Structures. C++ and Assembly

Ned Nedialkov

McMaster University
Canada

SE 3F03
March 2013
Outline

Structures

Accessing in assembly

Passing/returning structures

C++ and assembly

Classes

Inheritance and polymorphism
Structures

Example

```c
struct S {
    short int x; // 2 bytes
    int y;       // 4 bytes
    double z;   // 8 bytes
};
```

- Elements of a structure are arranged in memory in the same order as in the structure (ANSI C)
- The first element is at offset 0
- What is the offset of \( y \)? 2 or 4?
offsetof in stddef.h returns the offset of an element in a structure

What is the output of

```c
#include <stdio.h>
#include <stddef.h>

struct S{
    short int x; // 2 bytes
    int y; // 4 bytes
    double z; // 8 bytes
};

int main()
{
    printf("size of S: %ld\n", sizeof(struct S));
    printf("offset of x: %ld\n", offsetof(struct S, x));
    printf("offset of y: %ld\n", offsetof(struct S, y));
    printf("offset of z: %ld\n", offsetof(struct S, z));
    return 0;
}
```
What is the output of

```c
#include <stdio.h>
#include <stddef.h>

struct S{
    int x;         // 4 bytes
    long int y;    // 4 on 32-bit, 8 bytes on 64-bit machines
    double z;      // 8 bytes
};

int main()
{
    printf("size of S: %ld\n", sizeof(struct S));
    printf("offset of x: %ld\n", offsetof(struct S, x));
    printf("offset of y: %ld\n", offsetof(struct S, y));
    printf("offset of z: %ld\n", offsetof(struct S, z));
    return 0;
}
```
Accessing in assembly

- Example: set \( y \) to zero

```assembly
void zero_y(S *s_p)
%define y_offset 4
zero_y:
    enter 0,0
    mov eax, [ebp + 8]
    mov dword [eax + y_offset], 0
    leave
    ret
```

- Similar to accessing arrays
Passing/returning structures in C

- C allows passing structures by value
  - Not efficient, goes through the stack
  - Pass by reference/pointer
- C allows returning structures
  - Return values are in eax
  - How is it done?
  - Suppose `struct S foo()`, that is, returns a structure
  - The compiler may rewrite as

```c
struct S temp;
foo(&temp);
```
C++ and assembly

- Overloading: more than one function with the same name
- Which one to call?
- The compiler distinguishes by the number and type of arguments
  - Consider `void f(int x, int y)` and `void f(double x, int y)`
  - The compiler generates different names, e.g. `f__Fii` and `f__Fdi`, respectively
  - This is called name mangling
- Cannot overload by a return value
Calling C functions in C++

To call a C function in C++, use e.g.,

```cpp
extern "C" {
    int fun(int x, int y);
    double fun2(double x);
}
```

Without `extern "C"`... name mangling will occur
References

- We can pass parameters by reference
- More convenient than pointers
- Example: the output is 10

```c
#include <stdio.h>
void fun(int &x)
{
    x++;
}
int main()
{
    int y=9;
    fun(y);
    printf("%d\n", y);
    return 0;
}
```
Inline functions

- Disadvantages of macros
  - Consider
  - `#define SQR(x) (x*x)` and
  - `SQR(a+b)`
  - This expands to `(a+b*a+b)`

- Put as many brackets as you can, e.g.
  ```cpp
  #define SQR(x) ((x)*(x))
  ```

- This cannot happen with inline functions, e.g.,

```cpp
inline double sqr(double x)
{
    return x*x;
}
```
- If not inlined, normal function call
- If inlined, the code of the function is inserted where it is called
  - no stack frame
  - no parameter passing
Classes

- Essentially structures + functions
- Example

```cpp
class Simple {
public:
    Simple ();
    int get_data () const;
    void set_data ( int );
private:
    int data;
};
Simple :: Simple() { data = 0; }
int Simple::get_data() const { return data; }
void Simple :: set_data ( int x ) { data = x; }
```
The compiler may generate code corresponding to

```
struct Simple {
    int data;
};
void set_data (Simple *s, int x) {
    s->data = x;
}
int get_data (Simple *s) {
    return s->data;
}
```
Inheritance and polymorphism

Consider

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

class A {
public:
  virtual void print() = 0;
};
class B : public A {
public:
  void print() { cout << "Class B" << endl; }
};
class C : public A {
public:
  void print() { cout << "Class C" << endl; }
};
```
```cpp
int main()
{
    A *a1, *a2;
    a1 = new B();
    a2 = new C();
    a1->print();
    a2->print();
}
```

- **class A** is an abstract class
- **B** and **C** are derived from **A**
- Calling `print` on a pointer to **A** is essentially polymorphism
How are virtual functions implemented?

A class with virtual methods has a hidden pointer to a table with pointers to functions, vtable.

```cpp
a = new B; a points to an object of class B
a->print()
```

- a points to B
- in the virtual table of B, call `print`
- 3 times dereferencing