Punteros a miembros, operadores y alocación dinámica

Instituto Balseiro

Puntero a datos miembro

```
class Data
{
    public:
        int a, b, c;
        void print() const {
            cout << "a = " << a << ", b = " << b << ", c = "
                 << c << endl;
        }
};

int main() {
    Data d, *dp = &d;

    int Data::*pmInt = &Data::a;
    dp->*pmInt = 47;

    pmInt = &Data::b;
    d.*pmInt = 48;

    pmInt = &Data::c;
    dp->*pmInt = 49;
    dp->print();
}
```

```
a = 47, b = 48, c = 49
```

Puntero a funciones miembro

```
#include<iostream>
using namespace std;

class Simple2 {
    public:
        void f(float v) const {
            cout << v << endl;
        }
};

void (Simple2::*fp)(float) const;
void (Simple2::*fp2)(float) const = &Simple2::f;
//void (Simple2::*fp3)(float) = &Simple2::f;   -> error

int main() {
    Simple2 s;
    fp = &Simple2::f;

    (s.*fp)(3.14);

    return 0;
}
```

```
3.14
```

Puntero a funciones miembro

```
class Widget {
        void f(int i) const { cout << "Widget::f(" << i << ")\n"; }
        void g(int i) const { cout << "Widget::g(" << i << ")\n"; }
        void h(int i) const { cout << "Widget::h(" << i << ")\n"; }
        void i(int i) const { cout << "Widget::i(" << i << ")\n"; }
        enum { cnt = 4 };
        void (Widget::*fptr[cnt])(int) const;
    public:
        Widget() {
            fptr[0] = &Widget::f; // Full spec required
            fptr[1] = &Widget::g;
            fptr[2] = &Widget::h;
            fptr[3] = &Widget::i;
        }
        void select(int i, int j) {
            if(i < 0 || i >= cnt) return;
           (this->*fptr[i])(j);
        }
        int count() { return cnt; }
};
int main() {
    Widget w;
    for(int i = 0; i < w.count(); i++) w.select(i, 47 + i);
}
```

```
Widget::f(47)
Widget::g(48)
Widget::h(49)
Widget::i(50)
```

Puntero a funciones miembro

```
class Widget {
        void f(int i) const { cout << "Widget::f(" << i << ")\n"; }
        void g(int i) const { cout << "Widget::g(" << i << ")\n"; }
        void h(int i) const { cout << "Widget::h(" << i << ")\n"; }
        void i(int i) const { cout << "Widget::i(" << i << ")\n"; }
        enum { cnt = 4 };
        using WP = void (Widget::*)(int) const;
        WP fptr[cnt];
    public:
        Widget() {
            fptr[0] = &Widget::f; // Full spec required
            fptr[1] = &Widget::g;
            fptr[2] = &Widget::h;
            fptr[3] = &Widget::i;
        }
        void select(int i, int j) {
            if(i < 0 || i >= cnt) return;
           (this->*fptr[i])(j);
        }
        int count() { return cnt; }
};
```

Sobrecarga de funciones

- En una variable (objeto) se le asigna un nombre a una región de memoria. Una función es un nombre a una acción.
- En lenguaje natural una palabra puede significar diferentes cosas dependiendo del contexto, es decir está “sobrecargada”.
- En C++ la sobrecarga de funciones permite usar un mismo nombre para diferentes funciones (acciones).
- Otra facilidad que agrega C++ es la de argumentos con valor por defecto.

Decoraciones de Nombres

```
void f();

class X { void f(); };

void X::f() { }
```

- La decoración (name mangling) involucra más que nombres de clases y/o espacios de nombres.

```
void print(char);
void print(float);
```

- La sobrecarga de funciones se realiza sobre tipo y número de argumentos, no sobre tipo de retorno.

Type-safe linkage

```
// Function definition
void f(int)
{
}
```

```
// Function misdeclaration
void f(char);


int main()
{
    f(1); // Causes a linker error
}
```

Sobrecarga de operadores

- La sobrecarga de operadores es “azúcar sintáctico”
- Definir un operador es similar a definir una función, pero el nombre de la función es **`operator@`** donde `@` es el operador que se está sobrecargando.
- El número de argumentos en la lista de argumentos del operador sobrecargado depende de:
  - Si el operador es unario o binario
  - Si el operador está definido como una función global o como una función miembro.
    - Para un operador binario `@`, **`aa@bb`** puede ser interpretado como **`aa.operator@(bb)`** o **`operator@(aa,bb)`**.

