C++ Programming

Pointers and Memory Management

M1 Math
Michail Lampis
michail.lampis@dauphine.fr

Dynamic Memory Allocation

- Data in your program lives (mostly) in two areas
  - The stack
  - The heap
- So far, we have been using only the stack

The stack

- The stack contains local variables of each block/function call
- The reason it is called a stack is that it grows/shrinks in one direction (up/down)
- The stack follows the execution path/function calls

The stack and recursive functions

- Recall the factorial function

```
int fact(int n){
    if(n<2) return 1;
    return n*fact(n-1);
}
```

- This function will allocate n different integers before it calculates anything...

The stack

- Good about the stack
  - Fast
  - Clean
- Bad about the stack
  - Data is “lost” when a function terminates (except things we return)
  - Often, memory needs and execution flow do not go in parallel

The heap

- The heap allows us to store data in arbitrary places in memory
- Idea: a function f can start building some object
  - The caller function/main program should have access to this object when f terminates
- Heap manipulation in C++ is done with
  - new
  - delete

An example program

- You have a function that calculates two magic numbers
- How to return them both to main?

```
int magic(){
    int n1 = 42; int n2 = 2112;
    return n1; //???
}
```

Solution: first attempt

- Why not return an array?

```
int [ ] magic(){
    int n[2] = {42 , 2112};
    return n;
}
```

Solution: first attempt

- Why not return an array?

```
int * magic(){
    int n[2] = {42 , 2112};
    return n;
}
```

Solution: first attempt

- Why not return an array?

```
int * magic(){
    int n[2] = {42 , 2112};
    return n;
}
…
int *p = magic();
cout << p[0] << p[1] << endl;
```

How about this?

```
//All in one function!
int n[2] = {42 , 2112};
return n;
int *p = n;
cout << p[0] << p[1] << endl;
```

Shrinking stack

- The problem with the first solution is that the memory for n is lost when the function terminates
- The return statement (correctly) return the address of n[0]
- But this address points to a place in memory which may not hold the value 42 any more!

new

- We can allocate heap memory using new
- Syntax: new T, returns a pointer to a (new) place in memory that can hold data of type T.
  - This is why you NEED to understand pointers

```
int * p = new int; //p is pointing to a new int
```

- Syntax: new T[size], returns a pointer to a (new) array of size elements of type T

```
int * p = new int[2]; //p is pointing to a new array
```

Back to our problem

```
int * magic(){
    //int n[2] = {42 , 2112};
    int * n = new int[2];
    n[0] = 42; n[1] = 2112;
    return n;
}
…
int *p = magic();
cout << p[0] << p[1] << endl;
```

Managing the heap

- The new operator gives us great power!
  - We can allocate memory whenever we need it
  - The memory stays around as long as we need it
- With great power comes great responsibility!
  - Stack memory is cleaned up automatically, but we must take care of cleaning up heap memory we don't need!
  - We also need to make sure we don't “lose” memory we have

What is the problem?

```
int * magic1(){...}
int * magic2(){...}
int * magic3(){...}
…
int *p = magic1();
cout << p[0] << p[1] << endl;
p = magic2();
p = magic3();
```

Memory leak

- The program of the previous slide is a classic example of a “Memory Leak”
- We allocate some memory with new, but then LOSE its address (by overwriting p)
- This makes this memory UNREACHABLE
- However, the memory is still allocated for our program...
- Do this enough, and your program will crash/slow down the computer (see eg Firefox!)

delete

- When we no longer need some memory we have allocated in the heap we can “free” it with the delete operator
- Syntax: delete p; where p is a pointer returned by new.

  int * p = new int; … ; delete p;
- Syntax: delete [ ] p; where p is a pointer returned by new [ ]

  int * p = new int[5]; … ; delete [ ] p;

Guess the output

```
int *p1, *p2; //Note: not int *p1, p2; why?
p1 = new int;
p2 = new int;
*p1 = 100;
*p2 = 200;
cout << *p1 << *p2 << endl;
```

Guess the output

```
int *p1, *p2; //Note: not int *p1, p2; why?
p1 = new int;
p2 = new int;
*p1 = 100;
*p2 = 200;
cout << *p1 << *p2 << endl;
delete p1;
p1 = p2;
cout << *p1 << *p2 << endl;
```

