第4回C++講座

目次

・new演算子、delete演算子

・リスト構造

・new[]演算子、delete[]演算子

・Vectorクラスの作成

・vectorとstringの紹介

・C++の他の機能の紹介

ポイント

・newで確保した領域は、deleteするまで生存する
・new[]演算子は、指定した個数の領域を確保する
・newならdelete、new[]ならdelete[]を使用する

・2度deleteしてはいけない
・NULLのdeleteは安全

クラスとは？

クラス・構造体の定義 ⇒ 新しい”型”を作ること！

(型：プログラム上で使用される変数の種類(ex:　int, double)　)

プログラム上での振る舞いを定義する必要がある

そのクラスの大きさはいくらか？
そのクラスのメモリ構成はどうなっているか？ ⇒ **メンバ変数**

メンバ変数をどのように操作するか？ ⇒ **メンバ関数**

メンバにどこからアクセスできるか？ ⇒ **アクセス修飾子**

初期化時の処理
破棄時の処理はどのようにするか？ ⇒ **コンストラクタ・デストラクタ**
コピーコンストラクタ

代入処理・不等式等はどのように行なうか？ ⇒ 演算子オーバーロード

共通した性質を抜き出す ⇒ 継承

自分で定義した通りに作用する変数を作ることができる！

new演算子, delete演算子

変数の生存期間

スコープの終わりまで

⇒ 任意に生存期間を定めたい！

new、delete演算子

newで確保すると、deleteするまで
生存する領域を作成！

＊new、deleteを使うことで…

・必要なときに必要なメモリを確保→テキストデータなど
・敵オブジェクト、マップチップを必要なだけ生成、破棄できる！
・関数終了後も生存する変数→変数生成関数

new演算子の特徴

(1) new 型名(コンストラクタ引数);

(2)メモリ上に、指定した型の分の領域を確保

(3) 指定した型がクラスなら、その後に引数に
合うコンストラクタを実行

(4)最後に、確保した領域へのポインタを返す

(5) delete されるまでメモリ上には確保されたまま
(必ずdeleteすること)

delete演算子の特徴

(1) delete pointer; ←確保した領域へのポインタ

(2) 指定されたポインタの指し示す先を解放

(3) 解放するのがクラスならデストラクタを実行

(4) 複数回deleteしてはいけない

(5) NULL(= 0)は安全に処理される

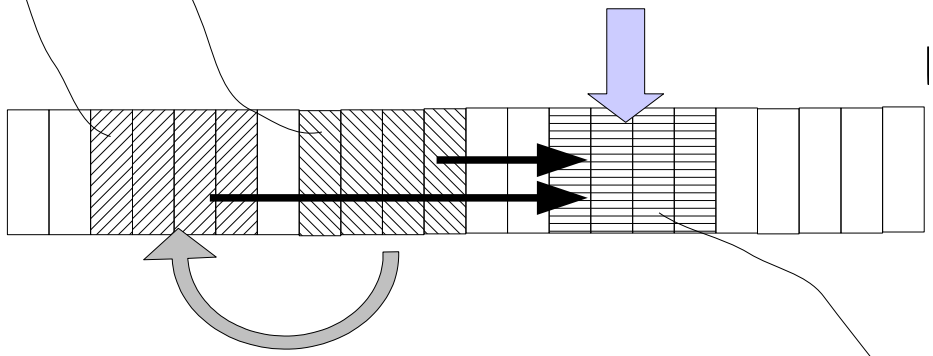
new演算子, delete演算子の動作確認

```
int main()
{
    int* p;             // ①
    {
        int* pp = new int(); // ②
        p = pp;         // ③
    }                   // ④
    *p = 3;             // ⑤
    delete p;           // ⑥
    return 0;           // ⑦
}
```

