Introduction to the C programming language
Pointers

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant’Anna – Pisa

February 11, 2010
Outline

1. Pointer syntax
2. Preprocessor
3. Arguments by reference
4. Pointers and arrays
5. Examples with strings
Outline

1. Pointer syntax
2. Preprocessor
3. Arguments by reference
4. Pointers and arrays
5. Examples with strings
A pointer is a special type of variable that can hold memory addresses.

Syntax

```c
char c;       // a char variable
char *pc;    // pointer to char variable
int i;       // an integer variable
int *pi;     // pointer to an int variable
double d;    // double variable
double *pd;  // pointer to a double variable
```

In the declaration phase, the * symbol denotes that the variable contains the address of a variable of the corresponding type.
A pointer variable may contain the address of another variable

```c
int i;
int *pi;
pi = &i;
```

The `&` operator is used to obtain the address of a variable. It is called the `reference` operator.

Warning: in C++ a reference is a different thing! Right now, pay attention to the meaning of this operator in C.
The reverse is called *indirection* operator and it is denoted by *

```c
int j;
j = *pi;  // get the value pointed by pi
*pi = 7;  // store a value in the address stored in pi
```

- In the first assignment, `j` is assigned the value present at the address pointed by `pi`.
- In the second assignment, the constant `7` is stored in the location contained in `pi`.
- `*pi` is an *indirection*, in the sense that it is the same as the variable whose address is in `pi`
Example

<table>
<thead>
<tr>
<th>i</th>
<th>5</th>
<th>23456</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>23460</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23464</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23468</td>
</tr>
<tr>
<td>pi</td>
<td></td>
<td>23472</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23476</td>
</tr>
<tr>
<td>j</td>
<td></td>
<td>23480</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23484</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23488</td>
</tr>
</tbody>
</table>
Example

- \( \pi \) is assigned the address of \( j \)
Example

\pi \text{ is assigned the address of } j
Example

- \texttt{pi} is assigned the address of \texttt{j}
- \texttt{j} is assigned the value of the variable pointed by \texttt{pi}
The commented line is a syntax error

- We are assigning a variable to a pointer
- The programmer probably forgot a & or a *
Outline

1. Pointer syntax
2. Preprocessor
3. Arguments by reference
4. Pointers and arrays
5. Examples with strings
The pre-processor

- It is time to look in more details at the *compilation* process
  - That is, translating from high level C code to low-level machine code
- The step are described below

```
C file -> pre-processor -> compiler -> object file -> linker -> Exe file
```

include file

C file
Pre-processor

- In this step, the input file is analyzed to process *preprocessor directives*
- A preprocessor directive starts with symbol `#`
  - Example are: `#include` and `#define`
- After this step, a (temporary) file is created that is then processed by the compiler
Directives

- With the **include** directive, a file is included in the current text file
  - In other words, it is copied and pasted in the place where the include directive is stated
- With the **define** directive, a symbol is defined
  - Whenever the preprocessor reads the symbol, it substitutes it with its definition
  - It is also possible to create macros
- To see the output of the pre-processor, run g++ with -E option (it will output on the screen)
  
  ```
g++ -E myfile.cpp
  ```
An example

```cpp
#include "myfile.hpp"
#include "yourfile.hpp"

int d;
int a=5;
int b=6;

int main()
{
    double c = PI;   // pi grego
d = MYCONST;      // a constant
    a = SUM(b,d);   // a macro
    return int(a);
}
```

```cpp
#define MYCONST 76
 extern int a, b;
#define SUM(x,y) x+y
```

```cpp
#define PI 3.14
 extern int d;
```
An example

preprocessor/main.cpp

```cpp
#include "myfile.hpp"
#include "yourfile.hpp"

int d;
int a=5;
int b=6;

int main()
{
    double c = PI; // pi grego
    d = MYCONST; // a constant
    a = SUM(b,d); // a macro
    return int(a);
}
```

preprocessor/myfile.hpp

```cpp
#define MYCONST 76
extern int a, b;
#define SUM(x,y) x+y
```

preprocessor/yourfile.hpp

```cpp
#define PI 3.14
extern int d;
```
Macros effects

Pay attention to macros, they can have bad effects

```c
#define SUM(x,y) x+y

int main()
{
    int a = 5, b = 6, c;

    c = 5 * SUM(a,b);
}
```

What is the value of variable \( c \)?
Some helpful “tricks”

- It is possible to define a macro for obtaining the literal name of a variable:

```
#define LIT_VAR(x) #x
```

A complete example: `pointers/point2.cpp`

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

#define LIT_VAR(a) #a
#define PVAR(y) cout << LIT_VAR(y) " = " " << y " << endl

int main()
{
    int d = 5;
    int x = 7;
    int *pi;

    pi = &x;