Guess the output

```
int *p1, *p2; //Note: not int *p1, p2; why?
p1 = new int;
p2 = new int;
*p1 = 100;
*p2 = 200;
cout << *p1 << *p2 << endl;
delete p1;
p1 = p2;
cout << *p1 << *p2 << endl;
*p2 = 300;
cout << *p1 << *p2 << endl;
//What's missing?
```

Find the bug

```
int *p1, *p2; //Note: not int *p1, p2; why?
p1 = new int;
p2 = new int;
*p1 = 100;
*p2 = 200;
cout << *p1 << *p2 << endl;

p1 = p2;
cout << *p1 << *p2 << endl;
*p1 = 300;
cout << *p1 << *p2 << endl;
```

Find the bug

```
int *p1, *p2, *p3;
p1 = new int;
p2 = new int;
p3 = p1;
*p1 = 100;
*p2 = 200;
cout << *p1 << *p2 << endl;
delete p1;
p1 = p2;
cout << *p1 << *p2 << endl;
*p1 = 300;
cout << *p1 << *p2 << endl;
delete p2;
```

Dangling pointers

- The last situation is a disaster waiting to happen (mild exaggeration)
- Problem: p3 is still pointing to the same area in memory as p1
- But this area in memory has now been deleted!
- Though p1 has changed value, you may still (accidentally try to access it through p3)

Find the bug

```
int *p1, *p2, *p3;
p1 = new int;
p2 = new int;
p3 = p1;
*p1 = 100;
*p2 = 200;
cout << *p1 << *p2 << endl;
//Clean everything up
delete p1;
delete p2;
delete p3;
```

No double-deletes!

- The previous program makes the dangling pointer problem worse
- Through p3, we try to delete **the same** area of memory twice!
- This is not allowed and will probably crash you program immediately

Heap operations summary

- Allocate memory with
  - new T; //returns T*
  - new T[ ]; //return T*
- Free memory with delete / delete [ ]
- Don't forget to delete what you allocate!
- Don't lose references to what you allocate!
- Don't keep pointing to what you deleted!
- No double deletes!

Back to Pointers/Arrays

- Warm-up exercise: write a function that checks if two **positive** int arrays are permutations of each other (they contain the same elements in perhaps different order)
- Task 1: define function prototype
- Task 2: algorithm?
- Task 3: program...
- (Spec: function should not change the two arrays)

One solution

```
bool isPerm(int a[ ], int b[ ], int size)
//Assume a,b have size size, only positive ints
{
    int i,j;
    for(i=0; i<size; i++){
        for(j=0;j<size;j++){
            if(a[i] == b[j]){
                b[j] = -1;
                break; //Only exits the inner loop
            }
        }
    }
    for(i=0; i<size; i++){
        if(b[i]!=-1) return false;
    }
    return true;
}
```

Make a copy of an array

- Given an array b of size size, construct an array c of the same size and the same elements (a copy)
- Recall, c = b does not work for arrays

Array copy

```
//Given int b[size]
int *c = new int [size]; //Allocate memory
for(int i=0; i<size; i++){
    c[i] = b[i];
}
… //do something with c
delete [ ] c; //Don't forget!
```

Array copy (old-fashioned)

```
//Given int *b, int size
int *c = new int [size]; //Allocate memory
while(size--){
    *c++ = *b++;
}
```

- Why does this work?? Pointer arithmetic
  - Note that this ruins the variable “size”
- This kind of copy is common in C code...

2-d Arrays

- C++ allows us to allocate 2-d arrays (or even higher dimensions) naturally:

```
int a[5][5];

int i;

a[2][3] = 7; //OK!

for(i=0; i<5; i++) cout << a[i] << endl;

//What does this mean?
```

2-d Arrays

- Recall the semantics of the [ ] operator
  - Is applied to expression of type “pointer to T”
  - Returns type T
- The expression a[2][3] can be read as (a[2])[3]
  - == apply [3] to the expression a[2]
  - → a[2] is an int *
- The way to view 2-d arrays in C++ is as arrays of pointers

2-d Arrays (stack)

```
int a[5][5];
int i;
a[2][3] = 7; //OK!
for(i=0; i<5; i++) cout << a[i] << endl;
//What does this mean?
```

- This prints (in hex):

  0x7fff97c34040

  0x7fff97c34054

  0x7fff97c34068

  0x7fff97c3407c

  0x7fff97c34090

- → Array is stored row-by-row by default...

