Olena: Présentation des types de données

Nicolas Burrus <nicolas.burrus@lrde.epita.fr>

Journée Olena du LRDE, 18 Février 2003

Table des matières

Table des matières

Introduction

Types de données ?

$\Rightarrow$ Types de base pour représenter les points d'une image.

- Scalaires
  - ▷ *Entiers, flottants*
- Complexes
  - ▷ *Représentation polaire, rectangulaire*
- Types énumérés
  - ▷ *Type binaire, labels*
- Types vectoriels
  - ▷ *Vecteurs, matrices*

Motivations

- Types builtins du C++ dangereux
  - ▷ Aucune vérification d'intervalle aux affectations

```
int i = 256;
unsigned char c = i; // c == 0
```

  - ▷ Opérations arithmétiques sans contrôles

```
unsigned int i = UINT_MAX;
unsigned int j = 5;
unsigned long long k = i + j; // k == 4
```

Objectifs

- Assurer une sécurité maximale sur les valeurs manipulées.
  - ▷ Vérification de débordement systématique, statique ou dynamique.
- Efficacité
  - ▷ Maximum de vérifications statiques.
- Généricité
  - ▷ Donner les outils pour écrire des algorithmes génériques.

Types implémentés

Note sur les comportements : *behavior*

▷ Types souvent paramétrés par un comportement.

▷ Spécifie les mesures à prendre en cas de débordement.

▷ Plusieurs comportements sont implémentés :

- *strict* : arrête le programme.
- *unsafe* : ne fait rien.
- *saturate* : sature le type par son minimum ou par son maximum.

Scalaires

- Entier non signé : `int_u<nbits,behavior>`
- Entier signé : `int_s<nbits,behavior>`
- Flottant simple précision : `sfloat`
- Flottant double précision : `dfloat`

Raccourcis usuels

- `int_uN` ⇔ `int_u<N,strict>` pour N = 8, 16, 32.
- `int_uNu` ⇔ `int_u<N,unsafe>` pour N = 8, 16, 32.
- `int_uNs` ⇔ `int_u<N,saturate>` pour N = 8, 16, 32.
- De même pour `int_s`.

Vecteurs

- `vec<N,T>`
- Opérateur [] usuel.
- Opérations arithmétiques :
  - ▷ Vecteur - Vecteur : +, -, * (produit scalaire)
  - ▷ Vecteur - Scalaire : +, -, *, /, modulo
- Le constructeur par défaut initialise à 0.

Complexes

▷ Considérés comme des vecteurs de 2 valeurs.

- `cplx<representation,T>`, 2 représentations implémentées :
  - ▷ Représentation polaire : `cplx<polar,T>`
  - ▷ Représentation rectangulaire : `cplx<rect,T>`

Opérations possibles

- Construction par copie inter-représentation.
- Accesseurs : magn(), angle(), real(), imag() pour les deux représentations.

Couleurs

- Type générique de couleur : `color<ncomps,qbits, color_system>`
- Accès aux composantes par l'opérateur [].
- Plusieurs types prédéfinis : rgb, yuv, yiq, ...
- Des conversions sont possibles grâce au module convert.
- Enum pour accéder aux composantes, exemple :

```
rgb_8 my_color (12, 24, 0);
c[rgb_R] = 51;
```

Décorateurs : *range* et *cycle*

$\Rightarrow$ Décorent des types existants.

▷ `range<T,Interval,behavior>`

- Spécifie un intervalle précis de valeurs.
- Exemple :

```
range<int_u8u , bounded_u<0, 51>, strict > r1;
r1 = 52; // aborts the program
```

▷ `cycle<T,Interval,behavior>`

- Permet de faire cycler le type autour de l'intervalle.
- Exemple :

```
cycle<int_u8u , bounded_u<0, 10> > c1 = 12; // c1 == 2
```

Propriété des types

Hiérarchies de traits

2 hiérarchies de traits favorisent l'écriture d'algorithmes génériques :

- `typetraits<T>` : Associe des types au type `T`. Exemple de types définis pour les scalaires :
  - `cumul_type` : Type plus gros permettant le cumul de variables de type T.
  - `signed_type` : Renvoie le type signé le plus proche.

- `optraits<T>` : Implémente certaines méthodes associées au type T, min(), max(), ou zero() par exemple.

Interactions avec les builtins

- Chaque type peut se convertir vers son type builtin le plus proche.

  $\Rightarrow$ Réutilisation possible des algorithmes non génériques existants.

- `optraits` et `typetraits` sont définis pour les builtins.

  $\Rightarrow$ Bonne intégration dans les algorithmes génériques.

- Les opérations arithmétiques fonctionnent en mixte. Exemple :

```
int_u8 i = 12;
int_u16 j = 5100;
unsigned char c = 51;
i += 1U; // Notice the U. Here a dynamic check is required
         // as 1U is unsigned int.
j = i + c; // Ok, no dynamic check
```

Typage fort

Les types grossissent le plus précisément possible :

- `int_u8+int_u8` $\Rightarrow$ `int_u<9>`
- `int_u16*int_u<2>` $\Rightarrow$ `int_u<18>`

$\Rightarrow$ Un maximum de vérification de débordements peuvent être statiques !

Kit de survie

Recherche de bugs

2 fonctions sont très utiles :

- `typename_of<T>` : renvoie une `string` avec le nom du type T.
- `typename_of_var(val)` : renvoie une `string` avec le nom du type val.

▷ Penser à afficher les informations disponibles sur chaque type grâce à `optraits`.

▷ Éventuellement faire afficher des valeurs directement dans les headers d'Olena.

Exemple de somme générique en mode verbose

```
template <unsigned N, class T>
typename typetraits<T>::cumul_type cumul (const vec<N, T>& lhs)
{
  std::cout << "lhs: " << typename_of_var(lhs) << std::endl;
  std::cout << "T: " << typename_of<T>() << std::endl;

  std::cout << "T::zero: " << optraits<T>::zero() << std::endl;
  std::cout << "T::min: " << optraits<T>::min() << std::endl;

  typename typetraits<T>::cumul_type result = optraits<T>::zero();
  for (unsigned i = 0; i < N; ++i)
    result += lhs[i];
  return result;
}
```

Mode unsafe

La macro `NDEBUG` désactive les checks dynamiques. Ceci est utile quand :

- Le programme est trop lent, et que la sécurité n'est pas une priorité.
- Le programme est considéré "sûr" et n'a plus besoin de vérifications.

Cependant :

- La différence de rapidité dépend beaucoup du contexte.
- Généralement, elle n'est pas énorme.

Questions ?