Introduction to the C programming language

Dynamic memory

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

Scuola Superiore Sant'Anna – Pisa

February 11, 2010

Outline

Outline

Global, stack, heap

- All memory in a C/C++ program can be divided into 3 types:
  - **Global static memory**: this memory contains all code and all global variables. It is created by the system when the program starts executing, and it is destroyed when the program terminates. Therefore, it is *static*, i.e. it never change its size

Global, stack, heap

- All memory in a C/C++ program can be divided into 3 types:
  - **Global static memory**: this memory contains all code and all global variables. It is created by the system when the program starts executing, and it is destroyed when the program terminates. Therefore, it is *static*, i.e. it never change its size
  - **Stack memory**: this memory contains all local variables of any function. The memory for a local variable is *dynamically* created when the function is called, and it is destroyed (i.e. it is not available anymore) when the function returns.

Global, stack, heap

- All memory in a C/C++ program can be divided into 3 types:
  - **Global static memory**: this memory contains all code and all global variables. It is created by the system when the program starts executing, and it is destroyed when the program terminates. Therefore, it is *static*, i.e. it never change its size
  - **Stack memory**: this memory contains all local variables of any function. The memory for a local variable is *dynamically* created when the function is called, and it is destroyed (i.e. it is not available anymore) when the function returns.
  - **Heap memory**: this memory is created by the programmer by calling an appropriate function (malloc()), and it is released by calling another function (free). It is the user that manages this mamoery, so he must be careful in not making errors.

Examples

dynmemory/distance.cpp

```
struct point2D {
  double x, y;
  int z;
};
int a;
char name[10];
double vect[5];
point2D x1, x2;

double distance(point2D *p1, point2D *p2)
{
    return sqrt(pow((p1->x - p2->x),2) + pow((p1->y - p2->y),2));
}

int main()
{
  double dist;

  x1.x = 0; x1.y = 0;
  x2.x = 2; x2.y = 2;
  cout << "distance: " << distance(&x1, &x2) << endl;
}
```

Pointers and dynamic memory

- We must be careful when using pointers with dynamically allocated memory
- A pointer is just variable that stores an address, i.e. a number. However, there is no assumption on what the address contains.
- In particular, nobody guarantees that the pointer always points to a **valid** location throught the life of the program.
- This can be the source of many subtle errors.

Example

- Where is the mistake in the following code? dynmemory/stackerror.cpp

```
char *get_substring(char *str)
{
    char sub[100];
    int i = 0;

    while (str[i] != ' ' && str[i] != 0)
        sub[i] = str[i++];
    sub[i] = 0;
    return sub;
}

int main()
{
    char name[100] = "Giuseppe Lipari";
    char *p = get_substring(name);

    cout << "substring: " << p << endl;
```

What happened?

- In the previous example, sub is a local variable
  - therefore, the memory for the array is stack memory; it is created (*allocated*) when the function is called, and it is destroyed (*deallocated*) when the function finished
- Function get_substring returns the address of a local variable
  - This address is not valid when the function terminates
  - Therefore, p contains an *invalid address*
  - The address is likely to be reused by the program at the next function call (the following cout to pring on the terminal), and the memory is overwritten by other local variables.
  - Recent compilers raise a warning to the programmer: this is surely an error!

Outline

Heap memory

- To solve the previous error, we need to store the results of function get_substring in a set of memory location that is not deallocated after the function has terminated

Heap memory

- To solve the previous error, we need to store the results of function get_substring in a set of memory location that is not deallocated after the function has terminated
- We could use some static memory (i.e. global variables)

Heap memory

- To solve the previous error, we need to store the results of function get_substring in a set of memory location that is not deallocated after the function has terminated
- We could use some static memory (i.e. global variables)
- However, this is not very flexible, because it requires to know a-priori the amount of memory that is needed

Heap memory

- To solve the previous error, we need to store the results of function get_substring in a set of memory location that is not deallocated after the function has terminated
- We could use some static memory (i.e. global variables)
- However, this is not very flexible, because it requires to know a-priori the amount of memory that is needed
  - How many times we will call the function?

Heap memory

- To solve the previous error, we need to store the results of function get_substring in a set of memory location that is not deallocated after the function has terminated
- We could use some static memory (i.e. global variables)
- However, this is not very flexible, because it requires to know a-priori the amount of memory that is needed
    - How many times we will call the function?
    - How long can a substring be?

Heap memory

- To solve the previous error, we need to store the results of function get_substring in a set of memory location that is not deallocated after the function has terminated
- We could use some static memory (i.e. global variables)
- However, this is not very flexible, because it requires to know a-priori the amount of memory that is needed
  - How many times we will call the function?
  - How long can a substring be?
- The C/C++ standard library (cstdlib) provides functions to precisely allocate/deallocate memory

Malloc

- In C, the standard library provides functions malloc() and free ()

```
#include <cstdlib>

void *malloc(size_t s);
void free(void *p);
```

- The malloc() takes an integer parameters to specify the amount of bytes to allocate, and returns the address of the allocated memory block. From now on, the memory block is available for use until the corresponding free is called
- The free () takes a pointer to a previously allocated memory block and releases it. After the call, the address is not valid anymore.

malloc

- Notice that the malloc() returns a pointer to void. This is because the programmer that wrote this function and included it in the library did not know what the caller will want to do with the memory
- Thus, the programmer must *cast* the result of the malloc to the correct pointer type.