Pointers to Pointers

- Pointer/Array equivalence is essential for 2-d Arrays
- Recall: in int a[5][5] the expression a[2] has type int *
  - What type does a have?
  - When [2] is applied to it we get int *
  - [ ] removes one *
  - → Answer: int ** (!!)

Pointers to Pointers

- Recall: we can define pointers to any valid type, including pointer types

```
int x = 2;
int *p = &x;
int **pp = &p; //What about int **pp = p?
cout << **pp; //Output?
```

Pointers to Pointers

```
int x = 2, y=3;
int *p = &x;
int **pp = &p;
*pp = &y; //?
cout << *p; //Output?
```

2-d Arrays on the heap

- We need ptrs to ptrs to dynamically allocate 2-d arrays

  ```
  int **a = new int[5][5];
  a[2][3] = 17; //OK
  ```

- Unfortunately, the first line doesn't work!
  - The second line is OK!
  - Recall ptr-array equivalence

2-d Arrays on the heap

- A dynamic 2-d arrays is an arrays of arrays
  - → it is an array of pointers!
- Step 1: define a to have appropriate type

```
int **a;
```

2-d Arrays on the heap

- Step 2: allocate space for the pointers that will hold each row

  int ** a = new int * [rows];

- Recall how the new operator works
  - (new Type [size])
  - This means: an array of size rows, each element of which has type (int *)

2-d Arrays on the heap

- Step 3: allocate each row individually

```
int ** a = new int * [rows];
for (int i=0; i<rows; i++)
    a[i] = new int [columns];
```

- Note: memory is not guaranteed to be allocated consecutively (as happens on stack)!
- When we are done, we need to delete all this...

2-d Arrays on the heap

- How to free a 2-d array allocated in this way
  - Step 1: free each row
  - Step 2: free arrays of row pointers

```
for(int i=0; i<rows; i++)
    delete [ ] a[i];
delete [ ] a;
```

Irregular 2-d Arrays

- Create a triangular array of “rows” rows:
  - Row r contains r+1 elements
  - The numbers 0,1,...,r+1

Triangular array

```
int ** a = new int * [rows];
for(int i=0; i<rows; i++){
    a[i] = new int [i+1];
    for(int j=0; j<i+1; j++)
        a[i][j] = j;
}
```

Const pointers

- We have seen the const keyword

  ```
  const int x = 5;
  x = 6; //Compiler error!
  ```

- Its semantics are a little complicated when pointers are involved

Const pointers

- We have seen the const keyword

```
const int x = 5;
x = 6; //Compiler error!
```

- Which of these is an error?

```
int i,j;
const int *p = &i;
p[0] = 2;
p = &j;
```

Const pointers

- p[0] = 2 is an error
  - The definition of p means it is pointing to a constant int
- To make p a “constant pointer” that always points to the same place use

  ```
  int * const p;
  ```
- More generally, semantics of * can be a little confusing with regards to precedence

Precedence of *

- The [ ] operator has higher priority

```
int *a[4];

int (*a)[4];

int *(*a)[4];
```

Precedence of *

- The [ ] operator has higher priority

  int *a[4]; //Pointer to 4-int array (Type: int *)

  int (*a)[4]; //Array of 4 int ptrs (Type: int **)

  int *(*a)[4]; //Pointer to array of 4 int ptrs (Type: int ***)

Precedence of *

- Also, the () operator has higher priority
  - Wait, is () an operator?

int * f (bool); //f is a function that takes bool returns int

int (* f) (bool); //f is pointer to function that...

Higher-order functions

- Usually associated with functional languages, can be used in C/C++

```
int add(int x, int y) { return x+y; }
int sub(int x, int y) { return x-y; }
int (*f)(int, int);
f = add;
cout << f(2,3) << endl;
f = sub;
cout << f(2,3) << endl;
```