FLUJOS

Los flujos se plantean como una soluciÃ³n a la falta de recursos de I/O de C. Los lenguajes en general tienen implementados recursos de I/O para manejar unos pocos tipos de datos. Con los flujos C++ provee un recurso para manejar I/O para tipos definidos por el usuario de manera fÃ¡cil, flexible y eficiente

Flujos estÃ¡ndares :

archivo de cabecera : iostream.h

cin : entrada estÃ¡ndar

cout : salida estÃ¡ndar

cerr : salida de error estÃ¡ndar

Operadores :

<< : “poner en”

>> : “extraer de”

En C : *printf(“un entero %d, un real %f, entero, real);*

scanf(“%d%f%c”, &entero, &real, &carÃ¡cter);

En C++ :

cout<< “Un entero”<<entero<< “, un real” <<real;

cin >> entero>> real >> carÃ¡cter;

No es necesario especificar el formato, la sobrecarga de operadores permite leer en cualquier formato. AdemÃ¡s, en la entrada de datos no es necesario darle el operador “direcciÃ³n de” *&*.

La entrada y salida de datos con los operadores >> y << se harÃ¡ siempre y cuando estos operadores estÃ©n definidos para el tipo especificado. El programador tiene la capacidad de definir los operadores para un nuevo tipo.

Los operadores << y >> se eligieron por tener baja prioridad para permitir, por ejemplo, incluir expresiones aritmÃ©ticas sin utilizar parÃ©ntesis

*cout<< “a*b+c2” << a*b+c2 << `\n';*

AdemÃ¡s de utilizar la sobrecarga de operadores los flujos tienen definida una jerarquÃa de clase sobre la cual estÃ¡n soportados :

IOS (superclase)

(estados del flujo, formato, manipuladores,…)

ISTREAM OSTREAM

(sobrecarga >>) (sobrecarga <<)

IFSTREAM OFSTREAM

(Para flujos en archivos)

La clase *ostream* por dentro :

```
class ostream: public virtual ios {
//…
public :
ostream& operator<<(const char*); //cadenas
ostream& operator<<(char);
ostream& operator<<(short i)
{ return *this<< int(i); }
ostream& operator<< (int);
…
…
};
```

la operaciÃ³n : *cout << "x =" << x; // x es entero*

se interpreta como :

(cout.operator<<("x =")).operator<<(x);

esto implica que lo que se imprime se hace de izquierda a derecha

Sobrecarga de operador para clase complejo :

```
ostream& operator<<(ostream& s, complejo z)
{
return s<< "(" << real(z) <<","<< imag(z) << ")";
}
main()
```

{

complejo x(1,2);

cout << “x =” << x,'\n';

} // Salida *x =(1,2)*

Notar que para definir una salida para un tipo definido por el usuario no es necesario modificar la clase *ostream* ni acceder a la estructura de datos mantenida por la clase

Entradas

La clase *istream* es similar a la *ostream* :

class istream: public virtual ios {

//…

public :

istream& operator>>(char); //cadenas*

istream& operator>>(char &);

istream& operator>> (short &);

istream& operator>> (int &);

…

};

Una funciÃ³n se define de la siguiente manera :

istream& istream::operator>>(T& varT)

{

// pasar por alto los espacios en blanco

// leer de algÃºn modo T (tipo) y colocarlo en vart

*return *this;}*

Como se puede observar los argumentos para las entradas son en general punteros

Funciones Ãºtiles para manejar entradas que se encuentran en <*ctype.h*> :

int isalpha(char) // `a'..'z' `A'..'Z'

int isupper(char) // `A'..'Z'

int islower(char) // `a'..'z'

int isdigit(char) // `0'..'9'

int isxdigit(char) // `0'..'9' `a'..'f' `A'..'F'

int isspace(char) // ` ` `\t' <CR> <LF> nueva pÃ¡gina

int iscntrl(char) //carÃ¡cter de control ASCII 0..31-127

int ispunct(char) // signos puntuaciÃ³n, ninguno anterior

int isascii(char c) { return 0<=c && c<=127}

Estados de flujos

Todo flujo *(istream* u *ostream)* tiene un estado asociado. Los estados se pueden examinar con operaciones sobre la clase *ios* :

class ios {

…

public :

int eof() const; // se detecta el fin de archivo;

int fail() const; // fracasarÃ¡ la siguiente operaciÃ³n

int bad() const; // el flujo estÃ¡ corrompido

int good() const; // la siguiente operaciÃ³n podrÃ–a tener

// Ã©xito

};