①pの宣言によってint*型の領域が確保される
⑦スコープの終わりに開放される

②pp の宣言によってint*型の領域が確保される
④スコープの終わりに開放される

⑤pのアドレスを用いて、newの領域に3を代入



③ppの値(アドレス)をpに代入

②newによってint型の領域が確保される
④deleteまで削除されない
⑥deleteによって削除される

単方向リスト構造

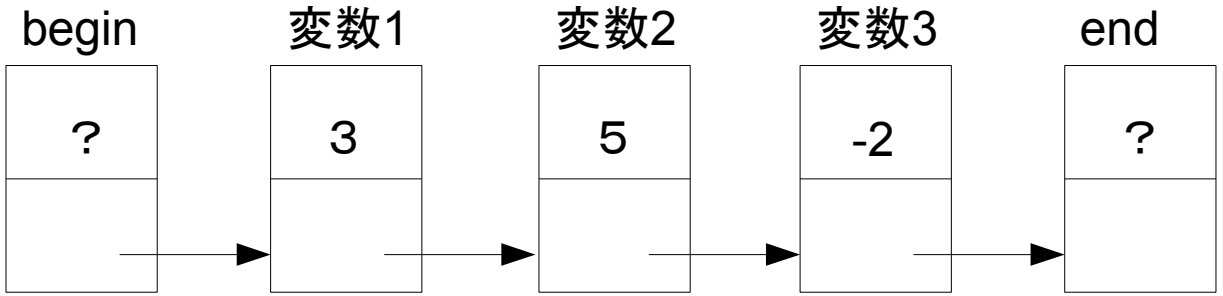
```
class ListContent {
public:
    int value;
    ListContent* next;
};

ListContent begin, end;
```

特徴

・ポインタを使って変数同士を結合して集合を作る
・間に値を挿入するのが楽(双方向リスト)
・要素を無制限に追加できる

・変数の検索には不向き
・単独要素に頻繁にアクセスするのにも不向き

要素数が不定で、先頭から順にアクセスする用途に便利！



＊変数はメモリ上に連続して並んでいない！
＊ポインタの値(矢印)を変える事で、変数の挿入、削除を行なう
＊begin、endは先頭と終端を表すダミーなのでvalueは使わない
＊beginからポインタをたどっていって、endになるまでがListの中身

＊Listにあわせて要素を生存させる→new演算子の使用

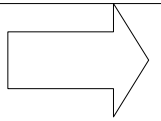
List構造の使用例

```
class ListContent {
public:
    int value;
    ListContent* next;
};

ListContent begin, end;
```

begin、endはListを作るたびに
必要になる→クラスにすれば良い

ListContent* prev; も加えると、
双方向リストになる！

```
int  main() {
    begin.next = &end;
    int val;
    while(true) {
        std::cout << “0で終了 : ”;
        std::cin >> val;
        if(val == 0) break;
        ListContent* temp = begin.next;
        ListContent* new_content = new ListContent();
        new_content->value = val;
        new_content->next = temp;
        begin.next = new_content;
    }

    ListContent* iterator = begin.next;
    while(  iterator != &end) {
        std::cout << iterator->value << “ “;
        ListContent* del = iterator;
        iterator = iterator->next;
        delete del;
    }
    return 0;
}
```



無制限に挿入できるStackに使える！

演習

1．int型の整数を無制限に格納できるスタックを作成せよ

データ構造：単方向リスト
内部で使用するクラス：ListContent

メンバ変数：ListContent begin, end;
コンストラクタ：　Stack();　　→begin、endの結合を行なう
デストラクタ：　 ~Stack();　 →要素のdeleteを行なう
メンバ関数：
　　bool isEmpty();　→スタックが空かどうかを返す。
　　void push(int val);→リストの先頭に値をプッシュする
　　void pop();　　→リストの先頭から値をポップする
　　int top();　　→先頭の値を返却する

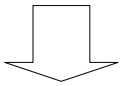
new 演算子、delete 演算子の注意点

```
class PointerHolder {
private:
    int *p;
public :
    PointerHolder(int i) : p(new int(i))
    {}
    ~PointerHolder() {delete p;}
};
int main() {
    PointerHolder p1(3);
    PointerHolder p2(2);
    p1 = p2;
    return 0;
}
```