Sobrecarga de operadores

```
class X
{
    public:
        void operator+(int);
        X(int);
};

void operator+(X, X);
void operator+(X, double);

void f(X a)
{
    a + 1;    // a.operator+(1)
    1 + a;    // ::operator+(X(1),a)
    a + 1.0; // ::operator+(a,1.0)
}
```

Sobrecarga de operadores

```
class Integer {
        int i;
    public:
        Integer(int ii) : i{ii} { }
        Integer operator+(const Integer &rv) const {
            return Integer(i + rv.i);
        }
        Integer &operator+=(const Integer& rv) {
            i += rv.i;
            return *this;
        }
};

int i = 1, j = 2, k = 3;
k += i + j;

Integer ii(1), jj(2), kk(3);
kk += ii + jj;
```

Operadores unarios (como globales)

```
class Integer
{
        long i;
        Integer *This() { return this; }

    public:
        Integer(long ll = 0) : i(ll) {}
        friend Integer &operator+(const Integer &a);
        friend Integer operator-(const Integer &a);
        friend Integer operator~(const Integer &a);
        friend Integer *operator&(Integer &a);
        friend int operator!(const Integer &a);
        // Prefix:
        friend const Integer &operator++(Integer &a);
        // Postfix:
        friend Integer operator++(Integer &a, int);
        // Prefix:
        friend const Integer &operator--(Integer &a);
        // Postfix:
        friend Integer operator--(Integer &a, int);
};
```

Operadores unarios (como globales)

```
const Integer &operator+(const Integer &a) { return a;              }
Integer operator-(const Integer &a)        { return Integer(-a.i); }
Integer operator~(const Integer &a)        { return Integer(~a.i); }
Integer *operator&(Integer &a)             { return a.This();      }
int operator!(const Integer &a)            { return !a.i;          }

// Prefix: return incremented/decremented value
const Integer& operator--(Integer& a)      { a.i--; return a;      }
const Integer& operator++(Integer& a)      { a.i++; return a;      }

// Postfix; return the value before increment/decrement:
Integer operator++(Integer& a, int)  {
    Integer before(a.i);
    a.i++;
    return before;
}
Integer operator--(Integer& a, int)  {
    Integer before(a.i);
    a.i--;
    return before;
}
```

Operadores unarios (como miembros)

```
class Byte
{
        unsigned char b;
    public:
        Byte(unsigned char bb = 0) : b(bb)
        {}
        const Byte &operator+() const {
            return *this;
        }
        const Byte operator-() const {
            return Byte(-b);
        }
        const Byte operator~() const {
            return Byte(~b);
        }
        Byte operator!() const {
            return Byte(!b);
        }
        Byte *operator&() {
            return this;
        }
        // ...
        // ...
        const Byte &operator++() {
            b++;
            return *this;
        }
        const Byte operator++(int) {
            Byte before(b);
            b++;
            return before;
        }
        const Byte &operator--() {
            --b;
            return *this;
        }
        const Byte operator--(int) {
            Byte before(b);
            --b;
            return before;
        }
};
```

Operadores binarios (como globales)

```
class Integer {
        long i;
    public:
        Integer(long ll = 0) : i(ll) {}

        // Operators that create new, modified value:
        friend Integer operator+(const Integer &left, const Integer &right);
        friend Integer operator-(const Integer &left, const Integer &right);
        friend Integer operator*(const Integer &left, const Integer &right);
        friend Integer operator/(const Integer &left, const Integer &right);
        friend Integer operator%(const Integer &left, const Integer &right);
        friend Integer operator^(const Integer &left, const Integer &right);
        friend Integer operator&(const Integer &left, const Integer &right);
        friend Integer operator|(const Integer &left, const Integer &right);
        friend Integer operator<<(const Integer &left, const Integer &right);
        friend Integer operator>>(const Integer &left, const Integer &right);

        // Assignments modify & return lvalue:
        friend Integer &operator+=(Integer &left, const Integer &right);
        friend Integer &operator-=(Integer &left, const Integer &right);
        friend Integer &operator*=(Integer &left, const Integer &right);
        friend Integer &operator/=(Integer &left, const Integer &right);
        friend Integer &operator%=(Integer &left, const Integer &right);
```

