Chapitre 7: Maintenance

INF3135

Construction et maintenance de logiciels

Alexandre Blondin Massé

Université du Québec à Montréal

v251

Construction

Définition (extraite de Wikipedia)

« La maintenance du logiciel (ou maintenance logicielle) désigne, en génie logiciel, les modifications apportées à un logiciel après sa mise en oeuvre, pour en corriger les fautes, en améliorer l'efficacité ou autres caractéristiques, ou encore adapter celui-ci à un environnement modifié. »

Mots-clé

- Modifications
- Corriger
- Améliorer
- Efficacité
- Adapter

Table des matières

Généralités

Maintenance de logiciels

- Modification d'un logiciel **déjà livré**
- Souvent pour **corriger des erreurs** ou des **bogues**
- Parfois aussi pour **améliorer** le fonctionnement, les performances, faire une refactorisation du code, traiter des nouveaux cas, etc.
- On évalue à **20%** les activités de développement de nouveaux logiciels
- Par opposition à **80%** qui concernent la maintenance: correction de bogues, adaptation de logiciels, amélioration

Cas des entreprises non informatiques

- Dans le cas des compagnies qui n'ont pas une **vocation informatique**, la maintenance occupe une place plus importante encore
- Il y a en particulier très peu de développement logiciel
- Souvent, on doit intégrer des logiciels déjà existants
- Par exemple, développer des interfaces entre différents logiciels, à l'aide de **scripts**
- Le **besoin** est souvent flou et mal défini
- Développement de **rapports** et autres outils de consultation des **données**

Coût de la maintenance (1/2)

Préparation

- Il est beaucoup plus difficile de modifier un système en **activité** qu'un système en **développement**
- Il faut en particulier mesurer les **impacts** sur les opérations et limiter ces impacts
- Souvent, les **responsables** de la maintenance ne sont pas ceux qui ont participé au développement

Temps d'apprentissage

Il faut prévoir un **temps d'apprentissage** au niveau:

- **Fonctionnel**: Que fait le logiciel? À quoi sert-il?
- **Structurel**: Comment le code est-il organisé? Quelle est la stratégie sous-jacente?
- **Technique**: Langages de programmation, outils logiciels, styles, etc.

Coût de la maintenance (2/2)

Plus un programme est vieux

- plus il a **subi** des activités de maintenance,
- plus il est **complexe** à modifier

Plusieurs critères font en sorte qu'il est préférable de garder un vieux système que de le changer

- Performance **connue** et satisfaisante du système
- **Coût** d'investissement trop élevé
- **Risque** trop élevé de changer de système
- **Exemple**: beaucoup de systèmes bancaires et financiers utilisent encore aujourd'hui Cobol et Fortran

Quoi faire pour intervenir? (1/2)

Avant de programmer:

- On doit **comprendre le logiciel**, au niveau fonctionnel et structurel
- Comprendre les modifications **demandées**
- Évaluer de quelles façons ces **modifications** peuvent être apportées
- Si possible, proposer une ou plusieurs approches de **mise en oeuvre** des modifications demandées
- Évaluer l'**impact** de la réalisation de ces modifications
- Évaluer si une **refactorisation** préalable est nécessaire
- Quelles **fichiers** et **sections** de code sont touchées?

Quoi faire pour intervenir? (2/2)

- Choisir la solution la **moins coûteuse**, la **moins complexe** et la **plus facile** à maintenir à long terme
- L'**implémenter** en respectant le plus possible le style de programmation
- Mettre à jour les **plans de tests** (unitaires et intégrés) ou en ajouter si inexistants
- **Documenter** les modifications apportées, en particulier, décrire le problème et la solution apportée
- Si nécessaire, donner une formation aux utilisateurs sur les nouvelles fonctionnalités
- **Coordonner** la mise en production

Modifier un logiciel

Rôle du gestionnaire de versions

- En plus de simplifier la **gestion des versions** d'un logiciel, un logiciel tel que Git est souvent bien intégré avec d'autres outils
- Par exemple, les **plateformes** telles que Github, Bitbucket et GitLab proposent plusieurs services (parfois payants) pour faciliter la maintenance

Quelques exemples:

- Demande d'**intégration** (*pull requests* ou *merge requests*)
- Questions ou **problématiques** soulevées (*issues*)
- **Tickets** (Trac, Jira, etc.)
- **Correctifs** (*patch*, ou *diff*)

Problèmes (*issues*)

- Les plateformes Github, Bitbucket et GitLab permettent toutes de rapporter des **issues**
- Une *issue* est essentiellement un **rapport** et, éventuellement, une **discussion**, à laquelle plusieurs personnes peuvent participer
- On lui attribue un **identifiant unique**
- Elle peut être **assignée** à une ou plusieurs personnes responsables
- On peut lui attribuer une ou plusieurs **étiquettes**

Quoi mettre dans une *issue*?

