@prog2ua

Unit 4: Dynamic memory

Programming 2

Degree in Computer Engineering
University of Alicante
2023-2024

Index

Memory layout

Static memory

- In `static memory` the size of the data is fixed and known before running the program
- The variables we have used so far are static:

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

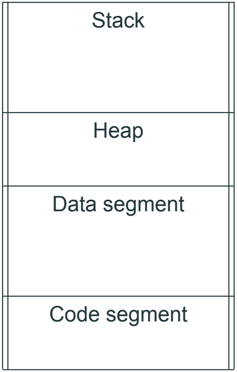
i	c	vf[0]	vf[1]	vf[2]
0		1.0	2.0	3.0
1000	1002	1004	1006	1008

Dynamic memory

- *Dynamic memory* allows storing large volumes of data, the exact size of which is unknown when implementing the program
- During program execution, the memory usage is adjusted to the needs at any given time
- In C++, dynamic memory can be implemented using pointers

Memory segments

- Different memory segments are used during the execution of a program:



- The *stack* stores the local data of a function: parameters passed by value and local variables
- The *heap* stores dynamic data allocated during the execution of the program
- The *data segment* stores global variables and static variables that are initialised by the programmer
- The *code segment* contains executable instructions (the code of the program)

Pointers

Definition and declaration

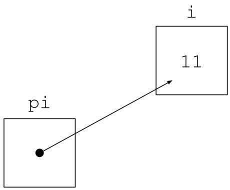
- A *pointer* stores the memory address where other data is located
- We say that the pointer “points” to that data
- Pointers are declared using the character `*`
- The data pointed by the pointer belongs to a specific type that must be indicated when the pointer is declared:

```
int *intPointer; // Integer pointer
char *charPointer; // Character pointer
int *intPointerArray[20]; // Array of integer pointers
double **doubleRealPointer; // Pointer to real pointer
```

Pointer operators (1/2)

- The `*` operator allows accessing the content of the variable pointed by the pointer
- The `&` operator allows obtaining the memory address in which a variable is stored:

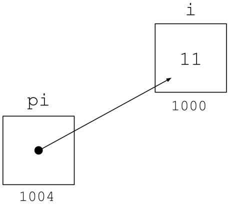
```
int i=3;
int *pi;
pi=&i; // pi contains the memory address of i
*pi = 11; // Content of pi is 11. Therefore, i = 11
```



Pointer operators (2/2)

- Assuming that `i` is at memory address `1000` and `pi` at `1004`:

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



Declaration with initialisation

- As any other variable, we can initialise a pointer at the time of its declaration:

```
int *pi=&i; // pi contains the memory address of i
```

- The `NULL` value can be used to indicate that a pointer does not point to any valid data:

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

- `NULL` is a constant with value zero. Since C++ 2011 standard, the constant `nullptr` can be also used as it represents zero as an address (pointer type)

Exercises

Exercise 1

Indicate what is the screen output of these code snippets:

```
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;
```

Usage of pointers

Memory allocation and deallocation (1/2)

- The `new` operator allows to dynamically allocate memory during program execution
- It returns the starting position of the allocated memory
- If there is not enough free memory, it returns `NULL`
- The address returned by `new` must be stored in a pointer:

```
double *pd;
pd=new double; // Allocates memory for a double
if(pd!=NULL){ // Check that memory was allocated
  *pd=4.75;
  cout << *pd << endl; // Prints "4.75"
}
```

Memory allocation and deallocation (2/2)

- The `delete` operator allows deallocating the memory allocated by `new`:

```
double *pd;
pd=new double; // Allocate memory
...
delete pd; // Free memory pointed by pd
pd=NULL; // Recommended if pd will be further used
```

- Whenever `new` is used to allocate memory, `delete` must be used to deallocate it
- Pointers can be reused after deallocation by using `new` again:

```
double *pd;
pd=new double; // Allocate memory
...
delete pd; // Free memory pointed by pd
pd=new double; // Allocate memory again with pd
...
```

Pointers and arrays (1/3)

- There is a close relationship between pointers and arrays
- An array variable is indeed a pointer to the first element of the array:

```
int vec[5]={4,5,2,8,12};
cout << vec << endl; // Prints the memory address
                     // of the first element of the array
cout << *vec << endl; // Prints "4"
```

- The array variable always points to the first element of the array and cannot be modified

Pointers and arrays (2/3)

- Pointers can be used as shortcuts to elements of an array:

```
int vec[20];
int *pVec=vec; // Both are integer pointers
*pVec=58; // Equivalent to vec[0]=58;
pVec=&(vec[7]);
*pVec=117; // Equivalent to vec[7]=117;
```

Pointers and arrays (3/3)

- Pointers can also be used to create dynamic *arrays*
- To allocate memory for a dynamic array, square brackets are used to specify the size
- To deallocate all the allocated memory it is also necessary to use (empty) brackets:

```
int *pv;
pv=new int[10]; // Allocated memory for 10 integers
pv[0]=585; // Access as with a static array
...
delete [] pv; // Deallocate all the allocated memory
```

Pointers defined with `typedef`

- As shown in *Unit 1*, new data types can be defined with `typedef`:

```
typedef int integer;
integer a,b; // Equivalent to int a,b;
```

- To get a clearer code, pointers can be defined with `typedef`:

```
typedef int *tIntegerPointer;
tIntegerPointer pi; // Integer pointer type
                    // Do not use * when declared
```

Pointers to structures

- When a pointer points to a `struct`, the `->` operator can be used to access its fields:

```
struct TStructure{
  char c;
  int i;
};
typedef TStructure *TStructurePointer;

TStructurePointer ps;
ps=new TStructure;
ps->c='a'; // Equivalent to (*ps).c='a';
ps->i=88; // Equivalent to (*ps).i=88;
```

