第14章「メモリを（ほとんど）自動的に管理する」

主なトピック

- ポインタクラス
- スマートポインタクラス

ポインタのようなハンドルクラス

- Handleはオブジェクトを指す
- Handleオブジェクトをコピーできる
- Handleオブジェクトが有効な内部データを指しているかテストできる
- Handleオブジェクトが派生クラスのオブジェクトを指す場合に多態性が表れる（正しい関数が自動的に選択され実行される）

Handleクラスの使い方

- vector< Handle<Core> > students;
- Handle<Core> record;
  - 前回のインプリメントでは vector<Core*> とするか，ハンドルクラスを利用した vector<Student_info>

Handle クラス

```
template <class T> class Handle {
public:
    Handle( ) : p(0) {};
    Handle( const Handle& s ) : p(0) { if(s.p) p = clone(s.p); }
    Handle& operator=( const Handle& );
    ~Handle( ) { delete p; }
    Handle( T* t ) : p(t) {}
    operator bool() const { return p; }
    T& operator*() const;
    T* operator->() const;
private:
    T* p;
};
```

Handle クラス

```
template <class T>
Handle<T>& Handle<T>::operator=(const Handle<T>& rhs)
{
  if(&rhs != this ){
    delete p;
    p = rhs.p? clone( rhs.p ): 0;
  }
  return *this;
}
```

Handle クラスの外部関数

```
template <class T>
T* clone(T* p)
{
  return new T(*p);
}
```

- すべての型でclone()関数が使えるようにテンプレート関数
- T型が T(T)にマッチするコンストラクタを持つこと
  →T(const T&) コピーコンストラクタの形式のコンストラクタを持つこと

型変換の演算子

- operator bool() const { return p; }
  - pをbool型に変換，つまり有効(pが0以外)のときtrue，無効(pが0)のときfalseを返す
  - 型変換の演算子は，戻り値の型がない

型変換の演算子

- pの参照を返す演算子 *

```
template <class T>
T& Handle<T>::operator*() const  {
    if(p)  return *p;
    throw std::runtime_error("unbound Handle");
}
```

- Handle<Core> record;
  - *recordで，recordが保持しているポインタの参照を返す
- (*record).valid() のような使い方

型変換の演算子

- p（ポインタ）を返す演算子 ->

```
template <class T>
T* Handle<T>::operator->() const  {
    if(p)  return p;
    throw std::runtime_error("unbound Handle");
}
```

- record -> valid() のような使い方

Handleクラスは，ポインタの管理が可能

さらにもう一歩進めて

- Handle型はデータを持つオブジェクトのコピーと代入の際は、必要のない時でもコピーする
- 一方で、データを共有するなどコピーが必要ない時もある

- データを共有できるかをどうかを決められるハンドル: Ptr型を作成する

Ptr型

- 参照カウンタという変数を導入する
  - そのオブジェクトが他のどのオブジェクトと結びついているのかを保持
  - 他のオブジェクトにコピーするときに、データを複製せずに参照カウンタを1増やす
  - 必要のなくなったデータを削除する
- 必要なときには、他のオブジェクトのデータを複製して自分に持たせる