- Elle peut concerner une **question**, la découverte d'un **bogue**, la demande d'intégration d'une **nouvelle fonctionnalité** (*feature request*), une discussion **préalable** à une requête d'intégration, avec demande de commentaires
- Il faut utiliser adéquatement le format **Markdown**
- Si c'est un bogue, donner une **suite d'étapes** permettant de le **reproduire**
- Donner le plus possible d'informations: la **version** testée (numéro ou *commit* exact), la **distribution** sur laquelle le bogue a été observé, une **capture d'écran**, un **vidéo**, etc.

Requête d'intégration (*pull/merge requests*)

C'est quoi?

- Une **requête d'intégration** est essentiellement un *patch* qu'on applique à un logiciel
- Elle peut être divisée en **plusieurs** *commits*

Information

- Quel type de modification? Correction d'un bogue? Ajout d'une fonctionnalité? de documentation?
- Souvent, on réfère à une *issue* reliée ou même une autre requête
- On peut aussi identifier une **personne**

Arbitrage

- Souvent un processus d'arbitrage par les **pairs**
- On peut la **mettre à jour** en fonction des commentaires

Simplification

- Lorsqu'un projet atteint une certaine **maturité**, le responsable exige souvent de produire des requêtes d'intégration **simplifiées**
- Par exemple, il peut demander de produire des *commits* propres, effectuant une tâche **unique**
- Les messages de *commits* sont particulièrement importants.
- Il est fréquent qu'on vous demande de **restructurer** vos *commits*
- Une opération très utile pour cela: `git rebase`

Approches programmatives

Justesse d'un programme

- Une fonction est **juste** (ou **correcte**) si elle ne retourne jamais de résultat inexact
- Autrement dit, une fonction est juste quand elle retourne un résultat exact **peu importe** les paramètres en entrée
- En particulier, on préfère **ne rien retourner** que retourner quelque chose de faux

Exemple

- Fonction **mal implémentée**
- Cas particuliers (parfois rares) **oubliés**
- **Algorithmes probabilistes**: on accepte que le résultat soit parfois erroné, mais avec une probabilité faible (test de primalité en cryptographie)

Robustesse d'un programme

- Une fonction est **robuste** si elle se termine toujours (sans faire « planter » le programme)
- La **justesse** et la **robustesse** sont des notions indépendantes

En C, c'est une fonction qui

- ne déclenche pas d'erreur de segmentation
- ne déclenche pas d'erreur de bus (*bus error*)
- ne déclenche pas d'assertion (on va y revenir)

Approches programmatives

On s'assure de la justesse et de la robustesse à l'aide de deux approches programmatives complémentaires

- La programmation **par contrat**
- La programmation **défensive**

Programmation par contrat

- Utilisation de **spécifications**
- Validation par des assertions

Programmation défensive

- Supposition des pires utilisations
- Communication de l'erreur à l'aide d'exceptions ou de codes d'erreur

Programmation par contrat

- Basé sur un modèle **client-fournisseur** (*client-provider*)
- Le **client** est le module appelant une fonction d'un autre module, le **fournisseur**

Contenu d'un contrat

- **Préconditions**: le client doit les satisfaire lorsqu'il fait appel au fournisseur (`@pre`)
- **Postconditions**: le fournisseur doit les satisfaire une fois le service rendu (`@post`)

Exemples

```
/**
 * Calcule l'inverse multiplicatif du nombre réel x
 *
 * @param x   Le nombre réel à inverser
 * @return y  L'inverse du nombre réel x
 * @pre       x != 0.0
 * @post      x * y == 1.0
 */
float inverse(float x);

/**
 * Insère la paire clé-valeur (k, v) dans la table associative.
 *
 * @param h  La table associative
 * @param k  La clé
 * @param v  La valeur
 * @post     treemap_has_key(t, k)
 */
void treemap_insert(Treemap* t, int k, int v);
```

Préconditions et postconditions

Avantages

- Le module **client** peut considérer les services offerts par le fournisseurs comme une **boîte noire** en ne considérant que les préconditions qui doivent être satisfaites
- De la même façon, le module **fournisseur** n'a pas à se soucier de l'utilisation qui sera faite de ses services, mais seulement de satisfaire les **postconditions**

