- Entrada/Sortida bàsica
- Funcions
- Vectors
- Llistes i iteradors
- Classes
- Punters
- Exemple complet: Pila genèrica
- The Elements of C++ Style

Basat en:

- J. Petit, S. Roura, A. Atserias. Anàlisi i Disseny d'Algorismes (ADA). Algorismes en C++. 2005.
- T. Misfeldt, G. Bumgardner, A. Gray. *The Elements of C++ Style*. Cambridge University Press, 2004.

Programa en C++ que escriu un missatge per la sortida estàndard.

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

int main () { // funcio principal
  cout << "Hola a tothom!" << endl;
}
```

Llegeix dos números per l'entrada estàndard i n'escriu la suma per la sortida estàndard.

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

int main () {
  int a,b;
  cin >> a >> b;
  int s = a + b;
  cout << a << " + " << b << " = " << s << endl;
}
```

Calcula la mitjana d'una seqüència de reals llegida per l'entrada estàndard.

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

int main () {
  double s = 0;
  int n = 0;
  double x;
  while (cin >> x) {
    s += x;
    ++n;
  }
  cout << s/n << endl;
}
```

Funció que calcula el màxim de dos enters.

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

int max (int a, int b) {
  return a > b ? a : b;
}

int main () {
  int x,y;
  cin >> x >> y;
  cout << max(x,y) << endl;
}
```

Sobrecàrrega de noms.

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

int max (int a, int b) { return a > b ? a : b; }

double max (double a, double b) { return a > b ? a : b; }

string max (string a, string b) { return a > b ? a : b; }

int main () {
  cout << max(3,4) << endl;
  cout << max(3.5,12.3) << endl;
  cout << max(3,15.0) << endl;
  cout << max("Hola","Adeu") << endl;
}
```

Funció màx genèrica.

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

template <typename tipus>
tipus max (tipus a, tipus b) {
  return a > b ? a : b;
}

int main () {
  cout << max(3,4) << endl;
  cout << max(1.12,3.14) << endl;
  cout << max("bon dia","bona tarda") << endl;
  cout << max('a','d') << endl;
}
```

Intercanvi incorrecte: Paràmetres per valor.

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

void swap (int x, int y) {
  int z = x;
  x = y;
  y = z;
}

int main () {
  int a,b;
  cin >> a >> b;
  swap(a,b);
  cout << a << b;
}
```

Intercanvi correcte: Paràmetres per referència.

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

void swap (int& x, int& y) {
  int z = x;
  x = y;
  y = z;
}

int main () {
  int a,b;
  cin >> a >> b;
  swap(a,b);
  cout << a << b;
}
```

Funció d'intercanvi genèrica.

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

template <typename T>
void swap (T& x, T& y) {
  T z = x;
  x = y;
  y = z;
}

int main () {
  int a,b;
  cin >> a >> b;
  swap(a,b);
  cout << a << b;
}
```

Declaracions de vectors.

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

int main () {
  // Vector per a 10 enters
  vector<int> v1(10);
  // Vector per a 20 reals, tots inicialitzats a 1
  vector<double> v2(20,1);
  // Vector de caracters, sense capacitat
  vector<char> v3;
  // Vector de booleans, eficient en espai pero no en temps
  vector<bool> vb(1000);
  // Matriu de 10 x 20 reals
  vector< vector<double> > M(10,20);
}
```

Ús de vectors.

- `[]` accedeix a les posicions (compte si fora de rang!),
- `size()` retorna la seva talla,
- `empty()` diu si la talla és zero,
- `front()` retorna el primer element,
- `back()` retorna el darrer element,
- `push_back()` afageix un element pel final del vector,
- els vectors es poden assignar entre ells.

Ús de vectors. (cont.)

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
  vector<int> v(10);
  for (int i=0; i<v.size(); ++i) { v[i] = i; }

  vector<int> v2 = v;
  v2.push_back(49);
  cout << v2.front() << " " << v2.back() << endl;
  v2.pop_back();
  cout << v2.empty() << endl;

  vector<int> v3 = v2;
}
```

Llistes.

- el constructor crea una llista buida,
- `push_back()` afegeix un element pel final,
- `push_front()` afegeix un element pel davant,
- `pop_back()` esborrar un element pel final,
- `pop_front()` esborrar un element pel davant,
- `front()` retorna el primer element,
- `back()` retorna el darrer element,
- `size()` retorna la seva talla,
- `empty()` diu si la talla és zero,
- altres mètodes: `sort()`, `reverse()`, `merge()`, `unique()`, `splice()`, ...

Llistes. (cont.)

