Tipus Recursius de Dades I
Programació 2
Facultat d'Informàtica d'Informàtica, UPC

Conrado Martínez

Primavera 2019

- Apunts basats en els d'en Ricard Gavaldà
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Part I

Apuntadors

En C++, per a cada tipus `T` hi ha un altre tipus “apuntador a `T`”

Apuntadors

En C++, per a cada tipus `T` hi ha un altre tipus “apuntador a `T`”

Una variable de tipus “apuntador a `T`” pot contenir

- una referència a una variable o objecte de tipus `T`,
- o un valor especial `nullptr`
- o res sensat, si no ha estat inicialitzada

Apuntadors

En C++, per a cada tipus `T` hi ha un altre tipus “apuntador a `T`”

Una variable de tipus “apuntador a `T`” pot contenir

- una referència a una variable o objecte de tipus `T`,
- o un valor especial `nullptr`
- o res sensat, si no ha estat inicialitzada

La referència pot estar implementada amb una adreça de memòria o d’altres maneres; és irrellevant a Programació 2

Operadors

1. `T* p`: Declaració de variable `p` com “apuntador a `T`”
2. `*p`: Objecte referenciat pel apuntador `p`
3. `->`: Composició de `*` i el selector `.` de `struct/class`
4. `&v`: Referència a `v`, de tipus “apuntador al tipus de `v`”
5. `new`, `delete`: Creació i destrucció de memòria dinàmica

Exemple

```
int x;
int* p;
p = &x;
x = 5;
cout << *p << endl;   // escriu 5
*p = 3;
cout << x << endl;    // escriu 3
```

Observacions

Error accedir a `*p` si `p` no referencia cap objecte

és a dir, si `p == nullptr` o si `p` no inicialitzat

Observacions

```
Estudiant x;
Estudiant* p;
```

- `x` sempre referenciarà el mateix objecte mentre viu
- `*p` pot anar referenciant diferents objectes quan canviem el valor de `p`

Observacions

```
Estudiant x;
Estudiant* p;
```

- `x` sempre referenciarà el mateix objecte mentre viu
- `*p` pot anar referenciant diferents objectes quan canviem el valor de `p`

- Quan fem `p = &x`, tenim un objecte amb dos noms, `*p` i `x`
- Això se'n diu aliasing. Molt útil però pot ser perillós

Exemple

```
int x = 1;
int y = 2;
int* p = &x;
int* q = &y;
cout << x << " " << y << endl; // escriu "1 2"
*q = *p;
cout << x << " " << y << endl; // escriu "1 1"
*q = 3;
cout << x << " " << y << endl; // escriu "1 3"
q = p;
*q = 4;
cout << x << " " << y << endl; // escriu "4 3"
// en aquest punt, {\tt x} té tres noms: x, *p i *q
```

Preguntes

Declarem

```
int x; int* p; int* q;
```

És sempre cert que...

- `*(&x) == x`?
- `&(*p) == p`?
- `p == q` implica `(*p) == (*q)`?
- `(*p) == (*q)` implica `p == q`?

Apuntadors i structs

És molt freqüent tenir un apuntador a un struct o un objecte d'una classe, i voler accedir a un camp de l'struct apuntat, invocar un mètode de l'objecte, etc.

Notació còmoda: `p->camp` equival a `(*p).camp`, `p->mètode(...)` equival a `(*p).mètode(...)`

Apuntadors i structs

És molt freqüent tenir un apuntador a un struct o un objecte d'una classe, i voler accedir a un camp de l'struct apuntat, invocar un mètode de l'objecte, etc.

Notació còmoda: `p->camp` equival a `(*p).camp`, `p->mètode(...)` equival a `(*p).mètode(...)`

Exemple

```
struct par {
  string nom;
  int edat;
};

par* ppar = ...;
++ppar -> edat;

Estudiant* pe = ...
...
if (pe->te_nota()) { cout << pe->consultar_DNI() << endl; }
```

Apuntadors i structs

És molt freqüent tenir un apuntador a un struct o un objecte d'una classe, i voler accedir a un camp de l'struct apuntat, invocar un mètode de l'objecte, etc.