Operadores binarios (como globales)

```
    friend Integer &operator^=(Integer &left, const Integer &right);
    friend Integer &operator&=(Integer &left, const Integer &right);
    friend Integer &operator|=(Integer &left, const Integer &right);
    friend Integer &operator>>=(Integer &left, const Integer &right);
    friend Integer &operator<<=(Integer &left, const Integer &right);

    // Conditional operators return true/false:
    friend int operator==(const Integer &left, const Integer &right);
    friend int operator!=(const Integer &left, const Integer &right);
    friend int operator<(const Integer &left, const Integer &right);
    friend int operator>(const Integer &left, const Integer &right);
    friend int operator<=(const Integer &left, const Integer &right);
    friend int operator>=(const Integer &left, const Integer &right);
    friend int operator&&(const Integer &left, const Integer &right);
    friend int operator||(const Integer &left, const Integer &right);

    // Write the contents to an ostream:
    void print(std::ostream &os) const { os << i; }
}
;
```

Operadores binarios (como globales)

```
Integer operator+(const Integer& left, const Integer& right) {
    return Integer(left.i + right.i);
}
Integer operator-(const Integer& left, const Integer& right) {
    return Integer(left.i - right.i);
}
Integer operator*(const Integer& left, const Integer& right) {
    return Integer(left.i * right.i);
}
Integer operator/(const Integer& left, const Integer& right) {
    require(right.i != 0, "divide by zero");
    return Integer(left.i / right.i);
}
Integer operator%(const Integer& left, const Integer& right) {
    require(right.i != 0, "modulo by zero");
    return Integer(left.i % right.i);
}
const Integer operator^(const Integer& left, const Integer& right) {
    return Integer(left.i ^ right.i);
}
// ...
// ...
```

Operadores binarios (como miembros)

```
class Byte {
        unsigned char b;
    public:
        Byte(unsigned char bb = 0) : b(bb) {}

        const Byte operator+(const Byte &right) const {
            return Byte(b + right.b);
        }
        const Byte operator-(const Byte &right) const {
            return Byte(b - right.b);
        }
        const Byte operator*(const Byte &right) const {
            return Byte(b * right.b);
        }
        const Byte operator/(const Byte &right) const {
            require(right.b != 0, "divide by zero");
            return Byte(b / right.b);
        }
        const Byte operator%(const Byte &right) const {
            require(right.b != 0, "modulo by zero");
            return Byte(b % right.b);
        }
        //...
```

Operador []

```
class Array10 {
        enum  {
            sz = 10
        };
        int a[sz];
    public:
        int &operator[](int i) {
            assert(i >= 0 && i < sz);
            return a[i];
        }
};

int main() {
    Array10 a;
    for (int i = 0; i < 10; i++)
        a[i] = i;

    for (int i = 0; i < 10; i++)
        cout << a[i] << " ";

    cout << endl;
    return 0;
}
```

Operador ->

```
class Obj {
        static int i, j;
    public:
        void f() const  {
            cout << i++ << endl;
        }
        void g() const {
            cout << j++ << endl;
        }
};
int Obj::i = 47;
int Obj::j = 11;

// Container:
class ObjContainer {
        vector<Obj *> a;
    public:
        void add(Obj *obj) {
            a.push_back(obj);
        }
        friend class SmartPointer;
};
```

```
class SmartPointer {
        ObjContainer &oc;
        int index;
    public:
        SmartPointer(ObjContainer &c) : oc(c) {
            index = 0;
        }
        // Return value indicates end of list:
        bool operator++() {          // Prefix
            if (index >= oc.a.size())
                return false;
            if (oc.a[++index] == 0)
                return false;
            return true;
        }
        bool operator++(int) {    // Postfix
            return operator++(); // Use prefix
        }
        Obj *operator->() const {
            assert(oc.a[index] != 0);
            return oc.a[index];
        }
};
```

