Fundamentals of Programming

Pointers

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

Scuola Superiore Sant'Anna – Pisa

February 29, 2012

Outline

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

Pointers

- A pointer is a special type of variable that can hold *memory addresses*
- Syntax

```
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

Syntax - cont.

- A pointer variable may contain the address of another variable

```
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.

Indirection

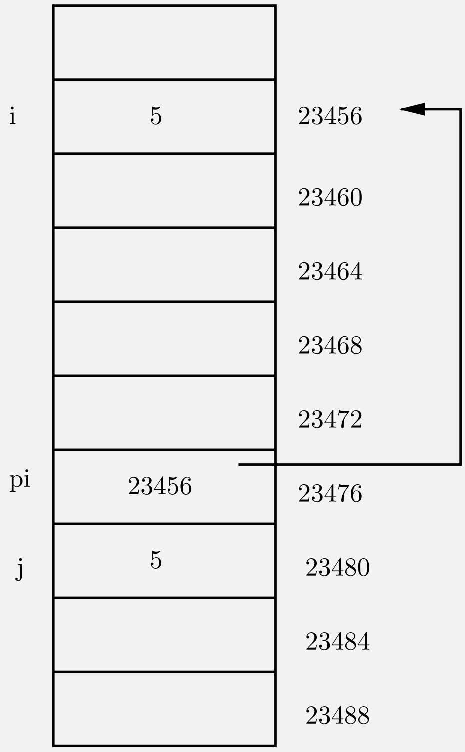
- The reverse is called *indirection* operator and it is denoted by `*`

```
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 is the same as the variable whose address is in `pi`

Example

- `pi` is assigned the address of `j`
- `j` is assigned the value of the variable pointed by `pi`



Examples

point1.c

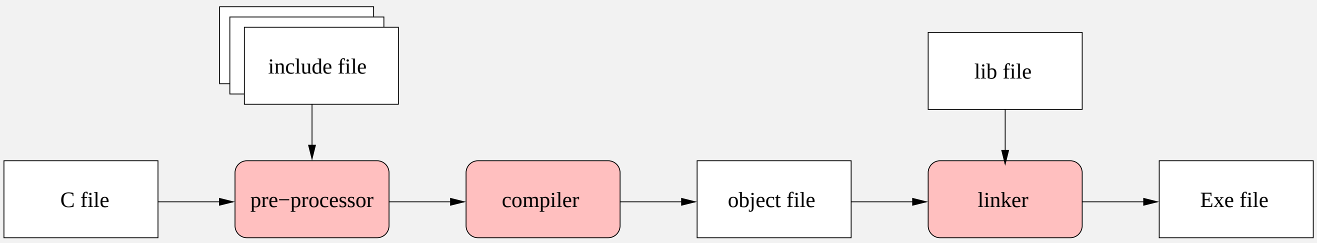
```
int main()
{
    int d = 5;
    int x = 7;
    int *pi;

    pi = &x;

    printf("%p\n", &x);
    printf("%p\n", &d);
    printf("%p\n", pi);

    printf("%d\n", *pi);

    //pi = d;  // compilation error

    d = *pi;

    printf("%p\n", pi);
    printf("%d\n", x);
    printf("%d\n", d);
}
```

The commented line is a syntax error

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

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



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 gcc with -E option (it will output on the screen)

```
gcc -E myfile.c
```

An example

main.c

```
#include "myfile.h"
#include "yourfile.h"

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;
}
```

myfile.h

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

yourfile.h

```
#define PI 3.14
extern int d;
```

main.c.post

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"
# 1 "myfile.h" 1


extern int a, b;
# 2 "main.c" 2
# 1 "yourfile.h" 1


extern int d;
# 3 "main.c" 2

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

int main()
{
    double c = 3.14;
    d = 76;
    a = b+d;
    return (int)a;
}
```


Macros effects

- Pay attention to macros, they can have bad effects

```
#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: point2.c

```
#include <stdio.h>

#define LIT_VAR(a) #a
#define PVAR(y) printf("%s = %d", LIT_VAR(y), y)
#define PPUN(y) printf("%s = %p", LIT_VAR(y), y)

