(1) C++ Abstractions

Nico Ludwig (@ersatzteilchen)

TOC

- Cited Literature:
  - Bruce Eckel, Thinking in C++ Vol I
  - Bjarne Stroustrup, The C++ Programming Language

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Separated Data that is not independent

- Let's assume following two functions dealing with operations on a date:
  - (The parameters: *d* for day, *m* for month and *y* for year.)

```
void PrintDate(int d, int m, int y) {
	std::cout<<d<<"."<<m<<"."<<y<<std::endl;
}

void AddDay(int* d, int* m, int* y) {
	// pass
}
```

- We can then use them like this:

```
// Three independent ints representing a single date:
int day = 17;
int month = 10;
int year = 2012;
PrintDate(day, month, year);
// >17.10.2012
// Add a day to the date. It's required to pass all the ints by pointer,
// because an added day could carry over to the next month or year.
AddDay(&day, &month, &year);
PrintDate(day, month, year); // The day-"part" has been modified.
// >18.10.2012
```

There are Problems with this Approach

- Yes! The presented solution works!
  - We end in a set of functions (and later also types).
  - Such a set to help implementing software is called an Application Programming Interface (API).
  - An API is a kind of collection of building blocks to create applications.

- But there are several very serious problems with our way of dealing with dates:
  - We have always to pass three separate ints to different functions.
  - We have to know that these separate ints belong together.
  - The "concept" of a date is completely hidden! – We have "just three ints".
  - So, after some time of developing you have to remember the concept once again!
  - => These are serious sources of difficult-to-track-down programming errors!

- The problem: we have to handle pieces of data that virtually belong together!
  - The "belonging together" defines the concept that we have to find.

A first Glimpse: User defined Types (UDTs)

- To solve the problem with separated data we'll introduce a User Defined Type (UDT).
  - For the time being we'll create and use a so called struct with the name *Date*
  - and belonging to functions operating on a *Date* object/instance/example.

```cpp
struct Date {
	int day;
	int month;
	int year;
};
```

```cpp
void PrintDate(Date date) {
	std::cout<<date.day<<"."<<date.month<<"."<<date.year<<std::endl;
}
void AddDay(Date* date) { /* pass */ }
```

- We can use instances of the UDT *Date* like this:
  - With the dot-notation the fields of a *Date* instance can be accessed.
  - The functions *PrintDate()*/*AddDay()* just accept *Date* instances as arguments.

```cpp
// Three independent ints are stored into one Date object/instance:
Date today;
today.day = 17; // The individual fields of a Date can be accessed w/ the dot-notation.
today.month = 10;
today.year = 2012;
PrintDate(today);
// >17.10.2012
AddDay(&today); // Add a day to the date. We only need to pass a single Date object by pointer.
PrintDate(today); // Date's day-field has been modified.
// >18.10.2012
```

6

- Arrays are also UDTs!
- The phrase "belonging to function" can be clearly explained now, e.g. the function *PrintDate()* depends on the bare presence of the definition of the UDT *Date*!
- Why do we need to pass a *Date* by pointer to the function *AddDay()*?
  - Because *AddDay()* needs to modify the passed *Date*.

Basic Features of Structs – Part I

- C/C++ structs allow defining UDTs.
  - A struct can be defined in a namespace (also the global namespace) or locally.
  - A struct contains a set of fields collecting a bunch of data making up a concept.
  - Each field needs to have a unique name in the struct.
  - The struct *Date* has the fields *day*, *month* and *year*, all of type int.
  - (UDT-definitions in C/C++ (e.g. structs) need to be terminated with a semicolon!)

```
struct Date {
        int day;
        int month;
        int year;
};
```

```
struct Person {
        const char* name;
        Date birthday;
        Person* superior;
};
```

  - The fields of a struct can be of arbitrary type.
    - Fields can be of fundamental type.
    - Fields can also be of another UDT! – See the field *birthday* in the struct *Person*.
    - Fields can even be of a pointer of the being-defined UDT! – See the field *superior* in *Person*.
    - But there can be no field of the being-defined UDT, because the type is counted as "incomplete" by the compiler:

```
struct Person {
        Date birthday;
        Person superior; // Invalid! Field has incomplete type 'Person'
};
```

  - The order of fields doesn't matter conceptually.
    - But the field-order matters as far layout/size of struct instances in memory is concerned.