Philosophie

Les modules fournisseurs de service devraient:

- exiger les **préconditions** les plus strictes
- accepter les **postconditions** les plus faibles

Les assertions

Si une **précondition** n'est pas satisfaite:

- il y a un **bogue** dans le code **client**
- le service ne devrait pas être fourni

Si une **postcondition** n'est pas satisfaite:

- il y a un **bogue** dans le code **fournisseur**
- le programme client devrait être avorté

Langage C

- Une bibliothèque très pratique: `assert.h`
- Aussitôt qu'une assertion n'est pas vérifiée, le programme est avorté

Deux approches

On distingue deux types de philosophie:

Approche exigeante

- Les préconditions sont vérifiées par des **assertions**
- Plus appropriée pour les traitements **internes** à un module ou un système

Approche tolérante

- Les préconditions sont vérifiées par des **structures de contrôle** et subissent un traitement en conséquence
- Plus appropriée lorsque les données proviennent de l'**extérieur** du système

Utilité des assertions (1/2)

- Mettre en place des **contrats**
- Signaler que certaines sections de code ne devraient pas être appelées

```
if (liste_taille(liste) == 0)
  ...
else if (liste_taille(liste) > 0)
  ...
else
  assert(FAUX && "taille de la liste negative");
```

- Signaler une portion de code **pas encore développée**

```
void creer_graphe() {
  // TODO
  assert(false && "non implemente");
}
```

Utilité des assertions (2/2)

- Les assertions permettent de détecter les cas **théoriquement impossibles**
- Si une **assertion** est activée, cela signifie qu'il y a un **bogue** dans le programme
- Les assertions ne devraient pas avoir d'effets de bord (`const`)

Exemples

- Un **arbre AVL** devrait demeurer **équilibré** (`is_balanced`) après une **insertion**, une **suppression**
- Un **tableau trié** devrait demeurer **trié** (`is_sorted`) après une **insertion** et une **suppression** (on parle alors d'**invariants**)

Contre-exemple

- Mauvaise entrée de l'utilisateur
- Format de fichier non conforme
- Pas de mémoire disponible

Utilisation des assertions

Activation des assertions:

- En phase de développement et de test
- Permet de vérifier les **contrats**, la **non-exécution** des sections **interdites**
- En principe, les assertions ne devraient **jamais** être déclenchées

Désactivation des assertions:

- Pour la **livraison** du code une fois testé
- Il suffit de passer l'option `-DNDEBUG` lors de la compilation

Exemple complet

Reprenons l'exemple de la structure de données `Treemap`

```
void treemap_initialize(Treemap* t);
char* treemap_get(const Treemap* t, const char* key);
void treemap_set(Treemap* t, const char* key,
                 const char* value);
bool treemap_has_key(const Treemap* t,
                     const char* key);
void treemap_print(const Treemap* t);
void treemap_delete(Treemap* t);
```

Quelles assertions pourrait-on ajouter?

Programmation défensive

Idée générale

Type d'approche qui préconise d'avertir le module appelant aussitôt qu'une erreur est rencontrée

Plusieurs modes de communication

- En déclenchant une **exception**
- En retournant une **valeur spéciale**: `-1` pour un entier, `NULL` pour un pointeur, etc.
- En retournant un **code d'erreur** (approche typique en C)
- Pour une fonction, cela se traduit par le fait que **tous les cas** d'erreur possibles sont vérifiés par des structures conditionnelles

Exceptions

- Sont soulevées lorsque des événements **indésirables**, mais **prévisibles** surviennent;
- Par opposition, les assertions sont réservées aux événements supposément **impossibles**

Exemples

- Une structure de données est à **pleine capacité**
- Il n'y a plus de **mémoire disponible**
- L'utilisateur entre une **chaîne de caractères** plutôt qu'un nombre entier

Signalement d'exceptions

- Affichage de messages d'erreur sur `stderr`
- Écriture de messages d'erreur dans un journal (*log file*)
- Terminer l'exécution d'un programme, en cas d'**erreur fatale**:
- Dans `stdlib.h`: constantes `EXIT_SUCCESS` et `EXIT_FAILURE`
- Peuvent être utilisés en combinaison avec

```
void exit(int statut);
```

- Équivaut à retourner `statut` dans le `main`
- Cette fonction ferme les fichiers ouverts et vide les tampons