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

    pi = &x;

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

    d = *pi;

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


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*

```
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);
}
```

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

```
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

```
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

```
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:

```
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 o view

Pointer arithmetic - II

- 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`**`()`

```
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;

struct.c

```
#include <stdio.h>

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

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

    printf("size of mystruct: %ld\n", sizeof(struct mystruct));

    struct mystruct *p = array;

    printf("p = %p\n", p);
    p++;
    printf("p = %p\n", p);
}
```

void pointers

- 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

```
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

```
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`

Copying a string (using arrays)

strcpy.c

```
#include <stdio.h>

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);

    printf("name = %s\n", name);
    printf("copy = %s\n", copy);
}
```


Copying a string, (using pointers)

strcpy2.c

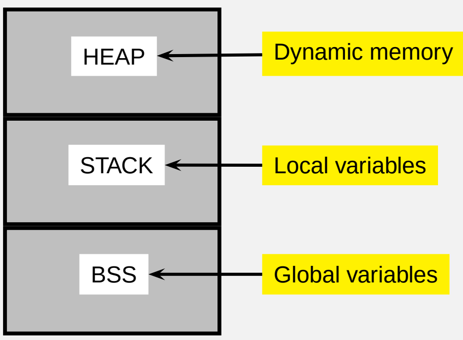
```
#include <stdio.h>

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);

    printf("name = %s\n", name);
    printf("copy = %s\n", copy);
}
```

Memory allocation

- We have discussed the rules for the lifetime and visibility of variables
  - **Global variables** are defined outside of any function. Their lifetime is the duration of the program: they are created when the program is loaded in memory, and deleted when the program exits
  - **Local variables** are defined inside functions or inside code blocks (delimited by curly braces { and }). Their lifetime is the execution of the block: they are created before the block starts executing, and destroyed when the block completes execution
- Global and local variables are in different **memory segments**, and are managed in different ways

Memory segments

- The main data segments of a program are shown below
- The BSS segment contains **global variables**. It is divided into two segments, one for initialised data (i.e. data that is initialised when declared), and non-initialised data.
  - The size of this segment is statically decided when the program is loaded in memory, and can never change during execution
- The STACK segment contains **local variables**
  - Its size is dynamic: it can grow or shrink, depending on how many local variables are in the current block



Example

- Here is an example:

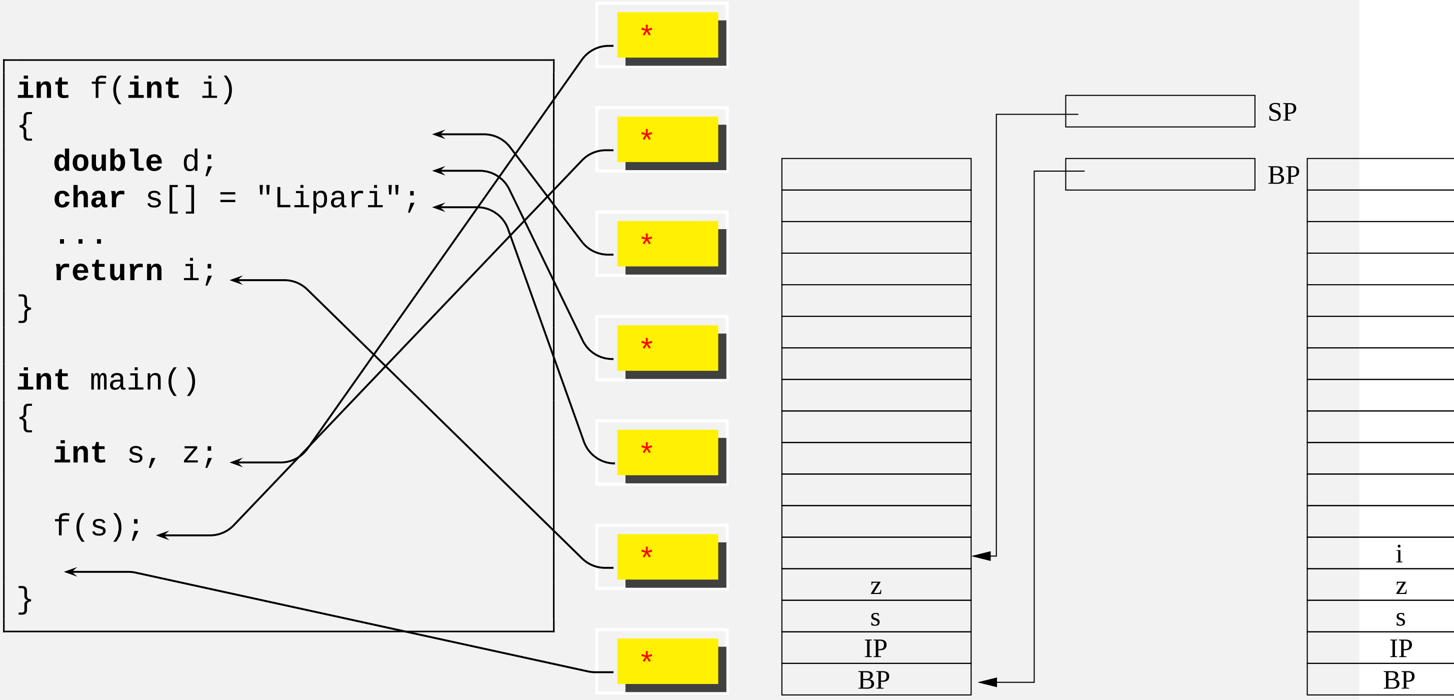
```
int a = 5; // initialised global data
int b;     // non initialised global data