Notació còmoda: `p->camp` equival a `(*p).camp`, `p->mètode(...)` equival a `(*p).mètode(...)`

Exemple

```
struct par {
  string nom;
  int edat;
};

par* ppar = ...;
++ppar -> edat;

Estudiant* pe = ...
...
if (pe->te_nota()) { cout << pe->consultar_DNI() << endl; }
```

Definint un apuntador

Quan declarem un apuntador `T* p`, està *indefinit*. El definim:

- Fent-lo apuntar a un objecte del tipus `T` ja existent:

```
p = q o p = &x;
```

Definint un apuntador

Quan declarem un apuntador `T* p`, està *indefinit*. El definim:

- Fent-lo apuntar a un objecte del tipus `T` ja existent:

  ```
  p = q o p = &x;
  ```

- O donant-li el valor `nullptr`, per explicitar “no referencia res”

Definint un apuntador

Quan declarem un apuntador `T* p`, està *indefinit*. El definim:

- Fent-lo apuntar a un objecte del tipus `T` ja existent:
  `p = q` o `p = &x;`
- O donant-li el valor `nullptr`, per explicitar “no referencia res”
- Reservant memòria perquè apunti a un nou objecte:
  `p = new T;`

Definint un apuntador

Quan declarem un apuntador `T* p`, està *indefinit*. El definim:

- Fent-lo apuntar a un objecte del tipus `T` ja existent:

  ```
  p = q o p = &x;
  ```

- O donant-li el valor `nullptr`, per explicitar "no referencia res"
- Reservant memòria perquè apunti a un nou objecte:

  ```
  p = new T;
  ```

  - Aquest objecte no tindrà nom propi: només `*p`
  - Queda inaccessible! si modifiquem `p` i no hi ha cap altre apuntador que l'hi apunta

`new` and `delete`

Operacions de gestió de memòria dinàmica:

- `new T`: reserva memòria dinàmica per a un nou objecte, li aplica la creadora de `T` i retorna un apuntador a ell
- `delete p`: aplica la destructora del tipus a l'objecte apuntat per `p` i allibera la memòria que ocupa ("esborra" l'objecte)

`new` and `delete`

Operacions de gestió de memòria dinàmica:

- `new T`: reserva memòria dinàmica per a un nou objecte, li aplica la creadora de `T` i retorna un apuntador a ell
- `delete p`: aplica la destructora del tipus a l'objecte apuntat per `p` i allibera la memòria que ocupa ("esborra" l'objecte)
- Atenció: "delete p" NO esborra el punter `p`; esborra l'objecte apuntat per `p`
- el valor de `p` després de `delete p` és indefinit

Exemples

```
struct T {
    int camp1;
    bool camp2;
}
void f(...) {
  T x;                  // es crida la creadora de T
  x.camp1 = 20; x.camp2 = true;
  T* p = new T;         // p apunta a un objecte nou;
                        // crida la constructora de T
  p->camp1 = 30; p->camp2 = false;
  ...
  delete p;     // es crida destructora de T
                // i s'allibera *p; p indefinit
// i aquí es crida automàticament a la destructora de T
// de la variable local x
}
```

`new` and `delete`: Errors

- Deixar memòria sense alliberar (objectes dinàmics sense esborrar) $\rightarrow$ memory leaks

`new` and `delete`: Errors

- Deixar memòria sense alliberar (objectes dinàmics sense esborrar) $\rightarrow$ memory leaks
- Accedir a memòria ja alliberada (objectes esborrats). Vigileu amb l'aliasing $\rightarrow$ dangling references

`new` and `delete`: Errors

- Deixar memòria sense alliberar (objectes dinàmics sense esborrar) $\rightarrow$ memory leaks
- Accedir a memòria ja alliberada (objectes esborrats). Vigileu amb l'aliasing $\rightarrow$ dangling references
- `delete` de memòria no creada amb `new`

`new` and `delete`: Errors

- Deixar memòria sense alliberar (objectes dinàmics sense esborrar) $\rightarrow$ memory leaks
- Accedir a memòria ja alliberada (objectes esborrats). Vigileu amb l'aliasing $\rightarrow$ dangling references
- `delete` de memòria no creada amb `new`
- confondre "`p = nullptr`" amb "`delete p`"
  - els dos s'hauran d'usar, però en circumstàncies diferents

Exemples d'errors

```
void f(...) {
  T x; ...
  T* p = &x;
  T* q = new T;
  T* r = q;  // r i q apunten al mateix valor
  T* s = new T;
  ...
  delete p; // ERROR: *p no creat amb new
  delete q; // OK
  if ((q->camp1 == 0) {...} // ERROR: q indefinit
  if (q == nullptr) {...}  // PERILL: q indefinit
  r->camp1 = 3;    // ERROR: r indefinit, *r
                   // alliberat amb delete q
  // ERROR: no fem delete s i *s inaccessible: leak!
}
```

Vectors d'apuntadors

Els apuntadors permeten moure objectes més eficientment.

```
void copiar(const vector<Estudiant*>& v, vector<Estudiant*>& w) {
   for (int i = 0; i < v.size(); ++i) {
     w[i] = v[i];
     v[i] = nullptr;
   }

}
```