Signaler une exception à l'aide d'un code d'erreur

- Lorsque l'erreur n'est pas fatale, on retourne une valeur d'**état** ou un **code d'erreur**
- La valeur `0` indique que tout s'est bien déroulé
- Sinon, on retourne une valeur appropriée en utilisant des constantes
- Autre stratégie, utiliser une valeur spécifique pour indiquer un problème (`-1`, `NULL`, etc.)
- **Problème**: n'indique rien sur l'erreur survenue
- Potentiellement **risqué** avec d'autres types de données

Programmation défensive/par contrat

Programmation par contrats

- Devrait être utilisée dans les modules auxiliaires
- Ne devrait pas dépendre des données provenant de l'extérieur du programme
- Se traduit par des **préconditions** et des **postconditions**
- Vérifiées par des **assertions**

Programmation défensive

- Peut-être utilisée dans tous les modules
- Devrait en particulier être présente lorsque les données proviennent de l'extérieur du programme
- Se traduit par des **exceptions**, des **stuctures conditionnelles**, par des code d'erreurs, etc.

Analyse de programmes

Analyse de programmes

Définition (extraite de Wikipedia)

« In computer science, program analysis is the process of analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness. Program analysis focuses on two major areas: program optimization and program correctness. The first focuses on improving the program's performance while reducing the resource usage while the latter focuses on ensuring that the program does what it is supposed to do. »

Mots-clé

- Comportement
- Justesse
- Robustesse
- Sécurité
- Optimisation

Graphes de flux

Définitions

- **Sommet**: une suite d'instructions sans branchement (instructions séquentielles)
- **Arc**: représente un lien de contrôle entre deux sommets
- **Source**: sommet n'ayant aucun **prédécesseur**
- **Puits**: sommet n'ayant aucun **successeur**

