

# Tema 4: Memòria dinàmica

## Programació 2

---

Grau en Enginyeria Informàtica  
Universitat d'Alacant  
Curs 2024-2025



1. Organització de la memòria
2. Punters
3. Ús de punters
4. Referències
5. Implementació d'una pila

# Organització de la memòria

---

# Memòria estàtica

- Les *dades estàtiques* són aquelles la grandària de les quals és fixa i es coneix en escriure el programa
- Les variables que hem usat fins ara són estàtiques:

```
int i=0;  
char c;  
float vf[3]={1.0,2.0,3.0};
```

i	c	vf[0]	vf[1]	vf[2]
0		1.0	2.0	3.0
1000	1002	1004	1006	1008

- Permet emmagatzemar grans volums de dades, la quantitat exacta de les quals es desconeix en implementar el programa
- Durant l'execució del programa s'ajusta l'ús de la memòria al que es necessita a cada moment
- En C++ es pot fer ús de la memòria dinàmica usant punters

# Zones de la memòria

- Durant l'execució d'un programa, s'utilitzen zones diferenciades de la memòria:

Pila ( <i>stack</i> )
Monticle( <i>heap</i> )
Segment de dades
Codi del programa

- La *pila* emmagatzema les dades locals d'una funció: paràmetres per valor i variables locals

- El *monticle* emmagatzema les dades dinàmiques que es van reservant durant l'execució del programa

- Al *segment de dades* s'emmagatzemen les dades d'aquests tipus, la grandària dels quals es coneix en temps de compilació

- El codi mateix també s'emmagatzema en la memòria, com les dades

# Punters

---

# Definició i declaració

- Un *punter* emmagatzema l'adreça de memòria on es troba una altra dada
- Es diu que el punter “apunta” a aquesta dada
- Els punters es declaren usant el caràcter \*
- La dada al qual apunta el punter serà d'un tipus concret que haurà d'indicar-se en declarar el punter:

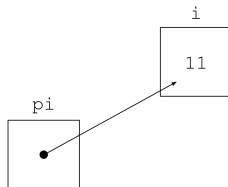
```
int *punterEnter; // Punter a enter
char *punterChar; // Punter a caràcter
int *vecPuntersEnter[20]; // Array de punters a enter
double **doblePunterReal; // Punter a punter a real
```



## Operadors de punters (1/2)

- L'operador `*` permet accedir al contingut de la variable a la qual apunta el punter
- L'operador `&` permet obtenir l'adreça de memòria en la qual està emmagatzemada una variable:

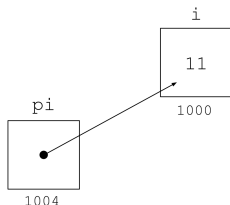
```
int i=3;  
int *pi;  
pi=&i; // pi conté l'adreça de memòria d'i  
*pi = 11; // Contingut de pi és "11". Per tant i = 11
```



## Operadors de punters (2/2)

- Suposant que `i` està en la posició de memòria 1000 i que `pi` està en la 1004:

```
int i=11;  
int *pi;  
pi=&i;  
cout << pi << endl; // Mostra "1000"  
cout << *pi << endl; // Mostra "11"  
cout << &pi << endl; // Mostra "1004"
```



# Declaració amb inicialització

- Com qualsevol altra variable, podem inicialitzar un punter en el moment de la declaració:

```
int *pi=&i; // pi conté l'adreça d'i
```

- Quan volem indicar que un punter no apunta a cap dada vàlida hi assignem el valor `NULL`:

```
int *pi=NULL;
```

- `NULL` és una constant entera amb valor zero. Des de l'estàndard C++ 2011, la constant `nullptr` pot usar-se també ja que representa el zero com una adreça de memòria (tipus punter)

## Exercici 1

Indica quina seria l'eixida per pantalla d'aquests fragments de codi:

```
int e1;  
int *p1,*p2;  
e1=7;  
p1=&e1;  
p2=p1;  
e1++;  
(*p2)+=e1;  
cout << *p1;
```

```
int a=7;  
int *p=&a;  
int **pp=&p;  
cout << **pp;
```

# Ús de punters

---

## Reserva i alliberament de memòria (1/2)

- L'operador `new` permet reservar memòria de manera dinàmica durant l'execució del programa
- Retorna l'adreça d'inici de la memòria reservada
- Si no hi ha suficient memòria per a la reserva, retorna `NULL`
- S'ha d'usar un punter per a emmagatzemar l'adreça que retorna `new`:

```
double *pd;  
pd=new double; // Reserva memòria per a un double  
if(pd!=NULL){ // Comprova que s'ha pogut reservar  
    *pd=4.75;  
    cout << *pd << endl; // Mostra "4.75"  
}
```



## Reserva i alliberament de memòria (2/2)

- L'operador `delete` permet alliberar memòria reservada amb `new`:

```
double *pd;  
pd=new double; // Reserva memòria  
...  
delete pd; // Allibera la memòria apuntada per pd  
pd=NULL; // Convenient si seguim usant pd
```