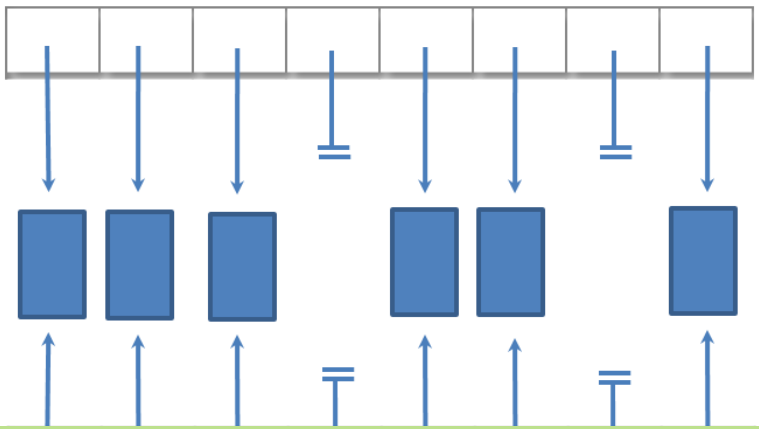
Vectors d'apuntadors

Els apuntadors permeten moure objectes més eficientment.

```
void copiar(const vector<Estudiant*>& v, vector<Estudiant*>& w) {
   for (int i = 0; i < v.size(); ++i) {
     w[i] = v[i];
     v[i] = nullptr;
   }

}
```

Si no posem `nullptr`s en `v` tenim:



Assignació, còpia, destrucció

Còpia:

- Assignació entre apuntadors a objectes no implica una còpia d'objectes
- Fonamental definir **constructora per còpia** per al corresponent tipus. Usarà `new`. La constructora per còpia per defecte crea un nou object a partir d'un altre , copiant atribut a atribut.
- També sovint es redefineix també l'operació $=$, per defecte fa assignació atribut a atribut de l'objecte origen a l'objecte destí. Copiar/assignar atributs que siguin punters $\rightarrow$ aliasing!

Assignació, còpia, destrucció

Esborrament:

- La destructora per defecte destruirà atributs que siguin punters però no els objectes als quals apuntin! $\rightarrow$ memory leaks
- Cal definir la destructora de la classe de manera que s'alliberi tots els objects creats a meòria dinàmica per a representar un object de la classe

Pas d'apuntadors com a paràmetres

Pas d'un objecte X que conté apuntadors com a paràmetre d'entrada:

- Pas per valor:
  - Fa servir la constructora per còpia, hem definirla si la constructora per còpia per defecte no serveix
  - Si l'objecte X té atributs que són punters, la constructora per còpia per defecte ens porta a una situació d'aliasing
- Pas per referència: passem X no es canviarà, però no es garanteix que no es modifiquin objectes apuntats per components de X

Part I

Tipus recursius de dades?

Els programes són dades més operacions (p.ex., accions o funcions)

Hem vist accions i funcions recursives:

casos directes + casos recursius

Té sentit parlar de tipus de dades recursius?

Tipus recursius de dades?

De fet, hem pensat en alguns tipus de manera recursiva:

- Piles: una pila o bé és buida o bé és push(una altra pila,valor)
- Cues, llistes: idem
- Arbres: un arbre, o bé és buit o bé és plantar(valor,arbre1,arbre2)

Tipus recursius de dades?

De fet, hem pensat en alguns tipus de manera recursiva:

- Piles: una pila o bé és buida o bé és push(una altra pila,valor)
- Cues, llistes: idem
- Arbres: un arbre, o bé és buit o bé és plantar(valor,arbre1,arbre2)

Només n'hem vist algunes implementacions amb vectors, no recursives

Tipus recursius de dades?

De fet, hem pensat en alguns tipus de manera recursiva:

- Piles: una pila o bé és buida o bé és push(una altra pila,valor)
- Cues, llistes: idem
- Arbres: un arbre, o bé és buit o bé és plantar(valor,arbre1,arbre2)

Només n'hem vist algunes implementacions amb vectors, no recursives
Una definició recursiva d'aquests tipus de dades podria donar:

- Correspondència natural amb definició recursiva
- No posar límits *a priori* en la mida

Tipus recursius de dades?

