1. Listes chaînées récursives

```
/**** LISTES CHAINEES RECURSIVES ****\
|***********************************|
\**** Par Jeremie Osmont [6-OAD] ****/


// Flots d'entree-sortie :
#include <iostream.h>
```

a) Définitions, constructeurs et destructeurs

```
// Definition de la classe maillon de liste chainee :
class Maillon
{
    int info;
    Maillon *suiv;
public:
    Maillon();
    Maillon(int);
    ~Maillon();
    void afficher();
    Maillon * inserer(int);
    Maillon * supprimer(int);
};


// Constructeur par defaut :
Maillon::Maillon()
{
    info=0;
    suiv=NULL;
}


// Constructeur a un argument :
Maillon::Maillon(int inf)
{
    info=inf;
    suiv=NULL;
}


// Destructeur recursif :
Maillon::~Maillon()
{
    // Efface le maillon suivant, s'il existe,
    // de cette facon, on detruit tous les maillons
    // qui se suivent, jusqu'au dernier :
    if(suiv) delete suiv;
}


// Definition de la classe liste chainee :
class Liste
{
    Maillon * debut;
public:
    Liste();
    ~Liste();
    void afficher();
    void inserer(int);
    void supprimer(int);
};


// Constructeur par defaut :
Liste::Liste()
{
    debut=NULL;
}


// Destructeur :
Liste::~Liste()
{
    // Utilise le destructeur recursif de maillon,
    // qui se charge de detruire toute la liste de
    // proche en proche :
    if(debut) delete debut;
}
```

b) Affichage des éléments d'une liste

```
// Affichage du contenu d'une liste :
void Liste::afficher()
{
    // Appel l'affichage de maillon, qui s'occupe
    // de tout :
    if(debut) debut->afficher();
    cout << endl;
}

void Maillon::afficher()
{
    // Affiche l'information en cours, puis affiche le
    // maillon suivant, ainsi on affiche toute la liste
    // de proche en proche :
    cout << info << ';';
    if(suiv) suiv->afficher();
}
```

c) Insertion d'une valeur

```
// Insertion d'une information a sa place et sans redondance :
void Liste::inserer(int i)
{
    if(debut)
        // On fait appel a une insertion recursive de maillon :
        // Remarque : cette technique un peu particuliere de
        // modifier le debut par le retour de la fonction inserer
        // permet de ne pas avoir a gerer le cas particulier
        // ou il faudrait inserer le maillon en tete.
        debut=debut->inserer(i);
    else
        // A moins que la liste soit vide, ou la il faut bien
        // inserer le maillon 'a la main':
        debut=new Maillon(i);
}

Maillon * Maillon::inserer(int i)
{
    if(info==i)
        // L'information existe deja, rien a faire :
        return this;
    if(info>i)
    {
        // On vient de depasser l'information, donc il
        // est temps d'inserer notre maillon :
        Maillon * tmp=new Maillon(i);
        // Ici on place notre maillon avant this :
        tmp->suiv=this;
        // Le fait de retourner le pointeur de notre maillon
        // tout neuf permet dans l'appel precedent de modifier
        // le suiv du maillon precedent :
        return tmp;
    }
    else
        // Sinon cas general, il nous faut continuer a inserer
        // c'est-a-dire a cherche l'emplacement ou inserer l'info :
        if(suiv)
        {
            // Voir trois lignes au dessus : ceci permet de ne pas
            // avoir a se trimbaler un maillon precedant celui sur
            // lequel on travaille. On exploite tout le potentiel
            // de la recursivite :
            suiv=suiv->inserer(i);
            return this;
        }
        else
        {
            // Ici il se trouve que l'on a atteint la fin de la
            // liste, inserons donc le maillon en fin. Le renvoi
            // de this connectera ensuite ce maillon en fin a
            // proprement parler :
            suiv=new Maillon(i);
            return this;
        }
}
```

d) Suppression d'une valeur

```
// Suppression d'une information dans la liste :
void Liste::supprimer(int i)
{
    if(debut) debut=debut->supprimer(i);
}

Maillon * Maillon::supprimer(int i)
{
    if(info==i)
    {
        // Nous avons trouve l'information a supprimer,
        // procedons maintenant a son effacement :
        Maillon * tmp=suiv;
        // On met le suivant a NULL, pour ne pas detruire
        // tous les maillons suivants (destructeur recursif) :
        suiv=NULL;
        // Attention, il se peut que le compilateur refuse
        // le 'suicide', mais avec Visual C++, ca marche :
        delete this;
        // Puis retour du maillon suivant, pour le 'linkage' :
        return tmp;
    }
    else
        if(suiv && info<i)
        {
            // Nous n'avons pas atteint la fin de la liste,
            // ni depasse la valeur recherchee, donc on continue
            // la recherche par un appel recursif :
            suiv=suiv->supprimer(i);
            return this;
        }
        else
        {
            // La valeur est deja presente, on ne supprime rien
            // et on a fini le parcours de la liste :
            return this;
        }
}
```