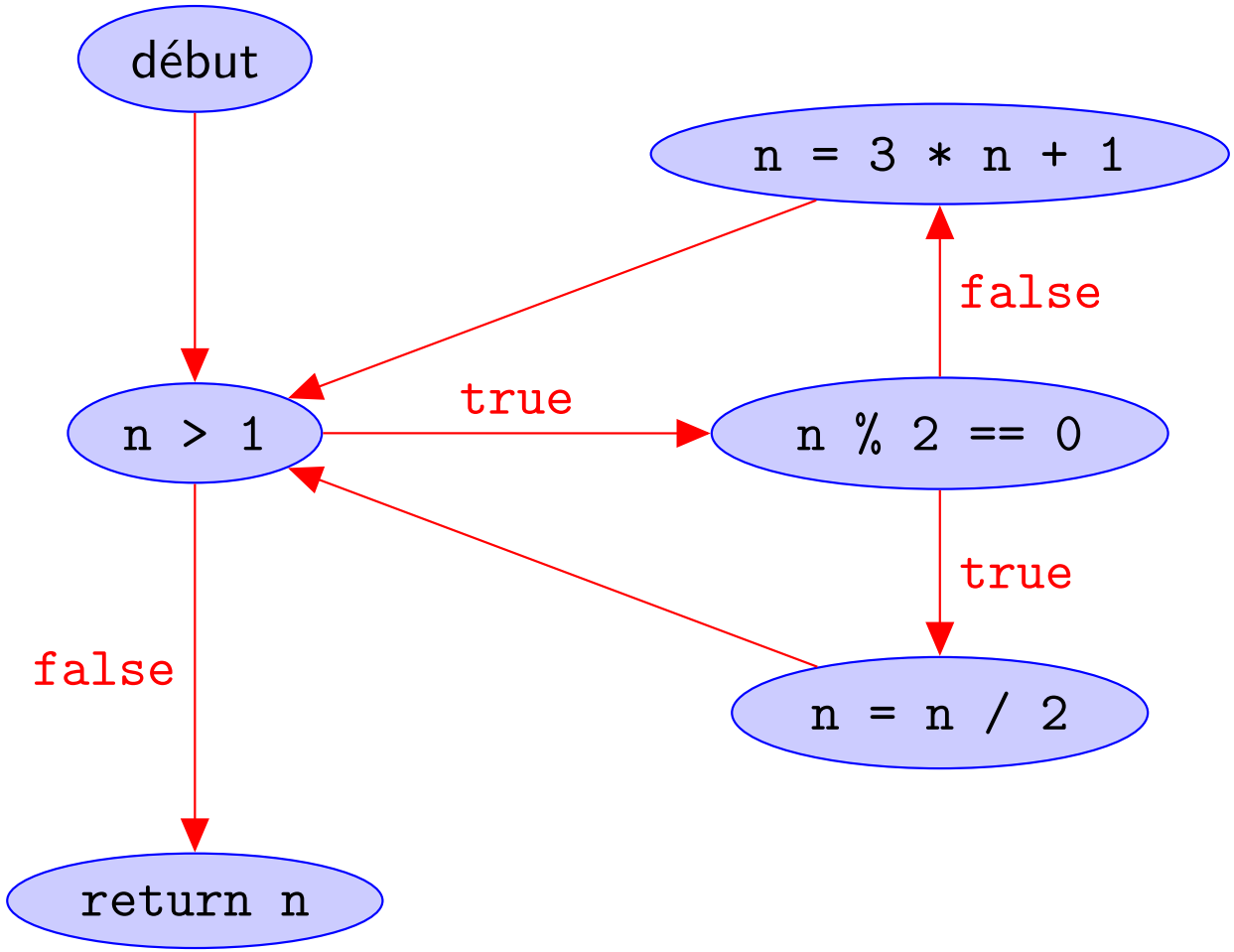
Graphe de flux

- Graphe représentant un bout de programme
- Ayant une **source unique** et
- Un **puits unique**

La fonction `syracuse` (1/2)

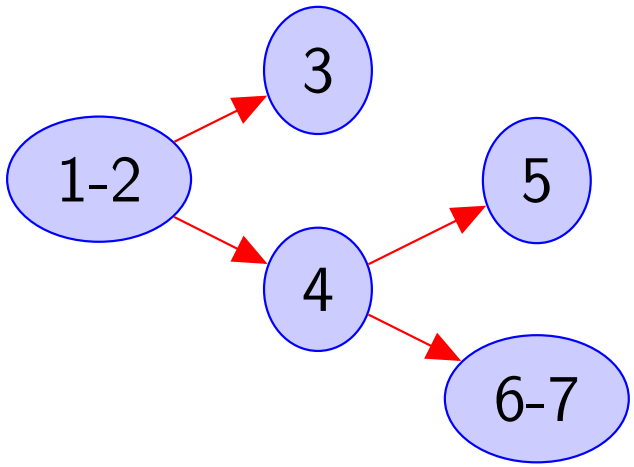
```
1  unsigned int syracuse(unsigned int n) {
2    while (n > 1) {
3      if (n % 2 == 0)
4        n = n / 2;
5      else
6        n = 3 * n + 1;
7    }
8    return n;
9  }
```

La fonction `syracuse` (2/2)

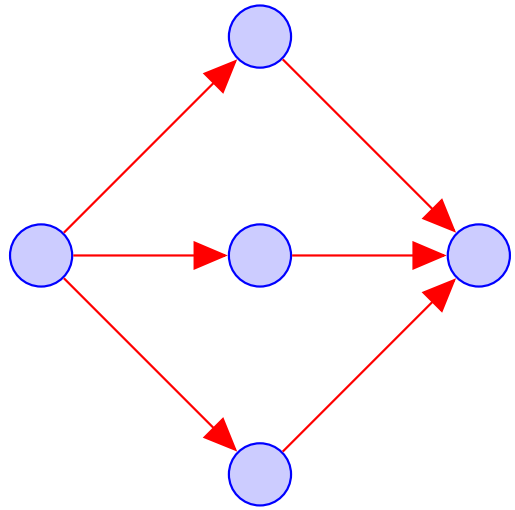


Représentation compacte

- Les **branchements binaires** peuvent être représentés par des **branchements n-aires**
- Par exemple, une structure `if-elseif-else` peut être représentée par un branchement **ternaire**



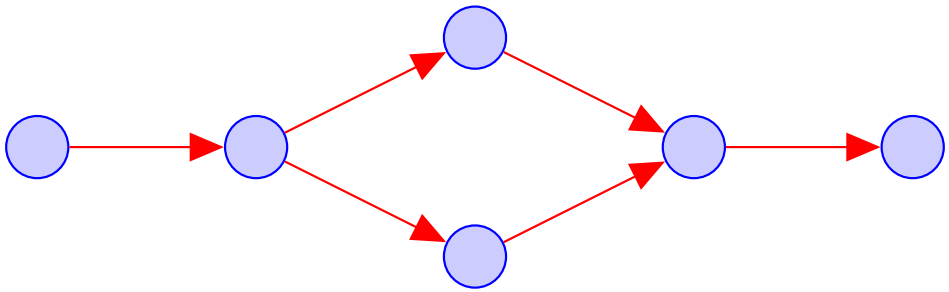
binaire



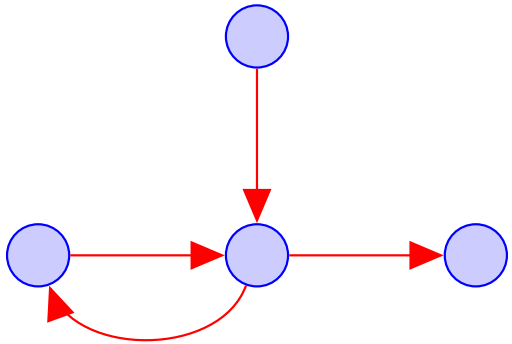
ternaire

Graphes élémentaires (1/2)