Operador ->

```
int main()
{
    const int sz = 10;
    Obj o[sz];
    ObjContainer oc;
    for (int i = 0; i < sz; i++)
        oc.add(&o[i]);            // Fill it up
    SmartPointer sp(oc); // Create an iterator
    do
    {
        sp->f(); // Pointer dereference operator call
        sp->g();
    } while (sp++);

} ///:~
```

Operador =

```
class Value {
        int a, b;
        float c;
    public:
        Value(int aa = 0, int bb = 0, float cc = 0.0)
            : a(aa), b(bb), c(cc) {}
        Value &operator=(const Value &rv) {
            if (&rv != this) {
                a = rv.a;
                b = rv.b;
                c = rv.c;
            }
            return *this;
        }
        friend ostream & operator<<(ostream &os, const Value &rv) {
            return os << "a = " << rv.a << ", b = "
                      << rv.b << ", c = " << rv.c;
        }
};
```

Conversión de tipo automático

```
class One {
    public:
        One() {}
};

class Two {
    public:
        Two(const One &) {}
};

void f(Two) {}

int main()
{
    One one;
    f(one); // Wants a Two, has a One
}
```

Operador de conversión

```
class Three {
        int i;
    public:
        Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
        int x;
    public:
        Four(int xx) : x(xx) {}
        operator Three () const { return Three(x); }
};

void g(Three) {}

int main()
{
    Four four(1);
    g(four);
    g(1); // Calls Three(1,0)
}
```

Alocación dinámica de memoria

- Los objetos tiene una vida asociada al scope donde fueron definidos.
- Es conveniente que existan objetos independientemente del scope donde fueron creados.
- El operador **`new`** crea estos objetos en el “free store” o “heap”.
- El operador **`delete`** destruye objetos creados por el operador **`new`**.

```
void f() {
     int *p = new int;
     // ...
     // ...
     delete p;
}
```

Ejemplos de new y delete

```
struct Node {
     int value;
     Node *pNext;
     Node(int v, Node *pN)
          :value{v}, pNext{pN} {}
};

Node *put(Node *h, int v)
{
     return new Node(v,h);
}
Node *get(Node *pHd, int *pVal)
{
     assert(pHd);
     Node *pNHd = pHd->pNext;
     *pVal = pHd->value;
     delete pHd;
     return pNHd;
}
```

```
int main() {
     Node *head{nullptr};

     srand(time(nullptr));

     for(int i = 0; i < 10; ++i)
          head = put(head,
                     rand()%100);

     do {
          int val;
          head = get(head, &val);
          cout << val << endl;
     } while( head != nullptr);

     return 0;
}
```

```
16
12
77
73
95
14
76
40
30
44
```

Ejemplos de new y delete

```
class PStash
{
        int quantity;          // Number of storage spaces
        void **storage;        // Pointer storage
        int next;              // Next empty space

        void inflate(int increase);

    public:
        PStash()
          : quantity(0), storage(0), next(0) {}
        ~PStash();

        int add(void *element);
        void *operator[](int index) const; // Fetch

        // Remove the reference from this PStash:
        void *remove(int index);

        // Number of elements in Stash:
        int count() const { return next; }
};
```

Ejemplos de new y delete

```
static void require( bool cnd, const char *m) {
    if(!cnd) {
        cout << "Error: " << m << endl;
        exit(1);
    }
}
```

```
int PStash::add(void *element)
{
    const int inflateSize = 10;
    if (next >= quantity)
        inflate(inflateSize);
    storage[next++] = element;
    return (next - 1); // Index number
}

// No ownership:
PStash::~PStash()
{
    for (int i = 0; i < next; i++)
        require(storage[i] == 0,
                "PStash not cleaned up");
    delete[] storage;
}
```

```
// Operator overloading replacement for fetch
void *PStash::operator[](int index) const
{
    require(index >= 0,
            "PStash::operator[] index negative");
    if (index >= next)
        return 0; // To indicate the end

    // Produce pointer to desired element:
    return storage[index];
}

void *PStash::remove(int index)
{
    void *v = operator[](index);

    // "Remove" the pointer:
    if (v != 0)
        storage[index] = 0;

    return v;
}
```