Los valores de los estados tambiÃ©n estÃ¡n definidos en la clase *ios :*

class ios {

// …

public :

enum io-state {

goodbit=0; eofbit=1, failbit=2; badbit=4;};

//…

};

switch (cin.rstate()) {

case ios::goodbit :

// la Ãºltima operaciÃ³n con cin tuvo Ã©xito

break;

case ios::eofbit :

// final de archivo

break;

case ios::failbit :

// faquÃ– se tuvo un error

break;

case ios::badbit :

// quizÃ¡s se perdieron caracteres de cin

break;

};

while (cin>>z) cout<<z << `\n';

En el caso del *while* siempre se comprueba si la operaciÃ³n tiene Ã©xito *(estado good)*

Entradas para tipos definidos por el usuario

Se pueden definir entradas para tipos definidos por el usuario, pero en este caso al entrada se debe hacer por referencia :

istream& operator<<(istream& s, complejo &a)

{

double re=0, im=0;

char c=0;

s>>c;

if(c== `(`) {

s>>re>>c;

if (c ==',') s >> im >>c;

if(c != `)') s.clear(ios::badbit); // fijar estado

else {

s.putback(c)

s >> re;

}

if (s) a = complejo(re,im);

return s;

}

Formatos

ios es quien controla las operaciones de entrada salida y quien tiene el buffer para estas operaciones, tambiÃ©n es quien controla el formato de inserciÃ³n y extracciÃ³n de datos.

class ios {

//…

public:

…

int width(int w); // fijar la anchura del campo

int width() const;

char fill(char); // fijar carÃ¡cter para relleno

char fill() const; // devolver carÃ¡cter para relleno

…

int precision(int); //fijar precision numeros flotantes

int precision() const;

int rdstate() const; // estados de flujo

…

}

La funciÃ³n *width()* especifica el nÃºmero mÃ–nimo de caracteres que se emplearÃ¡ para la siguiente operaciÃ³n de salida numÃ©rica o de cadena.

cout.width(4);

cout << `(` << 12 << `)'; // Salida (12)

width() especifica el nÃºmero mÃnimo de caracteres si se especifican mÃ¡s se imprimirÃ¡n todos:

cout.width(4);

cout << `(` << 121212 << “)\n”;// Salida (121212)

La filosofÃa de C++ es que es preferible permitir imprimir con un formato desbordado, desprolijo pero que se entienda.

Si queremos especificar un carÃ¡cter de relleno :

cout width(4);

cout fill(`#');

cout << `(` << “ab” << `)'; // Salida (##ab)

CarÃ¡cter de relleno por omisiÃ³n : espacio

TamaÃ±o de campo por omisiÃ³n : 0 tantos caracteres como sean necesarios. Para restablecer el tamaÃ±o de campo por omisiÃ³n :

cout.width(0); // tantos caracteres como se necesiten

Una llamada de *width()* solo afecta la salida siguiente :

cout.width(4);

cout.fill(`#');

cout << `(` << 12 << “) , (“ << 12 << “) \n”;

Salida : *(##12), (12)* no *(##12)(##12)*

Estado de formato

ios tiene un estado de formato especificado por las funciones *flags* y *setf(),* que sirven para prender y apagar las banderas de especificaciÃ³n de formato

class ios {

public:

// banderas para controlar el formato :

enum {

skipws=01,// pasar por alto el espacio en blanco en

// las entradas

//AJUSTE DE CAMPO :

left=02, // relleno despues del valor

right=04, // relleno antes del valor

internal= 010, // relleno entre el signo y el valor

//BASE ENTERA :

dec=020, oct=040, hex=0100,

showbase=0200, // mostrar base entera

showpoint=0400, // imprimir ceros a la derecha

uppercase=01000, // `E', `X' en lugar de `e', `x'

showpos=02000, // `+' explÃ−cito para enteros positivos

//ESPECIFICACION PARA PUNTOS FLOTANTES:

scientific=04000, // .dddddd Edd

fixed=010000 // dddd.dd

//

Para especificar un conjunto de opciones :

const int mis_opciones_de_io = ios::left/ios::oct/ios::showpoint/ios::fixed/;

se instalan con una sola opciÃ³n :

cout.flags(mis_operaciones_de_io);

Una vez encendida una bandera se conserva hasta que se apaga nuevamente.