Solution to the previous example

dynmemory/stackcorrect.cpp

```
char *get_substring(char *str)
{
    char *sub;
    int i = 0;

    while (str[i] != ' ' && str[i] != 0) i++;
    sub = (char *)malloc((i+1)*sizeof(char)); // allocate mem
    strncpy(sub, str, i);   // copies i chars from str to sub
    return sub;
}

int main()
{
    char name[100] = "Giuseppe Lipari";
    char *p = get_substring(name);

    cout << "substring: " << p << endl;
    free(p);
}
```

Comments

- The programmer can allocate exactly the right amount of memory:
  - In the previous example, the programmer allocated exactly `i+1` bytes for the string
- However, the programmer must deal with this memory in the program
  - The stack memory is managed by the run-time system of the computer; it is automatically allocated when the function is called, and automatically deallocated when the function finishes
  - for this reason, local variables are also called *automatic variables*
  - heap memory, instead, must be managed by the programmer
  - heap memory management and pointers are the source of more than 90% of program bugs.

Outline

Memory leak

- One important thing to remember is to store the address of the allocated memory, so that we can later free it

dynmemory/leak.cpp

```
// function to swap the contents of two strings
void str_swap(char *p, char *q)
{
    char *tmp_p = (char *)malloc(strlen(p)+1);
    strcpy(tmp_p, p);
    strcpy(p, q);
    strcpy(q, tmp_p);
    return;
}

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

    str_swap(name, surn);
    cout << name << " " << surn << endl;
    str_swap(name, surn);
    cout << name << " " << surn << endl;
}
```

Problem

- In the previous example, the problem is that after the function str_swap() returns, the address of the allocated memory is lost (it was stored in a local variable), so we cannot free the memory anymore

Problem

- In the previous example, the problem is that after the function str_swap() returns, the address of the allocated memory is lost (it was stored in a local variable), so we cannot free the memory anymore
- Every time we call the str_swap() function, the total amount of allocate memory increases

Problem

- In the previous example, the problem is that after the function str_swap() returns, the address of the allocated memory is lost (it was stored in a local variable), so we cannot free the memory anymore
- Every time we call the str_swap() function, the total amount of allocate memory increases
- If the program is expected to run forever (for example a web server), at some point the computer memory will be over!

Problem

- In the previous example, the problem is that after the function str_swap() returns, the address of the allocated memory is lost (it was stored in a local variable), so we cannot free the memory anymore
- Every time we call the str_swap() function, the total amount of allocate memory increases
- If the program is expected to run forever (for example a web server), at some point the computer memory will be over!
- What happens is that you will see the program slow down a lot, until it crashes with an *out of memory* message

Problem

- In the previous example, the problem is that after the function str_swap() returns, the address of the allocated memory is lost (it was stored in a local variable), so we cannot free the memory anymore
- Every time we call the str_swap() function, the total amount of allocate memory increases
- If the program is expected to run forever (for example a web server), at some point the computer memory will be over!
- What happens is that you will see the program slow down a lot, until it crashes with an *out of memory* message
- This bug is called **memory leak**

Another example

- In the following example, we lose the reference to a memory block

```
int *p = (int *)malloc(10);
p[0] = 0;
for (i=1; i<10; i++) p[i] = p[i-1] + i;
...
p = (int *)malloc(20);
// we have lost reference to the previous memory block!
```

- The Java language has a feature called *garbage collector* that looks around for memory blocks that are not referenced by any pointer, and delete them.
- Garbage collection is an heavy task, so many languages like C/C++ do not have such a feature

Outline

Dynamic sizes

- As discussed before, one of the advantages of using heap memory is that we can use exactly as much memory as it is needed
  - Consider a program for representing 2D polygons
  - each polygon is represented by its vertexes in clockwise order
  - each vertex is represented by a struct point2D structure
  - therefore, the polygon can be represented by an array of vertexes
- Polygons can have any number of vertexes
  - If we only use static or automatic variables, we must allocate the memory at *compilation time*
  - therefore, our only choice is to decide a-priori the maximum size of the array (i.e. a maximum of vertexes) and the maximum number of polygons

Dynamic memory handling

- A first improvement consists in deciding dynamically the size of the array
- In the following program, we start writing a complete example of handling of 2D polygons
- First we write the prototype functions in a heaer file poly.h
- The implementation of such functions goes in a file called poly.cpp
- Finally, the usage goes in a file called main.cpp