`if-else`

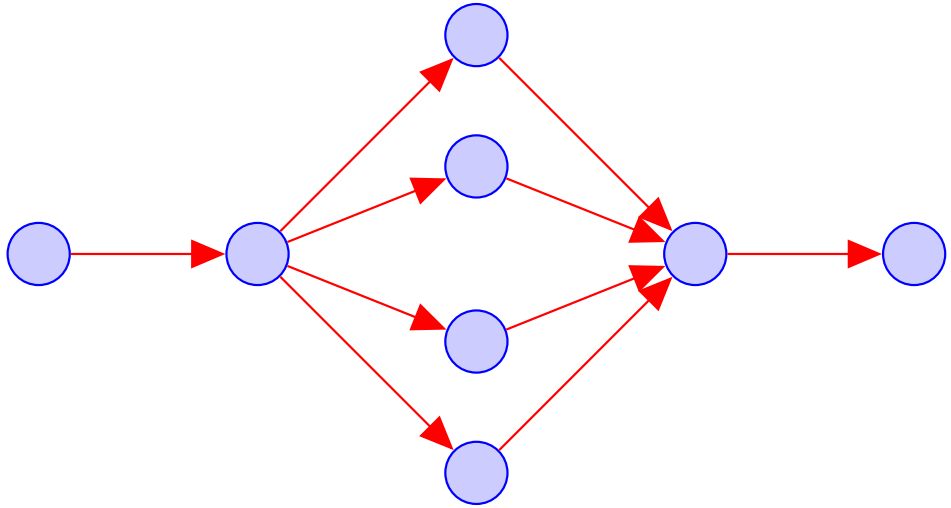


`while`

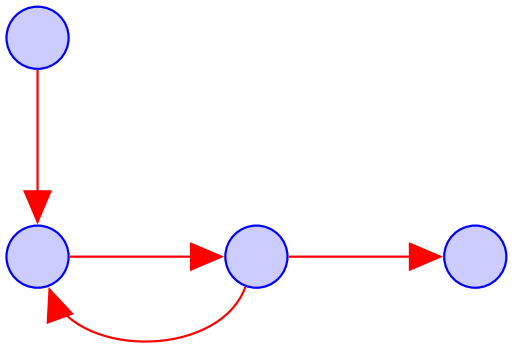


Graphes élémentaires (2/2)