Implementació en C++??

```
class stack<T> {
private:
    bool es_buida;
    T valor;
    stack<T> resta_pila;
public:
    ...
};
```

Problema: En C++, quan es crea un objecte es crida recursivament a les creadores de totes les seves components. Procés infinit

Com es fa: la Pila

```
template <class T> class stack {
  private:
    // tipus privat nou
    struct node_pila {
        T info;
        node_pila* seg;  // <-- recursivitat
    };

    int altura;                 // guardada un sol cop
    node_pila* cim;  // primer d'una cadena de nodes

    ...  // especificació d'operacions privades

  public:
    ...  // especificació d'operacions públiques
};
```

Com es fa: la Pila

```
template <class T> class stack {
  private:
    // tipus privat nou
    struct node_pila {
        T info;
        node_pila* seg;  // <-- recursivitat
    };

    int altura;                 // guardada un sol cop
    node_pila* cim;  // primer d'una cadena de nodes

    ...  // especificació d'operacions privades

  public:
    ...  // especificació d'operacions públiques
};
```

Els apuntadors `seg` no s'inicialitzen automàticament: no es creen objectes recursivament quan es crea un stack

Definició d'una estructura de dades recursiva I

Dos nivells:

- Superior: classe amb atributs
  - Informació global de l'estructura (que no volem que es repeteixi per a cada element)
  - Apuntadors a alguns elements distingits (el primer, l'últim, etc., segons el que calgui).
- Inferior: *struct* privada que defineix nodes enllaçats per apuntadors
  - informació d'un i només un element de l'estructura
  - apuntador a un o més nodes "següents"

Avantatges de les estructures de dades recursives

- Correspondència natural amb una definició recursiva abstracta

Avantatges de les estructures de dades recursives

- Correspondència natural amb una definició recursiva abstracta
- No cal fixar a priori un nombre màxim d'elements
- Es pot anar demanant memòria per als nous nodes a mesura que s'hi volen afegir elements

Avantatges de les estructures de dades recursives

- Correspondència natural amb una definició recursiva abstracta
- No cal fixar a priori un nombre màxim d'elements
- Es pot anar demanant memòria per als nous nodes a mesura que s'hi volen afegir elements
- Eficiència: modificant enllaços entre nodes podem:
  - inserir o esborrar elements - sense moure els altres
  - moure parts senceres de l'estructura - sense fer còpies

Part I

Implementació de piles

```
template <class T> class stack {
  private:
    // tipus privat nou
    struct node_pila {
        T info;
        node_pila* seg;  // nullptr indica final de cadena
    };

    int altura;                 // guardada un sol cop
    node_pila* cim;  // element en el cim de la pila

    ...  // especificació d'operacions privades

  public:

    ...  // especificació d'operacions públiques
};
```

Mètodes públics: construcció/destrucció

```
stack() {
    altura = 0;
    cim = nullptr;
}

// Constructora per copia
stack(const stack& original) {
    altura = original.altura;
    cim = copia_node_pila(original.cim);
}
```

Mètodes públics: construcció/destrucció, modificació

```
~stack() {
    esborra_node_pila(cim);
}


void clear() {
    esborra_node_pila(cim);
    altura = 0;
    cim = nullptr;
}
```

Mètodes públics: consultors

```
T top() const {
// Pre: el p.i. és una pila no buida
// = en termes d'implementacio, cim != nullptr
    return cim -> info;
}

bool empty() const {
    return cim == nullptr;
}

int size() const {
    return altura;
}
```

Mètodes públics: modificadors

```
void push(const T& x) {
    node_pila* aux = new node_pila; // espai per al nou element
    aux -> info = x;
    aux -> seg = cim
    cim = aux;
    ++altura;
}
```

Mètodes públics: modificadors

```
void pop() {
  // Pre: el p.i. és una pila no buida
  // => cim != nullptr
    node_pila* aux = cim; // conserva l'accés a primer
    cim = cim -> seg; // avança
    delete aux; // allibera l'espai de l'antic cim
    --altura;
}
```

Mètodes privats I

```
static node_pila* copia_node_pila(node_pila* m) {
/* Pre: cert */
/* Post: si m és nullptr, el resultat és nullptr; en cas contrari,
         el resultat apunta al primer node d'una cadena
         de nodes que són còpia de la cadena que té
         el node apuntat per m com a primer */
    if (m == nullptr) return nullptr;
    else {
        node_pila* n = new node_pila;
        n -> info = m -> info;
        n -> seg = copia_node_pila(m -> seg);
        return n;
    }
}
```

Exercici: Versió iterativa

Mètodes privats II

```
static void esborra_node_pila(node_pila* m) {
/* Pre: cert */
/* Post: no fa res si m és nullptr, en cas contrari,
         allibera espai dels nodes de la cadena que
         té el node apuntat per m com a primer */
    if (m != nullptr) {
      esborra_node_pila(m -> seg);
      delete m;
    }
}
```