Ejemplos de new y delete

```
void PStash::inflate(int increase)
{
    const int psz = sizeof(void *);
    void **st = new void *[quantity + increase];
    memset(st, 0, (quantity + increase) * psz);
    memcpy(st, storage, quantity * psz);
    quantity += increase;
    delete[] storage; // Old storage
    storage = st;     // Point to new memory
} ///:~
```

Ejemplos de new y delete

```
int main()
{
    PStash intStash;

    // 'new' works with built-in types, too.
    // Note the "pseudo-constructor" syntax:
    for (int i = 0; i < 25; i++)
        intStash.add(new int(i));

    for (int j = 0; j < intStash.count();j++)
        cout << "intStash[" << j << "] = "
             << *(int *)intStash[j] << endl;

    // Clean up:
    for (int k = 0; k < intStash.count(); k++)
        delete intStash.remove(k);
}
```

```
int main()
{
    ifstream in ("main.cpp");

    PStash stringStash;

    string line;
    while(getline(in, line))
        stringStash.add(new string(line));

    // Print out the strings:
    for(int u = 0; stringStash[u]; u++)
        cout << "stringStash[" << u << "] = "
            << *(string*)stringStash[u] <<endl;

    // Clean up:
    for(int v = 0; v < stringStash.count(); v++)
        delete (string*)stringStash.remove(v);

}
```

Arrays dinámicos

```
char *save_string(const char *p) {
    char *s = new char[strlen(p) + 1];
    strcpy(s, p);
    return s;
}

int main(int argc, char *argv[]) {
    if( argc < 2 )
        return 1;
    char *p = save_string(argv[1]);
    // ...
    delete [] p;
    return 0;
}
```

Arrays dinámicos

- La memoria alocada con **`new`** debe ser liberada con **`delete`** y la alocada con **`new[]`** debe ser liberada con **`delete[]`**.

```
void f(int n) {
    vector<int> *p = new vector<int>(n);  // one object
    int *q = new int[n];                  // array
    // ...
    delete p;
    delete [] q;
}
```

Memory exhaustion

- Por default, el operador **`new`** dispara una excepción de tipo **`bad_alloc`** si no consigue la memoria solicitada. Por ejemplo:

```
void f(int n) {
    try {
        for(;;)
            new char[10000];
    } catch( bad_alloc ) {
        cerr << "Memory exhausted\n";
    }
}
```

Memory exhaustion

- Se puede instalar una función a ser llamada por **new** cuando se acaba la memoria. Esto se hace a través de la función **set_new_handler**, declarada en **<new>**.

```
void out_of_store() {
    cerr << "memory exhausted\n";
    throw bad_alloc();
}

int main() {
    set_new_handler(out_of_store);
    for(;;) new char[10000];
    cout << "done\n";
}
```

Memory exhaustion

- Existe una variante de operador **new** que no dispara excepciones y devuelve 0 como **malloc**, está declarada en **<new>**.

```
#include <new>
using namespace std;

int main() {
    char *p = new (nothrow) char[1<<30];
    if( p )
        *p = 0;
    delete [] p;
    return 0;
}
```

Operator `new` overload

- El sistema de alocación de memoria utilizada por **`new`** y **`delete`** esta diseñada para un uso de propósito general. Existen situaciones en que no son adecuadas para la funcionalidad deseada dado el contexto en que van a usarse. Razón más común: eficiencia. Otra: fragmentación.
- En sistemas embebidos y/o de tiempo real, un programa debe correr por períodos muy largos de tiempo, y en general requieren que la alocación de memoria siempre tarde la misma cantidad de tiempo.
- C++ permite sobrecargar estos operadores para implementar un esquema propio de alocación.

Operator `new` overload

- Cuando se sobrecarga el **`operator new()`** y el **`operator delete()`** solo se cambia la forma en que se aloca, el compilador conserva la responsabilidad de llamar al constructor y destructor sobre esa alocación.
- La sobrecarga de **`new`** y **`delete`** es como la sobrecarga de cualquier otro operador. Sin embargo se tiene la opción de sobrecargar el operador new global o usar un diferente alocador solo para una clase particular.