- Sempre que es reserva amb `new` cal alliberar amb `delete`
- Un punter es pot reutilitzar després d'alliberar el seu contingut i reservar memòria una altra vegada amb `new`:

```
double *pd;  
pd=new double; // Reserva memòria  
...  
delete pd; // Allibera la memòria apuntada per pd  
pd=new double; // Reservem de nou memòria  
...
```

## Punters i arrays (1/3)

- Existeix una estreta relació entre els punters i els arrays
- La variable de tipus array és en realitat un punter al primer element de l'array:

```
int vec[5]={4,5,2,8,12};  
cout << vec << endl; // Mostra l'adreça de memòria  
                        // del primer element de l'array  
cout << *vec << endl; // Mostra "4"
```

- Sempre apunta al primer element de l'array i no es pot modificar



## Punters i arrays (2/3)

- Els punters es poden usar com a accessos directes a components d'arrays:

```
int vec[20];  
int *pVec=vec; // Tots dos són punters a enter  
*pVec=58; // Equivalent a vec[0]=58;  
pVec=&(vec[7]);  
*pVec=117; // Equivalent a vec[7]=117;
```

## Punters i arrays (3/3)

- Els punters també poden usar-se per a crear *arrays dinàmics*
- Per a reservar memòria per a un array dinàmic cal usar claudàtors i especificar-ne la grandària
- Per a alliberar tota la memòria reservada és necessari també usar claudàtors (buits):

```
int *pv;  
pv=new int[10]; // Reserva memòria per a 10 enters  
pv[0]=585; // Accedim com en un array estàtic  
...  
delete [] pv; // Alliberem tota la memòria reservada
```

# Punters definits amb typedef

- Com vam veure en el *Tema 1*, es poden definir nous tipus de dades amb typedef:

```
typedef int enter;  
enter a,b; // Equivalent a int a,b;
```

- Per a facilitar la claredat en el codi poden definir-se els punters amb typedef:

```
typedef int *tPunterEnter;  
tPunterEnter pi; // Variable de tipus punter a enter  
                // No cal posar * en declarar-la
```

- Quan un punter referencia un registre, es pot usar l'operador `->` per a accedir-ne als camps:

```
struct TRegistre{
    char c;
    int i;
};
typedef TRegistre *TPunterRegistre;

TPunterRegistre pr;
pr=new TRegistre;
pr->c='a'; // Equivalent a (*pr).c='a';
pr->i=88;  // Equivalent a (*pr).i=88;
```

## Punters com a paràmetres de funcions (1/2)

- Un punter, com qualsevol altra variable, es pot passar com a paràmetre per valor o per referència a una funció:

```
void funcValor(int *p){ // Pas per valor
    ...
    p=NULL;
}
void funcReferencia(int *&p){ // Pas per referència
    ...
    p=NULL;
}
int main(){
    int i=0;
    int p=&i;
    funcValor(p);
    // p continua apuntant a i
    funcReferencia(p);
    // p val NULL
}
```

## Punters com a paràmetres de funcions (2/2)

- El mateix exemple d'abans usant typedef:

```
typedef int* tPunterEnter;
void funcValor(tPunterEnter p) {
    ...
    p=NULL;
}
void funcReferencia(tPunterEnter &p) {
    ...
    p=NULL;
}
int main() {
    int i=0;
    tPunterEnter p=&i;
    funcValor(p);
    funcReferencia(p);
}
```

## Errors comuns (1/2)

- No alliberar la memòria reservada dinàmicament:

```
void func(){  
    int *pEnter=new int;  
    *pEnter=8;  
    return; // Error! Falta delete pEnter;  
}
```

- Utilitzar un punter que no apunta a cap lloc:

```
int *pEnter;  
*pEnter=7; // Error! pEnter sense inicialitzar
```

## Errors comuns (2/2)

- Usar un punter després d'haver-ho alliberat:

```
int *p,*q;  
p=new int;  
...  
q=p;  
delete p;  
*q=7; // Error! La memòria ja s'havia alliberat
```

- Alliberar memòria no reservada amb new:

```
int *pEnter=&i;  
delete pEnter; // Error! Apunta a memòria estàtica
```



## Exercici 2

Donat el següent registre:

```
struct tClient{  
    char nom[32];  
    int edat;  
}tClient;
```

Realitzeu un programa que llixi un client (només un) d'un fitxer binari, l'emmagatzeme en memòria dinàmica usant un punter, imprimisca el contingut i finalment allibere la memòria reservada.