```
#include <iostream>
#include <list>
using namespace std;
int main () {
  list<int> L;
  L.push_back(3); L.push_back(4); L.push_back(5);
  L.push_front(2); L.push_front(1); L.push_front(0);
  cout << L.size()  << " " << L.empty() << endl;
  cout << L.front() << " " << L.back()  << end;

  list<int> L2 = L;
  L2.reverse();
  cout << L2.front() << " " << L2.back()  << end;
  L2.sort();
  cout << L2.front() << " " << L2.back()  << end;
}
```

Iteradors.

- Els iteradors permeten recorrer els contenidors com ara les llistes.
- Els mètodes `begin()` i `end()` retornen iteradors.
- L'iterador a `end()` senyala el final de la llista (no s'hi pot accedir).

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

typedef list<int>::iterator iter;

void escriure_llista (list<int>& L) {
  for (iter it=L.begin(); it!=L.end(); ++it) {
    cout << *it << endl;
} }
```

Iteradors. (cont.)

```
bool hi_es (list<int>& L, int x) {
  iter it = L.begin();
  while (it!=L.end() and *it!=x) ++it;
  return it!=L.end();
}

bool hi_es2 (list<int>& L, int x) {
  for (iter it=L.begin(); it!=L.end(); ++it) {
    if (*it==x) return true;
  }
  return false;
}
```

Iteradors.

- Els iteradors també permeten fer insercions en punts qualsevols.
- En aquest exemple es mostra una ordenació per inserció.

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

typedef list<int> llista;
typedef llista::iterator iter;
```

Iteradors. (cont.)

```
iter cerca_pos (llista& L, int x) {
  iter it=L.begin();
  while (it!=L.end() and *it<x) ++it;
  return it;
}

llista ordenar (llista& L) {
  llista L2;
  for (iter it=L.begin(); it!=L.end(); ++it) {
    iter it2 = cerca_pos(L2,*it);
    L2.insert(it2,*it);
  }
  return L2;
}
```

Les estructures permeten agrupar diverses dades en una de sola.

```
struct data {
  int dia, mes, any;
};
struct persona {
  string nom,cognom1,cognom2;
  data naix;
};

data d;
d.dia = 30; d.mes = 11; d.any = 1971;

persona p;
p.nom = "Jordi";
p.naix = d;
cout << p.naix.any;
```

Les estructures poden definir constructors, destructors i mètodes.

```
struct data {
  int dia, mes, any;

  data (int d, int m, int a) {
    dia = d;  mes = m;  any = a;
  }

  bool legal () { ... }
  void incrementa () { ... }
};

data d(30,11,1971);
if (d.legal()) {
  data d2 = d;
  d2.incrementa();
}
```

Classes.

- Les classes són el mateix que les estructures, però en elles se sol definir una part privada que implementa una interfície pública.

```
class data {
private:
  int dia, mes, any;

public:
  data (int d, int m, int a) {
    dia = d;  mes = m;  any = a;
  }

  bool legal () { ... }

  void incrementa () { ... }
};
```

Classes. (cont.)

```
data d(30,11,1971);
if (d.legal()) {
  data d2 = d;
  d2.incrementa();
}
```

Operadors.

- Es poden definir operadors aritmètics i relacionals.

```
class data {
private:
  int dia, mes, any;
public:
  data (int d, int m, int a) {
    dia = d;  mes = m;  any = a;
  }
  bool legal () { ... }
  data& operator++ () {...}
  data& operator+= (int n) {...}
  friend static bool operator<  (data d1, data d2) {...}
  friend static bool operator== (data d1, data d2) {...}
  ...
};
```

Punters.

- Declaració d'un punter `p` a un enter:

```
int* p;
```

- Assignació d'un punter nul a `p`:

```
p = 0;
p = null; // null es una constant definida a <ada.hh>
```

- Agafar memòria dinàmica per a un enter i guardar-ne el seu punter:

```
p = new int;
```

- Desreferenciar el punter: (evidentment, no es pot seguir mai un punter nul):

```
*p = 21;
if (*p==13) ...
```

- Alliberar la memòria dinàmica:

```
delete p;
```

Punters. (cont.)

- Els punters que apunten a un mateix tipus es poden assignar (amb =) i comparar (amb == o !=).

```
int* p1 = ...;
int* p2 = ...;
int* p3 = ...;
if (p1==p2) {
  p3 = p1;
}
```

- Els punters es converteixen automàticament en booleans si cal: nul es converteix a fals, no nul es converteix a cert.

```
int* p = ...;
if (p and *p!=24) ...
```

Punters i estructures (o classes).

- En aquest curs sempre usarem els punters per apuntar a estructures. En aquest cas `*punter.camp` no funciona, cal escriure `(*punter).camp` o, més fàcil, `punter->camp`.
- Es mostra un exemple per definir llistes enllaçades.

```
struct node {
  node* seg;
  int   inf;
};
typedef node* llista;
bool hi_es (llista p, int x) {
  node* q = p;
  while (q and q->inf!=x) q = q->seg;
  return q;
}
```

Classes i punters.

- El fet d'utilitzar memòria dinàmica per a implementar classes fa que calgui anar en compte amb no tenir problemes d'aliasing i fuites de memòria. Per això, en aquest cas, sovint caldrà definir operadors i constructors de còpia i destructors.
- Vegeu l'exemple de la pila.

Exemple complet: Pila genèrica amb memòria dinàmica.