- メンバ変数のコピーはビットコピー
- p 1の持っていたp を delete していない
- p2 の持っていた p を2回 delete している
- ポインタをメンバに持つクラスの代入は注意が必要

演習

前回作成したリスト構造を用いたStackクラスについて、どのような問題が生じうるか考察せよ。また、この問題を解決するにはクラスのどのような機能を使用する必要があるか



＊ 問題は起こりうるが、それを理解していれば間違いではない
＊ 汎用的なクラス作成は難しい→STLを使う！

コピーコンストラクタ、代入演算子

```
class PointerHolder {
private:
    int *p;
public :
    PointerHolder(int i) : p(new int(i)) {}
    PointerHolder(const PointerHolder& init)
    : p(new int(*(init.p))
    {}
    const PointerHolder& operator=(const
     PointerHolder& rhs) {
        *p = *(rhs.p);
        return *this;
    }
    ~PointerHolder() {delete p;}
};
```

コピーコンストラクタ

自分と同じ型の参照を引数にとるコンストラクタ

←では、メンバのビットコピーが行なわれることを防いでいる。

書式は、ClassName(const ClassName&)

代入演算子

どのように代入するかを定める。

←では、ポインタの指している領域の値をコピーしている。

書式は、const ClassName&
operator=(const ClassName& val)

STLのvector、string、stack、queue等は、
安全に代入処理ができるように設計されている

new[]演算子, delete[]演算子

一般の配列

ソースコード内で配列の数を指定

⇨ 実行中に配列の数を決めたい！

new[]、delete[]演算子

new[]で確保すると、delete[]するまで生存する領域を作成

指定した数(変数可！)の領域を確保

＊new[]、delete[]を使うことで…

・必要なときに必要なメモリを確保→テキストデータなど
・可変長配列クラスの作成→vector
・可変長文字列クラスの作成→string

new演算子の特徴

(1) new 型名[変数の数];

(2)メモリ上に、指定した個数の分の
連続した領域を確保

(3) 指定した型がクラスなら、その後にデフォルト
コンストラクタを実行

(4)連続した領域の先頭へのポインタを返す

(5) delete[] されるまでメモリ上には確保された
まま(必ずdelete[]すること)

delete演算子の特徴

(1) delete[] pointer; ←確保した領域へのポインタ

(2) new → delete, new[] → delete[]
必ず使い分ける！

(3) 解放するのがクラスならデストラクタを実行

(4) 複数回delete[]してはいけない

(5) NULL(= 0)は安全に処理される

new[] 演算子と delete[]演算子の動作確認

```
#include <iostream>

int main() {

    int num;

    std::cin >> num;

    int *ar = new int[num];

    for(int i = 0; i < num; ++i) {

        ar[i] = i;

        std::cout << ar[i] << “\n”;

    }

    delete[] ar;

    return 0;

}
```

ポイント→プログラム実行中に配列の大きさを決めている

演習

可変配列Vectorクラスを作成する(コピー・代入に問題あり)

メンバ:　配列の数 int d_arr_num;
　　可変配列(ポインタ) int* d_arr;
コンストラクタ:　Vector()→要素数0の配列を作成
　　Vector(int n)→要素数nの配列を作成
デストラクタ　:　~Vector()→可変配列をdelete[]する
関数:
void resize(int size); で配列の大きさを変える
void set(int index, int val); でindex番目にval を代入。
　　足りなければ自動で拡張
int get(int index); で、index番目の値を返却
int getSize(); で、配列のサイズを返却
int* getPointer(); で、可変配列のポインタを返却

ステップアップ:演算子オーバーロード
int& operator[](int& index); でindex番目の要素の参照を返却
Vector(const Vector& init)　→コピーコンストラクタの定義
const Vector& operator=(const Vector& temp) →代入演算子