e) Programme de lancement

```
// Programme de test
int main()
{
    // Cree la liste de travail :
    Liste l;

    // Test de l'insertion de base :
    cout << "Entrez une serie d'entiers separes par ENTREE :"
         << "\n(0 pour arreter )\n";
    int ent=1;
    while(ent)
    {
        cout << ">"; cin >> ent;
        if(ent) l.inserer(ent);
    }
    cout << "\nValeurs entrees : ";
    l.afficher();

    // Test de la suppression :
    cout << "\nEntrez les nombres a supprimer :"
         << "\n0 pour arreter !\n";
    ent=1;
    while(ent)
    {
        cout << ">"; cin >> ent;
        if (ent) l.supprimer(ent);
        cout << "Valeurs restantes : ";
        l.afficher();
    }
    cout << endl;
    return 0;
}
```

2. Le problème de Flavius Josephus

```
// A ajouter au programme precedent (mettre les membres prives
// des classes en public pour que ca fonctionne).

// Remplissage d'une liste circulaire de n entiers de 1 a n :
Liste * remplissage(int n)
{
    Liste * res=new Liste();
    res->inserer(n);
    Maillon * dernier=res->debut;
    // Remplissage de la liste :
    for(int i=n-1; i>0; i--)
        res->inserer(i);
    // Puis relie la fin au debut : la boucle est bouclee !
    dernier->suiv=res->debut;
    return res;
}


// Elimination du m-ieme, jusqu'au dernier :
Liste * elimination(Liste * l, int m)
{
    Liste * elimines=new Liste();
    // Insere un maillon 'a blanc', afin de ne pas
    // avoir a traiter le cas particulier du premier
    // maillon. On supprimera ce -1 a la fin.
    elimines->inserer(-1);
    Maillon * dernier_elimine=elimines->debut;
    // Cette boucle for un peu speciale permet de positionner zero
    // juste avant le maillon debut de la liste cycle :
    for(Maillon *zero=l->debut;zero->suiv!=l->debut;zero=zero->suiv);
    // Boucle d'elimination (on ne s'arrete que lorsque la liste
    // ne contient plus qu'un element connecte sur lui-meme, ce
    // sera le dernier survivant) :
    while(zero!=zero->suiv)
    {
        // Positionne le pointeur zero sur le maillon juste
        // avant le m-ieme :
        for(int i=1;i<m;i++)
            zero=zero->suiv;
        // Extrait le maillon zero->suiv et le connecte
        // en fin de liste elimines :
        dernier_elimine->suiv=zero->suiv;
        // Restaure le lien dans l :
        zero->suiv=zero->suiv->suiv;
        // Puis positionne dernier_elimine sur le dernier :
        dernier_elimine=dernier_elimine->suiv;
    }
    // Termine la liste des elimines :
    dernier_elimine->suiv=NULL;
    // Ici on efface le premier maillon -1 :
    Maillon * tmp=elimines->debut;
    elimines->debut=elimines->debut->suiv;
    // On n'oublie pas ceci, sinon on detruira
    // tous nos precieux maillons a la queue leu leu :
    tmp->suiv=NULL;
    delete tmp;
    // Notre liste l ne contient plus qu'un element,
    // c'est zero, c'est quand meme mieux si debut
    // pointe vers cet element...
    l->debut=zero;
    return elimines;
}


// Programme de test
int main()
{
    int n,i;
        cout << "Entrez le nombre d'individus de la ronde : ";
    cin >> n;
    if(n<0) n=1;
    Liste * cycle=remplissage(n);
    cout << "Quel individu voulez-vous eliminer a chaque tour ? ";
    cin >> i;
    if(i<0) i=1;
    Liste * mauvais=elimination(cycle,10);
    cout << "\nVoici la liste des elimines dans l'ordre :" << endl;
    mauvais->afficher();
    Maillon * tmp=cycle->debut;
    // Sans oublier d'effacer nos listes qui sont creees
    // par allocation dynamique :
    delete mauvais;
    // Ces deux lignes sont necessaires pour rompre la
    // liste circulaire :
    cycle->debut->suiv=NULL;
    cycle->debut=tmp->suiv;
    cout << "\nL'heureux survivant est : " << tmp->info << endl;
    delete cycle;
    return 0;
}
```