    PVAR(d);   PVAR(&d);
    PVAR(x);   PVAR(&x);
    PVAR(pi);  PVAR(*pi);

    d = *pi;

    PVAR(pi);  PVAR(x);
    PVAR(d);
}```
Outline

1. Pointer syntax
2. Preprocessor
3. Arguments by reference
4. Pointers and arrays
5. Examples with strings
Arguments of function

- In C, arguments are passed by value
  - With the exception of arrays
- However, we can use pointers to pass arguments by reference

```c
void swap(int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);
    PVAR(x);
    PVAR(y);
}
```
Outline

1. Pointer syntax
2. Preprocessor
3. Arguments by reference
4. Pointers and arrays
5. Examples with strings
Arrays

- An array denotes a set of consecutive locations in memory.
- In C, the name of an array is seen as a *constant pointer* to the first location.
- Therefore, it can be assigned to a pointer, and used as a pointer.

```c
int array[5] = {1, 2, 4, 6, 8};
int *p;
int d;

p = a;
d = *p;  // this expression has value 1
```
Pointer arithmetic

- It is possible to modify a pointer (i.e. the address) by incrementing/decrementing it

```c
int a[5] = {1, 2, 3, 4, 5};
int *p;
p = a // p now points to the first
    // element in the array

p++;
// p now points to the second
// element (a[1])

p+=2;
// p now points to the fourth
// element (a[3])
```

- Notice that in `p++`, `p` is incremented by 4 bytes, because `p` is a pointer to integers (and an integer is stored in 4 bytes)
Array and pointers

- Array are constant pointers, they cannot be modified

```c
int a[10];
int d;
int *p;
p = &d;
a = p; // compilation error, a cannot be modified
```

- Remember that the name of an array is not a variable, but rather an address!
- It can be used in the right side of an assignment expression, but not in the left side.
Equivalent syntax

A pointer can be used to access the elements of an array in different ways:

```c
int a[10];
int *p;
p = a;
*(p+1);  // equivalent to a[1]

int i;

*(p+i);  // equivalent to a[i]
p[i];    // this is a valid syntax
*(a+i);  // this is also valid
```

In other words, `a` and `p` are equivalent also from a syntactic point of view.
The number of bytes involved in a pointer operator depend on the pointer type.

An operation like \( p++ \) increments the pointer by:
- 1 byte if \( p \) is of type `char`
- 2 bytes if \( p \) is of type `float`
- 4 bytes if \( p \) is of type `int`

To obtain the size of a type, you can use the macro `sizeof()`

```c
int a, b;
char c;
double d;

a = sizeof(int);  // a is 4 after the assignment
a = sizeof(c);    // c is a char, so a is assigned 1
```

`sizeof()` must be resolved at compilation time (usually during preprocessing).
Pointer arithmetic - III

- Pointer arithmetic is also applied to user-defined types;

```cpp
#include <iostream>

using namespace std;

struct mystruct {
    int a;
    double b[5];
    char n[10];
};

int main() {
    struct mystruct array[10];

    cout << "size of mystruct: " << sizeof(mystruct) << endl;

    mystruct *p = array;

    cout << "p = " << p << endl;
    p++;
    cout << "p = " << p << endl;
}
```
In C/C++, the keyword `void` denotes something without a type
- For example, the return value of a function can be specified as `void`, to mean that we are not returning any value.

When we want to define a pointer that can point to a variable of any type, we specify it as a void pointer:

```c
void *p;
int d;

p = &d;
p++;  // error, cannot do arithmetic
// with a void pointer
```
Pointers and structures

- When using pointers with structures, it is possible to use a special syntax to access the fields.

```c
struct point2D {
    double x, y;
    int z;
};

point2D vertex;
point2D *pv; // pointer to the structure

pv = &vertex;
(*pv).x; // the following two expressions
p->x; // are equivalent
```

- Therefore, to access a field of the structure through a pointer, we can use the arrow notation \( p->x \).
Outline

1. Pointer syntax
2. Preprocessor
3. Arguments by reference
4. Pointers and arrays
5. Examples with strings
#include <iostream>
using namespace std;

int strcpy(char *p, char *q) {
    int c = 0;
    while (q[c] != 0) p[c] = q[c++];
    p[c] = 0;
    return c;
}

int main() {
    char name[] = "Lipari";
    char copy[10];

    strcpy(copy, name);

    cout << "name = " << name << endl;
    cout << "copy = " << copy << endl;
}
#include <iostream>
using namespace std;

int strcpy(char *p, char *q)
{
    int c = 0;
    while (*q != 0) {
        *(p++) = *(q++); c++;
    }
    *p = 0;
    return c;
}

int main()
{
    char name[] = "Lipari";
    char copy[10];

    strcpy(copy, name);

    cout << "name = " << name << endl;
    cout << "copy = " << copy << endl;
}