Cuando la opciÃ³n a especificar tiene mÃ¡s de un bit o un conjunto de bits es que *setf* reciba un segundo seudoargumento indicando la opciÃ³n ademÃ¡s del valor :

cout.setf(ios::dec, ios::basefield);//decimal

cout << 1234 << ` `; // por omisiÃ³n decimal

cout.setf(ios::oct, ios::basefield); //octal

cout << 1234 << ` `;

cout.setf(ios::hex, ios::basefield); //hexadecimal

cout << 1234 << ` `;

1234 2322 4d2

Para ajustar el campo de caracteres tenemos las siguientes opciones :

cout.setf(ios::right, ios::adjustfield);// justificaciÃ³n derecha

cout.width(4);

cout << `(` << -12 << ") \n";

cout.width(4);

cout.setf(ios::left, ios::adjustfield); // a la izquierda

cout << `(` << -12 << ") \n";

cout.width(4);

cout.setf(ios::internal, ios::adjustfield); // interna

cout << `(` << -12 << ") \n";

Salida :

(-12)

(-12)

(- 12)

Salidas punto flotante

cout << 1234.56789 << `\n';

cout.setf(ios::scientific, ios::floatfield); // notaciÃ³n cientÃ–fica

cout << 1234.56789 << `\n';

cout.setf(ios::fixed, ios::floatfield);

cout << 1234.56789 << `\n';

Salida :

1234.57

1.2345678e+03

1234.567890

Si se quiere especificar la cantidad de dÃ−gitos :

cout.precision(n); // valor por omisiÃ³n es n=6

La llamada a precisiÃ³n afecta a todas las salidas de punto flotante hasta la siguiente llamada a precisiÃ³n :

cout.precision(8);

cout << 1234.56789 << `\n';

cout.precision(4);

cout << 1234.56789 << `\n';

Salida :

1234.5679

1235

Los valores se redondean en lugar de truncarse

Para evitar el empleo de prendido y apagado de banderas existen manipuladores estÃ¡ndares para istream y ostream que pueden ser invocados directamente en las entradas *(cin)* y salidas *(cout)* estÃ¡ndares

Estos manipuladores son :

// manipuladores simples. Se debe incluir <iomanip.h>

ios& oct(ios&);

ios& dec(ios&);

ios& hex(ios&);

ostream& endl(ostream&); // agregar `\n' y vaciar

ostream& ends(ostream&); // agregar `\0' y vaciar

ostream& flush(ostream&); // vaciar el flujo

istream& ws(istream&); // eliminar espacioblanco

// Manipuladores que reciben argumentos :

SMANIP<int> setbase(int b); //fijar base

SMANIP<int> setfill(int f); //fijar relleno

SMANIP<int> setprecision(int p); //fijar precisiÃ³n

SMANIP<int> setw(int w); //fijar anchura

SMANIP<long> resetiosflags(long b); //restablece banderas

SMANIP<long> setiosflags(long b); //establece banderas

Se emplean de la siguiente manera :

cout << 1234 << ` ` << hex << 1234 << oct << 1234 << endl;

cout << setw(4) << setfill(`#') << `(` << 12<< “)\n”;

Salida : 1234 4d2 2322

(##12)

Un programador puede crear sus propios manipuladores

Flujos en archivos

Archivo de cabecera : *fstream.h (que incluye a iostream.h)*

Como ya vimos *ifstream* y *ofstream* se derivan de *istream* y *ostream* y heredan las operaciones de extracciÃ³n e inserciÃ³n respectivamente.