`switch-default`



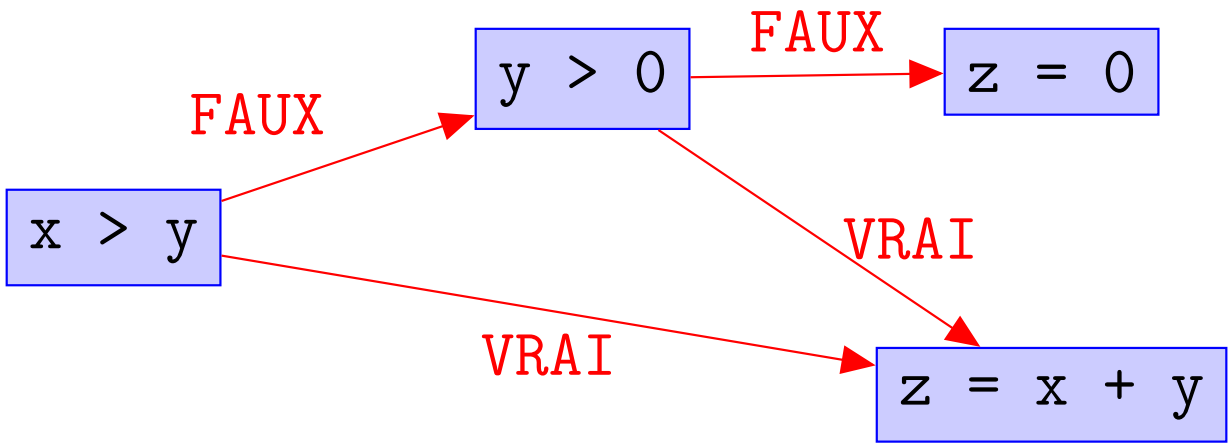
`do-while`



Expressions booléennes et branchements

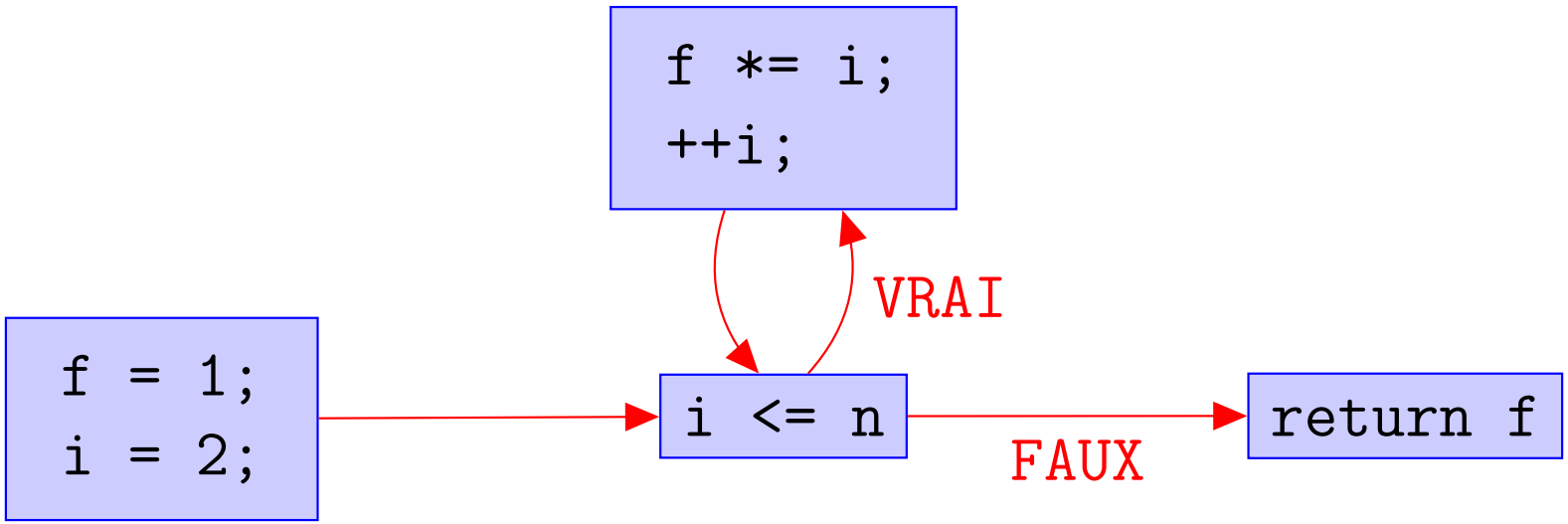
- **Attention!** Si les **conditions** sont compliquées, les graphes doivent être plus détaillés pour tenir compte de tous les **scénarios**
- Autrement dit, une ligne peut contenir plusieurs **instructions**, **branchements** ou **conditions**

```
if (x > y || y > 0)
  z = x + y;
else
  z = 0;
```



Exemple avec une boucle

```
int factorielle(int n) {
  int i, f = 1;
  for (i = 2; i <= n; ++i)
    f *= i;
  return f;
}
```



Instructions `return` multiples

- Souvent, il y a plusieurs instructions `return`
- Pour dessiner le graphe de flux, on doit d'abord « réécrire » le programme pour qu'il n'y ait qu'un seul point de **sortie**

Exercice

Dessiner le graphe de flux de la fonction suivante:

```
 1  unsigned int nombre_jours(unsigned int m,
 2                            unsigned int a) {
 3    if (m == 1 || m == 3  || m == 5 || m == 7 ||
 4        m == 8 || m == 10 || m == 12) {
 5      return 31;
 6    } else if (m == 4 || m == 6 || m == 9 || m == 11) {
 7      return 30;
 8    } else if (a % 4 != 0 ||
 9               a % 100 == 0 && a % 400 != 0) {
10      return 28;
11    } else {
12      return 29;
13    }
14  }
```

Complexité cyclomatique

Chemins

- On considère tous les chemins possibles
- **Commençant** à la source et
- **Terminant** au puits

Couverture

On dit qu'un ensemble de chemin $\mathcal{C}$ forme une **couverture** d'un graphe de flux G s'il recouvre tous les **sommets** et tous les **arcs** de G