## Referències

---

## Referències (1/4)

- Les referències de C++ són com a punters però amb una sintaxi menys carregada (*sucre sintàctica*)
- No hi ha res que pugueu fer amb referències que no pugueu fer amb punters

```
int a=10;
int *b=&a; // Variable punter
*b=20;
cout << a << " " << *b; // Mostra "20 20"
int &c=a; // Variable referència
c=30;
cout << a << " " << c; // Mostra "30 30"
```

- En el codi anterior, `c` pot considerar-se com a un segon nom per a la variable `a`

## Referències (2/4)

- Les referències no poden ser `NULL`, sempre estan connectades a una dada
- Una vegada s'ha inicialitzat una referència, no es pot fer que es referisca a una posició de memòria diferent, la qual cosa si que és possible amb punters
- En crear una referència cal inicialitzar-la, però els punters es poden inicialitzar en qualsevol moment després de la declaració

## Referències (3/4)

- Les referències simplifiquen el codi de les funcions que tenen paràmetres passats per referència
- La següent funció usa punters per a passar dos paràmetres per referència:

```
void swap(int *x, int *y) {  
    int temp=*x;  
    *x=*y;  
    *y=temp;  
}  
  
int main() {  
    int a=10, b=20;  
    swap(&a, &b);  
    cout << a << " " << b; // Mostra "20 10"  
}
```

## Referències (4/4)

- La següent funció és equivalent a la anterior, però usa referències en lloc de punters:

```
void swap(int &x,int &y){  
    int temp=x;  
    x=y;  
    y=temp;  
}  
  
int main(){  
    int a=10,b=20;  
    swap(a,b);  
    cout << a << " " << b; // Mostra "20 10"  
}
```

- Aquesta és la sintaxi que hem estat usant en l'assignatura
- És més senzilla i còmoda que la de l'exemple anterior

# Implementació d'una pila

---

## Implementació d'una pila (1/6)

- Una *pila* és una estructura de dades molt usada en programació
- Consisteix en una llista d'elements
- Es pot afegir o eliminar elements a una pila amb una restricció: l'últim element afegit (*push*) serà el primer element a ser eliminat (*pop*)
- Exemples de pila en el món real:
  - En una pila de plast, el plat que està damunt i que acaba de ser empilat sempre serà el primer a ser desempilat
  - Els carrets de la compra en el supermercat, on sempre es trau l'últim que s'ha deixat



## Implementació d'una pila (2/6)

- Una pila pot implementar-se usant vectors de grandària fixa, però això limita el nombre d'elements que poden empilar-s'hi
- Podem solucionar-lo (parcialment) si usem un vector molt gran, però si empilem pocs elements estarem malgastant la memòria
- Usar punters per a implementar la pila permet usar només la memòria que necessiten cada volta
- Es pot implementar usant la idea de *llista enllaçada*
  - En empilar un nou element es reserva dinàmicament espai en memòria per a un registre
  - Aquest registre conté les dades a guardar i un punter a l'últim element de la pila
  - Tindrem un punter (anomenat `head`) que sempre apuntarà al cim de la pila

## Implementació d'una pila (3/6)

- En la següent implementació el punter `head` es passa com a paràmetre a les diferents funcions
- Es passa per referència quan alguna d'aquestes funcions pot canviar el punter perquè apunte a un altre registre
- Estructura d'un element de la pila:

```
struct Node{  
    int data; // Informació que volem guardar  
    struct Node *next; // Punter al següent element  
};
```

## Implementació d'una pila (4/6)

- Funcions per empilar (push) i desempilar (pop) elements:

```
void push(Node *&head,int newData) {  
    Node *newNode=new Node; // Reservem memòria  
    newNode->data=newData; // Guardem les dades  
    newNode->next=head; // Apuntem a l'últim node  
    head=newNode; // head apunta al nou node  
}  
  
void pop(Node *&head) {  
    Node *ptr;  
    if(head!=NULL) { // Ens assegurem que hi ha elements  
        ptr=head->next; // Segon element de la pila  
        delete head; // Esborrem el cim  
        head=ptr; // head apunta ara al segon element  
    }  
}
```

## Implementació d'una pila (5/6)

- Funcions per mostrar (display) i buidar (destroy) la pila:

```
void display(Node *head){
    Node *ptr;
    ptr=head;
    while(ptr!=NULL){ // Fins recòrrer tota la pila
        cout << ptr->data << " "; // Mostrem les dades
        ptr=ptr->next; // Passem al següent element
    }
}

void destroy(Node *&head){
    Node *ptr,*ptr2;
    ptr=head;
    while(ptr!=NULL){ // Fins recòrrer tota la pila
        ptr2=ptr; // Eliminem el node actual
        ptr=ptr->next; // Apuntem al següent element
        delete ptr2; // Esborrem el node actual
    }
    head=NULL; // La pila ja està buida
}
```

# Implementació d'una pila (6/6)

- Exemple de funció principal usant dues piles:

```
int main(){
    // Declarem i inicialitzem les dues piles
    Node *head1=NULL;
    Node *head2=NULL;
    // Afegim tres elements a la primera pila
    push(head1,3);
    push(head1,1);
    push(head1,7);
    display(head1); // Mostra "7"
    pop(head1); // Eliminem el cim
    display(head1); // Mostra "1"
    destroy(head1); // Buida la primera pila
    // Afegim un element a la segona pila
    push(head2,9);
    display(head2); // Mostra "9"
    destroy(head2); // Buidem la segona pila
}
```