Global operator new overload

```
void *operator new(size_t sz)
{
    printf("operator new: %d Bytes\n", sz);
    void *m = malloc(sz);
    if (!m)
        puts("out of memory");
    return m;
}

void operator delete(void *m)
{
    puts("operator delete");
    free(m);
}
```

Local operator new overload

```
class Framis {
        enum { sz = 10 };
        char c[sz]; // To take up space, not used
        static unsigned char pool[];
        static bool alloc_map[];
    public:
        enum { psize = 100 }; // frami allowed
        Framis() { cout << "Framis()\n"; }
        ~Framis() { cout << "~Framis() ... "; }
        void *operator new(size_t);
        void operator delete(void *);
};

unsigned char Framis::pool[psize * sizeof(Framis)];
bool Framis::alloc_map[psize] = {false};
```

Local operator new overload

```
// Size is ignored -- assume a Framis object
void *Framis::operator new(size_t) {
    for (int i = 0; i < psize; i++)
        if (!alloc_map[i]) {
            cout << "using block " << i << endl;
            alloc_map[i] = true; // Mark it used
            return pool + (i * sizeof(Framis));
        }
    cout << "out of memory" << endl;
    throw bad_alloc();
}
```

```
void Framis::operator delete(void *m)
{
    if (!m)
        return; // Check for null pointer
    // Assume it was created in the pool
    // Calculate which block number it is:
    uintptr_t block = (uintptr_t)m -
                      (uintptr_t)pool;
    block /= sizeof(Framis);
    cout << "freeing block " << block << endl;
    // Mark it free:
    alloc_map[block] = false;
}
```

Local operator new overload

```
int main()
{
    Framis *f[Framis::psize];
    try{
        for (int i = 0; i < Framis::psize; i++)
            f[i] = new Framis;
        new Framis; // Out of memory
    } catch (bad_alloc) {
        cerr << "Out of memory!" << endl;
    }
    delete f[10];
    f[10] = nullptr;
    // Use released memory:
    Framis *x = new Framis;
    delete x;
    for (int j = 0; j < Framis::psize; j++)
        delete f[j]; // Delete f[10] OK
} ///:~
```

Sobrecarga de new/delete para arreglos

```
class Widget {
        enum { sz = 10 };
        int i[sz];
    public:
        Widget() { cout << "*"; }
        ~Widget() { cout << "~"; }

        void *operator new(size_t sz) {
            cout<< "\nWidget::new: "
                  << sz << " bytes" << endl;
            return ::new char[sz];
        }
        void operator delete(void *p) {
            cout<< "\nWidget::delete" << endl;
            ::delete[] p;
        }

//    ...
```

```
//    ...
        void *operator new[](size_t sz) {
            cout<< "\nWidget::new[]: "
                  << sz << " bytes" << endl;
            return ::new char[sz];
        }
        void operator delete[](void *p) {
            cout<< "\nWidget::delete[]" << endl;
            ::delete[] p;
        }
};

int main() {

    Widget *p = new Widget[10];

// ...

    delete [] p;
}
```

```
Widget::new[]: 408 bytes
**********~~~~~~~~~~
Widget::delete[]
```

new/delete en address específico

```
class X {
        int i;
    public:
        X(int ii = 0) : i(ii) {
            cout << "this = " << this << endl;
        }
        ~X() {
            cout << "X::~X(): " << this << endl;
        }
        void *operator new(size_t, void *loc) {
            return loc;
        }
};

int main()
{
    int space[10];
    cout << "space = " << space << endl;
    X *xp = new (space) X(47);  // X at location space
    xp->X::~X();                // Explicit destructor call
                                // ONLY use with placement! (Or overload delete to do nothing)
} ///:~
```

Operator new overloaded

```
void* operator new(size_t);
void* operator new[](size_t);
void operator delete(void*);
void operator delete[](void*);

void* operator new(size_t, const nothrow_t&);
void* operator new[](size_t, const nothrow_t&);
void operator delete(void*, const nothrow_t&);
void operator delete[](void*, const nothrow_t&);

void* operator new(size_t, void* );
void* operator new[](size_t, void*);
void operator delete(void*, void*);
void operator delete[](void*, void*);
```