Pointers as parameters to functions (1/2)

- A pointer, as any other variable, can be passed as a parameter by value or by reference to a function:

```
void funcValue(int *p){ // Parameter by value
  ...
  p=NULL;
}
void funcReference(int *&p){ // Parameter by reference
  ...
  p=NULL;
}
int main(){
  int i=0;
  int *p=&i;
  funcValue(p);
  // p stills pointing to i
  funcReference(p);
  // p is NULL
}
```

Pointers as parameters to functions (2/2)

- Previous example using `typedef`:

```
typedef int* tIntegerPointer;
void funcValue(tIntegerPointer p){
  ...
  p=NULL;
}
void funcReference(tIntegerPointer &p){
  ...
  p=NULL;
}
int main(){
  int i=0;
  tIntegerPointer p=&i;
  funcValue(p);
  funcReference(p);
}
```

Common errors (1/2)

- Not releasing dynamically allocated memory:

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

- Using a pointer that points to nowhere:

```
int *pInteger;
*pInteger=7; // Error! pInteger not initialised
```

Common errors (2/2)

- Using a pointer after deallocating memory:

```
int *p,*q;
p=new int;
...
q=p;
delete p;
*q=7; // Error! Memory already deallocated
```

- Deallocating memory not allocated with `new`:

```
int *pIntegetr=&i;
delete pInteger; // Error! Points to static memory
```

Exercises

Exercise 2

Given the following structure:

```
struct tClient{
  char name[32];
  int age;
}tClient;
```

Write a program for reading a client (only one) from a binary file. The program must allocate the structure in dynamic memory using a pointer, print its content and finally deallocate the memory.

References

References (1/4)

- C++ reference variables are actually pointers but with a lighter syntax (*syntactic sugar*)
- There is nothing we can do with references that cannot be done with pointers

```
int a=10;
int *b=&a; // Pointer variable
*b=20;
cout << a << " " << *b; // Prints "20 20"
int &c=a; // Reference variable
c=30;
cout << a << " " << c; // Prints "30 30"
```

- In the previous code, `c` can be considered as another name for `a`

References (2/4)

- References cannot be `NULL` and they are always connected to some data
- Once a reference is initialised, it cannot be changed to refer to another memory position, but pointers can
- A reference must be initialised when it is created, but pointers can be initialised at any time after their declaration

References (3/4)

- References simplify the code of functions that have parameters passed by reference
- The following function gets two parameters passed by reference using pointers:

```
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; // Prints "20 10"
}
```

References (4/4)

- The following function is equivalent to the previous one, but using references instead of pointers:

```
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; // Prints "20 10"
}
```

- This is the sintax we have been using in this course so far
- It is simpler and more user-friendly than the previous example

Implementation of a stack

Implementation of a stack (1/6)

- A *stack* is a data structure widely used in programming
- A stack is a list of elements
- Elements can be added or removed from the stack with one restriction: the last element added (*push*) is the element that will be first removed (*pop*)
- Examples of stacks in real life
    - A pile of stacked plates, where the plate on top is the first to be taken (popped)
    - Supermarket shopping trolleys, where you always pick up the last one left

Implementation of a stack (2/6)

- A stack can be implement by using fixed size arrays, but it will limit in the number of elements that can be pushed to the stack
- This issue could be (partially) solved by using a very large array, but if the number of elements in the stack is small, memory will be wasted
- A stack implementation using pointers will allow the memory requirements to grow or shrink depending on the current number of elements
- This implementation is based on the idea of *linked list*
  - When a new element is stacked, memory space is dynamically allocated for a register
  - This register contains the data to be saved and a pointer to the last element in the stack
  - There will always be a *head* pointer to the top of the stack

Implementation of a stack (3/6)

- In the following implementation the `head` pointer is passed as a parameter to the functions
- It is passed by reference when a function may change it to point to another register
- Structure of an element (node) in the stack:

```
struct Node{
  int data; // Information we want to store
  struct Node *next; // Pointer to the next node
};
```

Implementation of a stack (4/6)

- Functions for stacking (`push`) and unstacking (`pop`) elements:

```
void push(Node *&head,int newData){
  Node *newNode=new Node; // Memory allocation
  newNode->data=newData; // Data stored
  newNode->next=head; // Point to the last node
  head=newNode; // head points to the new node
}

void pop(Node *&head){
  Node *ptr;
  if(head!=NULL){ // Check that there are elements
    ptr=head->next; // Second element in the stack
    delete head; // Delete the top element
    head=ptr; // head points now to the second element
  }
}
```

Implementation of a stack (5/6)

- Functions to show (`display`) and empty (`destroy`) the stack:

```
void display(Node *head){
  Node *ptr;
  ptr=head;
  while(ptr!=NULL){ // Go through the whole stack
    cout << ptr->data << " "; // Show the data
    ptr=ptr->next; // Go to the next element
  }
}
void destroy(Node *&head){
  Node *ptr,*ptr2;
  ptr=head;
  while(ptr!=NULL){ // Until the whole stack is covered
    ptr2=ptr; // Remove the current node
    ptr=ptr->next; // Point to the next element
    delete ptr2; // Delete the current node
  }
  head=NULL; // The stack is empty
}
```

Implementation of a stack (6/6)

- Example of main function using two stacks:

```
int main(){
  // Declare and initialise both stacks
  Node *head1=NULL;
  Node *head2=NULL;
  // Add three elements tot he first stack
  push(head1,3);
  push(head1,1);
  push(head1,7);
  display(head1); // Print "7"
  pop(head1); // Delete the head
  display(head1); // Print "1"
  destroy(head1); // Empty the first stack
  // Add one element to the second stack
  push(head2,9);
  display(head2); // Print "9"
  destroy(head2); // Empty the second stack
}
```