int f(int i)    // i, d and s[] are local variables
{               // will be created on the stack when the
  double d;     // function f() is invoked
  char s[] = "Lipari";
  ...
}

int main()
{
  int s, z;     // local variables, are created on the stack
                // when the program starts

  f();          // here f() is invoked, so the stack for f() is created
}
```


Stack

- A Stack is a data structure with two operations
  - **push** data on top
  - **pop** data from top
- The stack is a LIFO (last-in-first-out) data structure
- The stack memory is managed in the same way as the data structure
- When a function is called, all parameters are **pushed** on to the stack, together with the local data
  - The set of function parameters, plus return address, plus local variables is called **Stack Frame** of the function

  - The CPU internally has two registers:
    - **SP** is a pointer to the top of the stack
    - **BP** is a pointer to the current *stack frame*
  - while the function is working, it uses **BP** to access local data
  - when the function finishes, all data is **popped** from the stack

Stack



Stack frame

- We will analyse the stack frame later in the course
- Right now let's observe the following things:
  - The stack frame for the previous function starts from parameter `i` and ends with the last character of `s[]`
  - The stack frame depends only on the number and types of parameters, and number and types of local variables
  - The stack frame can be computed by the compiler, that knows how to access local variables from their position on the stack
  - For example, to access parameter `i` in the previous example, the compiler takes the value of **BP** and subtracts 4 bytes: `BP - 4`
  - To access local variable d, the compiler uses **BP** and adds 4 (skipping **IP**).

Recursive functions

- It is possible to write functions that call themselves
- This is useful for some algorithms
- Consider the following function to compute the factorial of a number

```
int fact(int n) {
  int f;
  if (n <= 0) f = 0;
  if (n == 1) f = 1;
  else f = n * fact(n-1);
  return f;
}
```

- The function uses itself to compute the value of the factorial
- What happens on the stack?

Stack for recursive functions

```
int fact(int n) {
  int f;
  if (n <= 0) f = 0;
  if (n == 1) f = 1;
  else f = n * fact(n-1);
  return f;
}
```

- First stack frame
- Second stack frame
- Third stack frame
- Fourth stack frame
- `f` has been computed, return

	f
	IP
	BP
	n = 3
f	f
IP	IP
BP	BP
n = 4	n = 4

Stack frames

- Every time we call a function we generate a different stack frame
  - Every stack frame corresponds to an *instance* of the function
  - Every instance has its own variables, different from the other instances
- Stack frame is an essential tool of **any** programming language
- As we will see later, the stack frame is also essential to implement the operating system