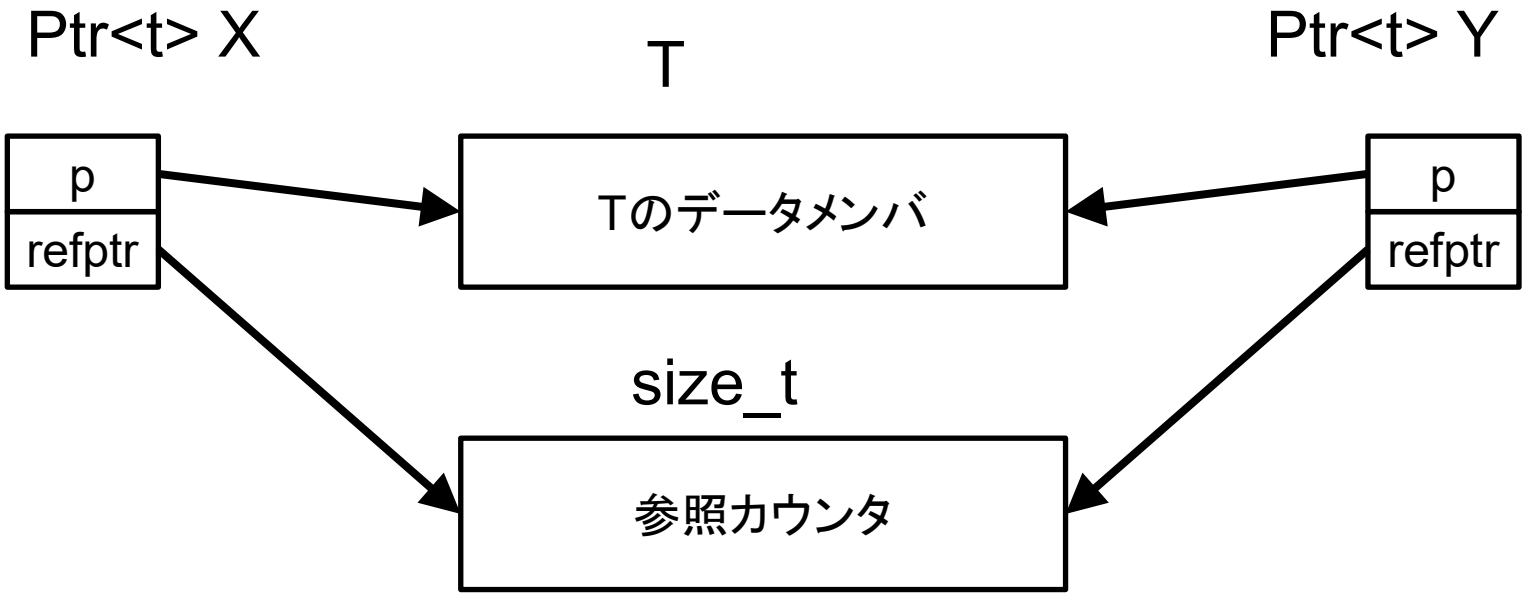
```
template<class T>
class Ptr  {
  private:
    T* p; //  データ保持するデータ
    size_t* refptr; //  参照カウンタ

  public:
    //  デフォルトコンストラクタ
    Ptr(): refptr( new size_t(1) ), p(0) {};
    Ptr(T* p): refptr(new size_t(1), p(t))  {};
    //  コピーコンストラクタ
    Ptr(const Ptr& h) : refptr(h.refptr), p(h.p) {++*refptr};
```

```
    Ptr& operator=(const Ptr&);
    ~Ptr();
    operator bool() const {return p};
    T& operator*() const  {
      if(p) return *p;
      throw std::runtime_error("unbound Ptr handle");
    };
    T* operator->() const  {
      if(p) return p;
      throw std::runtime_error("unbound Ptr handle");
    };
    void make_unique(); //  複製に利用
}
```

データの複製

Prt<T> x, y; x = y; としたとすると



データの複製

- 前のスライドのように同じオブジェクト指しているときに、同じ内容の別オブジェクト作成するときに使う
- 元のオブジェクトの参照数を減らして、自分には同じデータを複製し、参照数を1とする

```
void Ptr::make_unique()  {
  if(*refptr != 1)  {
    --*refptr;
    refptr = new size_t(1);
    p = p? clone(p): 0;
  }
}
```

clone()関数

```
//  clone()というメンバー関数があるクラスには
template<class T> T*clone(T* tp)  {
  return tp->clone();
}

//  clone()というメンバー関数がないクラスには個別に定義する
Vec<char>*clone(const Vec<char>* vp)  {
  return new Vec<char>(*vp);
}
```

その他の関数

```
//  デストラクタ、データが必要がなくなったら削除
template <class T> Ptr<t>::~Ptr()  {
  if(--*refptr == 0)  {
    delete refptr;
    delete p;
  }
}
```

その他の関数

```
//  代入、データの複製はしない、必要がなくなったデータが発生したら削除
template <class T>
Ptr<t>& Ptr<t>::operator=(const Ptr& rhs)  {
  ++*rhs.refptr;
  //  データがどこから参照されず、必要なくなったとき
  if(--*refptr == 0)  {
    delete refptr;
    delete p;
  }
  //  右辺から値コピー
  refptr = rhs.refptr;
  p = rhs.p;
  return *this
}
```