その他の重要な機能

C言語の積み残し

static, const, 修飾子
static →　プログラムを通して一つだけ、という宣言
const →　この変数は値が変わらない、という宣言(ポインタで注意)

voidポインタ→すべての型のポインタを保持可能→フォルダ構造や、引数の多様性

C++の重要な機能・その他

静的メンバ変数、静的メンバ関数→クラスに一つだけの変数、クラス固有の関数

テンプレート→型の機能に依存しないクラス(コンテナなど)で使用。
複数の型に対応するクラス/関数を自動で生成してくれる

継承　→差分プログラミング、インターフェース、関数オーバーライド

継承とテンプレートを用いたファンクタ　→　関数とクラスの同一化

演算子オーバーロード→+,-,=,(),==などの演算子でどのように振舞うか

STL→コンテナ、アルゴリズム、イテレータ、ストリーム、継承を用いた拡張

静的メンバ変数、静的メンバ関数

- 静的メンバ変数

(特徴）普通のメンバ変数と異なり、”各クラス”に一つだけ存在する変数

(用途) そのクラス固有の値を保存する場合に使う。

(使用例) あるクラスが何個の変数を生成したかカウントしたい場合、静的メンバ変数にして、コンストラクタやコピーコンストラクタでインクリメントすればよい。

(補足) メンバ関数、静的メンバ関数から使用可能

(アクセス) public なら、(クラス名)::（変数名) でアクセス可能

- 静的メンバ変数

(特徴) 普通のメンバ関数と異なり、静的メンバのみを扱える関数

(用途) メンバ変数の値によらない（変数の値によらない）処理をする場合に用いる

(使用例) シングルトンクラスを作成する場合に使われる

(アクセス) public なら、(クラス名)::（関数名）でアクセス可能

継承

特徴：親クラスの機能を引き継ぐ

差分プログラミング

完成したクラスに機能を追加したい場合

→完成したクラスは触りたくない
→完成したクラスはすでに使用しており、今回は少し機能を追加したクラスを別の場所で使いたい

SuperClass を親クラスとして設定した SubClass を作成→SuperClassの全機能を使用可能！

インターフェース

仮想関数を定義した親クラスを継承する使い方。機能の保証。

1. 親クラスDrawObjectで、virtual void draw() = 0; と仮想関数を宣言する。
2. SuperClass を継承した子クラスで、void draw() {} を定義する。
3. たとえば、Static1クラスは、メンバ変数として保持している画像を指定点に描画し、
   Enemy1クラスは、自分のいる位置に複数画像を描画し、
   StringDrawクラスは、指定文字列を指定フォントで指定位置に描画する。
4. 子クラスは、親クラスのポインタとして保持できる。
   DrawObject* p = new StringDraw(“aaa”, “mintyo”, 3, 3); p->draw(); とすると、
   自動でStringDrawクラスと判別して文字列を描画してくれる。
5. DrawObject*をvecotrなどで保持しておけば、ループでまわすだけで別描画可能！

⇨ **必要な変数の違いをメンバ変数として吸収！**

フォルダのような構造

特徴：継承を用いた同一化

基本的な考え方

1．親クラスとなるContentクラスを作成する。
2．Contentクラスを継承するFolderクラスとFileクラスを作成する。
3．FolderクラスはContentクラスのポインタを保持する＝子クラスのFolderとFileのどちらも保持できる！

⇒ **このように階層構造を作ることができる**

応用

動的な移動ルーチンの作成
1．親クラスMoveClassを定義する。
　経過フレームを引数にとり、x,yに値を加減算する関数void move(int frame, int& x, int& y);
　そのMoveClassルーチンの総フレームを返す　int totalFrame();
　を仮想関数に持つ。(必要ならループ回数フラグも)

2．基本的な移動クラスStraight,Tracking,Sine等を作成し、MoveClassを継承する。
3．移動クラスの保持クラスMoveHolderもMoveClassを継承する。
　経過フレームを受け取ったら、保持しているMoveClassを走査して、適応するMoveClassの
　move関数を実行させる。走査には各オブジェクトのtotalFrame関数を使用する。
4．MoveHolderはMoveHolderもStraight,Tracking,Sineも保持可能で柔軟なルーチン作成可能

＊これらの応用は、すべてC言語のvoidポインタで代用可能である。
型チェックの厳密性等、堅牢なプログラミングができる、という点がメリットである。