C++ Programming

Pointers

M1 Math

Michail Lampis
michail.lampis@dauphine.fr

Pointers: General idea

- C++ includes many data types (int, char, double, ...)
- One **special** class of data types are **pointers**
- A variable x can be of type **pointer to T**, where T is another C++ data type
- This means that the value of x is the address in memory of some piece of data of type T

  (think T == int for a concrete example)

The computer's memory

- For the purposes of C++ programming, think of the computer's memory as a long tape, made up of blocks
- Each block in the tape can hold one byte (8 bits) of data
- Every block has an **address**
  - Think of the blocks in memory as being consecutively numbered from 1 to ...

Simple memory allocation

- So far, your C++ programs contain variable declarations

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

- The first line means "allocate 4 bytes of space, call this area in memory x"
- The second line means "allocate 20 bytes of space, call this area a"

Simple memory allocation

- So far, your C++ programs contain variable declarations

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

- Where will your program's memory be?
  - This depends on many runtime factors
  - In any case, the blocks corresponding to x, a, have some **address**

Pointer

- A pointer variable is a variable that is allowed to store such an address
- Pointer variables are declared using the * operator. Example:

  int * p; // p is a variable with type "Pointer to int"
- Generally <Type> * x; declares x to be a pointer to data of type <Type>.

Giving values to pointers

- Pointer values can be initialized to "point" to other variables

  ```
  int x = 5;
  int *p = &x;
  ```

- This means "p is a pointer to int. Its value is the **address** where x is stored"

Pointer values

- But what value does that give to a pointer?

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

- The first line prints 5
- The second line prints something unpredictable
  - It is the address where x (5) is stored when the program runs

How to read a pointer

- Reading an address directly is usually useless
- What we want to do is tell the program to "dereference" a pointer, that is, read the data the pointer "points to".

```
cout << p << endl;

cout << *p << endl;
```

- The second line means "the value written in the address stored in p"

Dereferencing

- Generally, the * operator can be applied to a pointer to type T
- The result is an expression of type T
- It is also an lvalue. Example:

```
int x = 5;
int *p = &x;
*p = 6;
cout << x << endl; //Will print 6
```

Why do we need this?

- Pointers are useful in C++ for several reasons
  - To make your life miserable (No. 1 source of bugs)
  - To allow pass-by-ref semantics (can also be done with references)
  - To allow **dynamic memory management**
- The last is by far the most important. We will see more later...

Pass-by-ref with pointers

- Recall:

```
void f(int x){ x++; }
int main() {
    int y = 5;
    f(y);
    cout << y << endl; //Prints 5, f has no effect
}
```

Pass-by-ref with pointers

- Recall:

```
void f(int *x){ *x++; }
int main() {
    int y = 5;
    f(&y); //Parameter must match declared type!
    cout << y << endl; //Prints 6
}
```

Pointers vs. Arrays

- In C/C++ pointers and arrays are **basically the same thing**

```
int a[5];
int *p;
p = &a[0];
p = a; //These two are THE SAME
```

Pointers vs. Arrays

- Generally, a variable of type "Array of T" is of the same type as a variable of type "(const) pointer to T"
- Only difference

```
int a[5];
int x;
a = &x; //Error! (though types are OK)
```

Consequence: pointer arithmetic

- We can pretend that a pointer is an array and vice-versa

```
int x=5;
int *p = &x;
p[0] = 6; //This writes on x
```

Consequence: pointer arithmetic

- We can pretend that a pointer is an array and vice-versa

```
int a[5];

int *p = a;

p[2] = 6; //This writes on a[2]

*(p+2) = 7; //This also writes on a[2]
```

The [] operator

- The semantics of the [] operator are:
  - It is applied on an expression p of type "pointer to T" and an integer i
  - It evaluates to

    *(e+i)
- This interpretation works for both arrays and pointers

Arrays/Pointers and functions

- Recall that functions can take arrays as parameters
  - But size cannot be declared!
  - Reason: in reality the parameter is a pointer

```
int f(int a[]);
int f(int *a); //These are the same!!
```

Complicated Pointers

- We can declare a variable to be of type "pointer to T", where T is a type
- This is also a type
- -> We can declare a variable to be a pointer to a pointer to int

```
int **pp; int *p; int x = 5;

p = &x;

pp = &p;

**pp = 6; //This writes on x!
```

Returning pointers

- Recall that functions cannot return arrays
- But they can return pointers, which is the same thing!

  ```
  int *f(int x); //OK!
  ```

Bad Example

```
int *create_array(int size)
{
int myarray[size];
int *p = myarray;
return p;
}
```

- This compiles OK, but is a **BAD IDEA**
- Reason: p is pointing to freed memory
- Solution: dynamic memory allocation

Dynamic memory allocation

- The main reason for using pointers
- Two operators
  - new T, returns a pointer to T. Allocates new memory to store data of type T
  - Delete p, frees the memory pointed to by pointer p
- These allow us to handle memory outside the "stack"
- More in next class...