```
template <typename T>
class Pila {

  struct node {
    T x;          // informacio
    node* s;      // punter al node seguent
  };

  int n;          // nombre d'elements
  node* c;        // punter al cim
```

Exemple complet: Pila genèrica amb memòria dinàmica. (cont.)

```
public:

  // Constructor: crea una pila buida
  Pila () {
    c = null;  n = 0;
  }

  // Constructor de copia
  Pila (const Pila& P) {
    c = copiar(P.c);  n = P.n;
  }

  // Destructor
  ~Pila () {
    alliberar(c);
  }
```

Exemple complet: Pila genèrica amb memòria dinàmica. (cont.)

```
// Operador de copia
Pila& operator= (const Pila& P) {
  if (&P!=this) {
    alliberar(c);  c = copiar(P.c);  n = P.n;
  }
  return *this;
}

// Empila x sobre la pila
void empilar (T x) {
  node* p = new node;  p->x = x;  p->s = c;
  c = p;  ++n;
}
```

Exemple complet: Pila genèrica amb memòria dinàmica. (cont.)

```
// Treu l'element del cim de la pila (error si es buida)
void desapilar () {
  if (buida()) throw ErrorPrec("Pila buida");
  node* p = c;  c = c->s;  delete p;  --n;
}

// Retorna l'element del cim de la pila (error si buida)
T cim () {
  if (buida()) throw ErrorPrec("Pila buida");
  return c->x;
}

// Indica si la pila es buida
bool buida () { return n==0; }
```

Exemple complet: Pila genèrica amb memòria dinàmica. (cont.)

```
  // Retorna el nombre d'elements a la pila
  int talla () {
    return n;
  }

private:

  void alliberar (node* p) {
    if (p) {
      alliberar(p->s);
      delete p;
  } }
```

Exemple complet: Pila genèrica amb memòria dinàmica. (cont.)

```
  node* copiar (node* p) {
    if (p) {
      node* q = new node;
      q->x = p->x;
      q->s = copiar(p->s);
      return q;
    } else {
      return null;
  } }

};
```

Exemple complet: Pila genèrica amb memòria dinàmica. (cont.)

```
// Exemple d'us:

int main () {
  Pila<int> p;
  p.apilar(3); p.apilar(5);
  while (!p.buida()) {
    cout << p.cim() << endl;
    p.desapilar();
} }
```

116. *Use Templates Instead of Macros to Create Parameterized Code*

116. *Use Templates Instead of Macros to Create Parameterized Code*

- `#define MAX(a,b) (((a) > (b)) ? (a) : (b))`

116. *Use Templates Instead of Macros to Create Parameterized Code*

- `#define MAX(a,b) (((a) > (b)) ? (a) : (b))`

–
```
int a = 0;
int r;
r = MAX(a++, 0); // a incremented once: r = 0
r = MAX(a++, 0); // a incremented twice: r = 2
```

116. *Use Templates Instead of Macros to Create Parameterized Code*

- `#define MAX(a,b) (((a) > (b)) ? (a) : (b))`
  - ```
    int a = 0;
    int r;
    r = MAX(a++, 0); // a incremented once: r = 0
    r = MAX(a++, 0); // a incremented twice: r = 2
    ```
- ```
  template<class T>
  inline T max(const T& a, const T& b) {
    return (a > b) ? a : b;
  }
  ```

116. *Use Templates Instead of Macros to Create Parameterized Code*

- `#define MAX(a,b) (((a) > (b)) ? (a) : (b))`

–
```
int a = 0;
int r;
r = MAX(a++, 0); // a incremented once: r = 0
r = MAX(a++, 0); // a incremented twice: r = 2
```

-
```
template<class T>
inline T max(const T& a, const T& b) {
  return (a > b) ? a : b;
}
```

–
```
int a = 0;
int r;
r = max(a++, 0); // a incremented once: r = 0
r = max(a++, 0); // a incremented once: r = 1
```

46. *Document Important Preconditions, Postconditions, and Invariant Conditions*

46. *Document Important Preconditions, Postconditions, and Invariant Conditions*
59. *Program by Contract*

46. *Document Important Preconditions, Postconditions, and Invariant Conditions*
59. *Program by Contract*

```
class Object;
```

46. *Document Important Preconditions, Postconditions, and Invariant Conditions*

59. *Program by Contract*

```
class LinkedList {
  public:
    void prepend(Object* object);
    const Object* first(void) const;
  protected:
    virtual void doPrepend(Object* object);
};
```

46. *Document Important Preconditions, Postconditions, and Invariant Conditions*

59. *Program by Contract*

```
void
LinkedList::prepend(Object* object) {
  // Test precondition
  assert(object);

  doPrepend(object);

  // Test postcondition
  assert(first() == object);
}
```

46. *Document Important Preconditions, Postconditions, and Invariant Conditions*

59. *Program by Contract*

```
class Stack : public LinkedList {
  protected:
    virtual void doPrepend(Object* object);
};
```

46. *Document Important Preconditions, Postconditions, and Invariant Conditions*
59. *Program by Contract*
152. *Use Assertions to Enforce a Programming Contract*