#include <fstream.h>

main()

{

char ch;

ifstream entrada (“a:\\entrada.in”,ios::in);

if(!entrada)

cerr<<“Incapaz de abrir `entrada' para entrada”;

ofstream salida (“a:\\salida.out”,ios::out);

if(!salida)

cerr<<“Incapaz de abrir `salida' para salida”;

while(salida && entrada.get(ch)) //mientras not EOF

// sigue extrayendo caracteres y los pone en salida

salida.put(ch);

entrada.close();

salida.close()

```
return(0);
}
```

A veces es mejor pasar el archivo de entrada y de salida como argumentos de entrada al programa :

```
#include <fstream.h>
main(int argc, char* argv[])
{
if(argc !=3) cerr<<"nÃºmero de argumentos erroneo"
ifstream entrada (argv[1]);
if(!entrada)
cerr<<"Incapaz de abrir `entrada' para entrada";
ofstream salida (argv[2]);
if(!salida)
cerr<<"Incapaz de abrir `salida' para salida";
while(salida && entrada.get(ch)) //mientras not EOF
// sigue extrayendo caracteres y los pone en salida
salida.put(ch);
entrada.close();
salida.close()
return(0);
}
```

Otro ejemplo :

```
ifstream mi_arch;
…
mi_arch.open("entrada.dat");
…
mi_arch.close();
```

mi_arch.open(“parametros.dat”);

…

mi_arch.close();

Modos de operaciÃ³n de los flujos

Para modificar el modo en que se abre y/o se utiliza un archivo se emplea el segundo argumento en la construcciÃ³n del flujo de un archivo:

Modo de Bit	AcciÃ³n
ios::in	abre para lectura
ios::out	abre para escritura
ios::ate	se posiciona en EOF despuÃ©s de crear el archivo
ios::app	todas las escrituras van al final archivo
ios::trunc	si el archivo existe lo corta
ios::nocreate	si el archivo no existe no lo abre
ios::noreplace	si el archivo existe no lo abre
ios::binary	abre archivo en modo binario

Un archivo se puede abrir tanto para entrada como para salida :

fstream diccionario(“concordancia”, ios::in|ios::out);

Posicionamiento directo

Si visitamos la clase *istream* y *ostream* tenemos algunas funciones para posicionar los flujos directamente :

class ostream : public virtual ios {

…

public:

ostream& flush(); //envÃ–a contenido buffer a la salida

…

ostream& seekp(streampos);

ostream& seekp(streamoff, seek_dir);

streampos tellp();

…

};

class istream : public virtual ios {

…

public:

int peek();

istream& putback(char c);

…

istream& seekg(streampos);

istream& seekg(streamoff, seek_dir);

streampos tellg();

…

};

seekp y seekg se sitÃºan en una direcciÃ³n absoluta dentro del archivo el sufijo p indica poner *(put)* y el g extraer *(get)*

streampos representa una posiciÃ³n de carÃ¡cter dentro del archivo

streamoff representa un desplazamiento a partir de un punto indicado en *seek_dir*

Valores de seek_dir :

ios::beg desde el principio

ios::cur desde posiciÃ³n actual

ios::end desde el final del archivo

peek : permite determinar cual es el siguiente carÃ¡cter a leer sin afectar el resultado de la lectura siguiente

putback : permite colocar un carÃ¡cter no deseado de vuelta en el flujo para leerlo en otra ocasiÃ³n

Ejemplos :

fstream io(“revision.dat”, ios::in\ios::app);

streampos pos_actual = io.tellp(); // incializa la posiciÃ³n

// actual

io<<object1<<object2<<object3; // se escriben tres objetos

io.seekp(pos_actual); // se reposiciona el puntero en

// pos_actual (principio archivo)

io.seekp(sizeof(OBJECT), ios::cur); // se pasa por alto

// object1

io <<obj2_nuevo; //se graba sobre object2 obj2_nuevo

io.seekg(5,ios::cur); // se posiciona el puntero del archivo

// (get_file) cinco bytes desde la posiciÃ³n actual

io.seekg(-7,ios::end); // siete bytes atrÃ¡s desde la pos. actual

Universidad TecnolÃ³gica Nacional - Santa Fe - Departamento Sistemas -

Curso : Desarrollos de ProgramaciÃ³n en C++