on a fait en TD avec la fonction `write_exact`

write(const char* buf)
: cette variante appelle la méthode précédente en calculant elle-même la longueur de la chaîne de caractères

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

Exercice 3. classe Server

Nous allons créer une classe `Server` offrant une interface simplifiée pour fabriquer un serveur et le mettre en service en réseau. Cette classe hérite de `Socket` pour intégrer la socket sur laquelle on va accepter les demandes de connexion de clients. Elle a les méthodes suivantes :

run(int port)
: crée et lie la socket au `port` sur toutes les interfaces, puis indique qu'on va écouter sur cette socket. Enfin, elle appelle la méthode `loop` pour entamer la boucle de service des clients.

loop()
: méthode virtuelle (donc redéfinissable dans les classes dérivées) implémentant la boucle infinie qui accepte une connexion d'un client, invoque la méthode `interaction` pour traiter cette connexion, puis ferme la connexion.

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

[1] `socket(2)` signifie que la page de manuel est dans la section 2. On l'obtient donc avec la command `man 2 socket`

Exercice 4. Compilation de la bibliothèque

Vérifiez que la bibliothèque peut être compilée :

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

S'il y a des problèmes de compilation, lisez bien les messages d'erreur du compilateur pour comprendre ce que vous devez réparer.

Exercice 5. hello-server

Dans cet exercice, vous allez programmer le serveur que j'ai illustré en classe ; mais vous le ferez en utilisant la bibliothèque `libsocket` que vous venez d'implémenter. Le serveur doit accepter des connexions de clients. Lorsqu'une connexion est acceptée, le serveur écrit `hello` sur la socket de cette connexion, puis ferme la socket et attend la connexion suivante.

Copiez le fichier `hello-server.cc` qui est dans `/pub/2A/ASR-Reseau`. Vous devrez le modifier (les occurences de la chaîne de caractères `???` indiquent les endroits où vous devrez ajouter votre code). Il faut :

- définir une classe `HelloServer` dérivée de `Server`
- cette classe doit offrir une implémentation de la méthode `interaction` qui réalise l'interaction avec le client décrite ci-dessus
- le `main` doit créer un object `HelloServer` et le mettre en service sur le port précisé en argument sur la ligne de commande du programme

Vérifiez que le serveur compile :

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

Faites le tourner par exemple sur le port 8080 :

```
./hello-server 8080
```

Dans un autre shell, vérifiez que vous pouvez vous connecter à ce serveur et que vous recevez bien son message `hello` :

```
netcat localhost 8080
```