Exercici: Versió iterativa

Mètodes públics: redefinició operador assignació

```
stack<int> p1, p2, p3;
...
p1 = p2 = p3;
```

Mètodes públics: redefinició operador assignació

```
stack<int> p1, p2, p3;
...
p1 = p2 = p3;
```

L'assignació en C++ és un operador: una funció que retorna un valor, amb paràmetre implícit que queda modificat, i un paràmetre explícit no modificable

Mètodes públics: redefinició operador assignació

```
stack<int> p1, p2, p3;
...
p1 = p2 = p3;
```

L'assignació en C++ és un operador: una funció que retorna un valor, amb paràmetre implícit que queda modificat, i un paràmetre explícit no modificable

Return (*this): necessari per a encadenaments d'assignacions

```
stack& operator=(const stack& original) {
    if (this != &original) {
        node_pila* aux = copia_node_pila(original.cim);
        esborra_node_pila(cim);   // si no, leak!
        altura = original.altura;
        cim = aux;
    }
    return *this;
}
```

Implementació de cues

- Cal poder accedir tant tant al primer element (per consultar-lo o eliminar-lo) com a l'últim (per afegir un de nou)
- Atribut per la llargada (o mida) de la cua

Definició de la classe

```
template <class T> class queue {
  private:
    struct node_cua {
        T info;
        node_cua* seg;
    };
    int longitud;
    node_cua* primer;
    node_cua* ultim;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

Mètodes privats: copiar i esborrar cadenes I

```
static node_cua* copia_node_cua(node_cua* m, node_cua*& u) {
/* Pre: cert */
/* Post: si m és nullptr, el resultat i u són nullptr; en cas contrari,
   el resultat apunta al primer node d'una cadena de nodes
   que són còpia de de la cadena que té el node apuntat per m
   com a primer, i u apunta a l'últim node */

    if (m == nullptr) { u = nullptr; return nullptr; }
    else {
        node_cua* n = new node_cua;
        n -> info = m -> info;
        n -> seg = copia_node_cua(m- > seg, u);
        if (n -> seg == nullptr) u = n;
        return n;
      }
}
```

Mètodes privats: copiar i esborrar cadenes II

```
// op privada
static void esborra_node_cua(node_cua* m) {
/* Pre: cert */
/* Post: no fa res si m és nullptr, en cas contrari, allibera
         els nodes de la cadena que té el node apuntat
         per m com a primer */

    if (m != nullptr) {
        esborra_node_cua(m ->seg);
        delete m;
    }
}
```

Mètodes privats: construcccció/destrucció

```
queue() {
    longitud = 0;
    primer = ultim = nullptr;
}

queue(const queue& original) {
    longitud = original.longitud;
    primer = copia_node_cua(original.primer, ultim);
}

~queue() {
    esborra_node_cua(primer);
}
```

Mètodes públics: redefinició de l'operador d'assignació

```
queue& operator=(const queue& original) {
    if (this != &original) {
      node_cua* auxp, *auxu;
      auxp = copia_node_cua(original.primer, auxu);
      esborra_node_cua(primer); // si no, leak!
      longitud = original.longitud;
      primer = auxp;
      ultim = auxu;
    }
    return *this;
}
```

Mètodes públics: modificadors I

```
void clear() {
    esborra_node_cua(primer);
    longitud = 0;
    primer_node = nullptr;
    ultim_node = nullptr;
}

void push(const T& x) {
    node_cua* aux = new node_cua;
    aux -> info = x;
    aux -> seg = nullptr;
    if (primer == nullptr) primer = aux;
    else ultim -> seg = aux;
    ultim = aux;
    ++longitud;
}
```

Mètodes públics: modificadors I

```
void pop() {
// Pre: el p.i. és una cua no buida
// = en termes d'implementacio, primer != nullptr
    node_cua* aux = primer;
    if (primer == ultim) {
        primer = ultim = nullptr;
    } else primer = primer -> seg;
    delete aux;
    --longitud;
}
```

Mètodes públics: consultors

```
T front() const {
// Pre: el p.i. és una cua no buida
// = en termes d'implementacio, primer != nullptr
    return primer -> info;
}

bool empty() const {
    return longitud == 0;
}

int size() const {
    return longitud;
}
```

Exemple d'increment d'eficiència

```
// Pre: cert
// Post: retorna cert si la cua conté x
bool cerca(const T& x) const {
    node_cua* aux = primer;
    while (aux != nullptr) {
        if (aux -> info == x) return true;
        aux = aux -> seg;
    }
    return false;
}
```

la cua és const &, no és destruida, no hi ha còpies

. . . però s'ha de tenir accés a la representació!