Complexité cyclomatique

- Mesure de la **complexité** d'un bout de programme P
- Soit G le graphe de flux de P
- **Complexité cyclomatique** $=$ cardinalité minimale d'une couverture $\mathcal{C}$ de G

Complexité cyclomatique et planarité

Définitions

- Un graphe est dit **planaire** si on peut le dessiner dans le plan sans qu'il y ait de croisement d'arcs
- Une **face** d'un graphe planaire est une région connexe du plan induite par le graphe

Theorème

Soit $G = (V, A)$ un graphe de flux d'un programme. Alors

- G est **planaire**
- La **complexité cyclomatique** de G est égale à $f = a - v + 2$, où v, a et f sont respectivement le nombre de sommets, d'arcs et de faces de G (incluant la face extérieure infinie)

Exercice

```
 1  bool est_palindromique(const char *s) {
 2    unsigned int i = 0;
 3    unsigned int j = strlen(s) - 1;
 4    while (i < j) {
 5      while (!isalpha(s[i])) ++i;
 6      while (!isalpha(s[j])) --j;
 7      if (tolower(s[i]) != tolower(s[j])) {
 8        return false;
 9      }
10      ++i;
11      --j;
12    }
13    return true;
14  }
```

- Dessiner le graphe de flux
- Calculer la complexité cyclomatique
- Proposer un cadre de tests minimal qui couvrent tous les branchements

Limites de la complexité cyclomatique (1/2)

- Ne permet pas de détecter **toutes les erreurs**
- **Exemple:** une couverture minimale peut ne pas vérifier si une **division par zéro** survient

```
int fonction(int x, int y) {
  int z;
  if (x != 0)
    z = y - x;
  else
    z = 3;
  if (y == 5)
    return z;
  else
    return z / x;
}
```

Limites de la complexité cyclomatique (2/2)

- Basée sur la **structure** des graphes et non sur la **sémantique** du programme
- Certains enchaînements peuvent ne jamais **arriver**, puisque certains chemins ne correspondent pas à un test réalisable
- Insuffisant également en cas de **boucles**, car une boucle peut être parcourue un nombre variable de fois
- En revanche, une **couverture de branchements** permet de détecter plusieurs erreurs
- De plus, il est généralement **impossible** ou beaucoup trop long de tester tous les parcours possibles
- Un avantage est qu'il est possible d'**automatiser** ce type de couverture

Optimisation de programmes

Optimisation de programmes

Définition (extraite de Wikipedia)

« In computer science, program optimization, code optimization, or software optimization is the process of modifying a software system to make some aspect of it work more efficiently or use fewer resources.[1] In general, a computer program may be optimized so that it executes more rapidly, or to make it capable of operating with less memory storage or other resources, or draw less power. »

Plusieurs niveaux

Du plus haut vers le plus bas

- **Design**: organisation, architecture du système, utilisation des ressources, identification de la latence
- **Structures de données et algorithmes**: données manipulées, opérations à supporter, complexité temporelle et spatiale, cas usuels, pires cas
- **Code source**: utilisation des bibliothèques standards, expressions idiomatiques
- **Construction**: utilisation d'options et de directives adaptées
- **Compilation et assemblage**: dépend du compilateur choisi

Quand optimiser?

- L'optimisation peut réduire la **lisibilité** du code
- Affecte la **maintenance** et le **débogage**
- Attention à l'**optimisation prématurée**:

> *« We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. »*
>
> — *Donald Knuth*

- Prioriser le **design**, les **structures de données** et les **algorithmes**
- Le **compilateur** fera le reste

Options de GCC

Le compilateur GCC offre plusieurs options d'optimisation:

- `-O0`: réduit le temps de compilation (défaut)
- `-O1`: optimise la taille et le temps d'exécution
- `-O2`: optimise encore plus
- `-O3`: optimise encore beaucoup plus
- `-Os`: optimise la taille
- `-Oz`: optimise la taille plutôt que la vitesse
- `-Ofast`: optimise quitte à ne pas respecter strictement le standard
- Voir documentation officielle pour plus de détails

Exemple: évaluation de performance

- Soit $G = (V, E)$ un graphe simple (non orienté) de n sommets et m arêtes
- Soit $U \subseteq V$
- On dit que U est une **couverture (des sommets par les arêtes)** (en anglais, *vertex cover*) si pour toute arête $\{u, v\} \in E$, on a $u \in V$ ou $v \in V$
- On souhaite **compter** le nombre de couvertures de taille **minimale** de G.

Voir dépôt GitLab