- In C# and Java, the syntactic definitions of UDTs are not terminated by semicolons!

Basic Features of Structs – Part II

- A struct can generally be used like any fundamental type:
  - We can create objects incl. arrays of structs.
    - The terms object and instance (also example) of UDTs/structs are often used synonymously.
  - We can get the address of and we can have pointers to struct instances.
  - We can create instances of a struct on the stack or on the heap.

```
Date myDate; // Create a Date object on the stack.
myDate.day = 1; // Set and access a Date's fields w/ the dot notation...
myDate.month = 2;
myDate.year = 2012;
Date someDates[5]; // Create an array of five (uninitialized) Dates.
Date* pointerToDate = &myDate; // Get the address of a Date instance.
```

  - Functions can have struct objects as parameters and can return struct objects.

```
// A function accepting a Date object:
void PrintDay(Date date) {
	/* pass */
}
```

```
// A function returning a Date object:
Date CreateDate(int d, int m, int y) {
	Date date;
	date.day = d;
	date.month = m;
	date.year = y;
	return date;
}
```


- In a future lecture we'll learn how to create UDTs on the heap.

Basic Features of Structs – Part III

- C++ has a compact initializer syntax to create new struct instances on the stack.
  - Just initialize a new instance with a comma separated list of values put into braces:

```
// Create a Date object on the stack...
Date aDate;
aDate.day = 1;      // ...and set its fields w/ the dot
aDate.month = 2;    // notation individually.
aDate.year = 2012;
```

```
// Initialize a Date object with an initializer:
Date aDate = {17, 10, 2012};
```

```
// Ok! The field year will default to 0:
Date aDate2 = {10, 2012};
```

```
// Invalid! Too many initializer elements:
Date aDate3 = {11, 45, 17, 10, 2012};
```

C++11 – uniformed initializers

```
Date CreateDate(int d, int m, int y) {
    return {d, m, y};
}
PrintDate({17, 10, 2012});
int age{19}; // Also for builtin types.
```

  - The instance's fields will be initialized in the order of the list and in the order of their appearance in the struct definition.
  - => The order in the initializer and in the struct definition must match!

- Instances can also be created directly from a named/anonymous struct definition:

```
struct Point {  // Creates an object directly
    int x, y;   // from a struct definition.
} point;
point.x = 3;
point.y = 4;
```

```
struct {        // Creates an object directly
    int x, y;   // from an anonymous struct.
} point;
point.x = 3;
point.y = 4;
```


- The initializer syntax for structs can also be used for builtin types: `int age = {29};`

Basic Features of Structs – Part IV

- The size of a UDT can be retrieved with the sizeof operator.
  - The size of a UDT is not necessarily the sum of the sizes of its fields.
    - Because of gaps! – More to come in short...

```
struct Date { // 3 x 4B
	int day;
	int month;
	int year;
};
```

```
std::cout<<sizeof(Date)<<std::endl;
// >12
```

- In C/C++ we'll often have to deal with a pointer to a UDT's instance, e.g.:

```
Date aDate = {17, 10, 2012};
Date* pointerToDate = &aDate; // Get the address of a Date instance.
```

- UDT-pointers are used very often, C/C++ provide special means to work with them:

```
int day = (*pointerToDate).day; // Dereference pointerToDate and read day w/ the dot.
int theSameDay = pointerToDate->day; // Do the same with the arrow operator.
pointerToDate->month = 11; // Arrow: Dereference pointerToDate and write month.
++pointerToDate->year;  // Arrow: Dereference pointerToDate and increment year.
```

- The arrow-notation (->) is used very often in C/C++, we have to understand it!

`(*pointerToDate).day` ⟷ `pointerToDate->day`

UDTs and Instances

- The idea of a UDT is the invention of a new type, composed of other types.
  - UDTs can be composed of fundamental types and/or composed of other UDTs.
  - => UDTs make APIs really powerful, simple to use and simple to document.

- In general programming terms UDTs are often called record-types.
  - In C++, record-types can be defined with structs, classes and unions.
  - An API consisting of free functions and record-types is a record-oriented API.

- UDTs and instances:
  - A UDT is like a blue print of a prototypical object.
    - E.g. like the fundamental type int is a blue print.
  - An object is a concrete instance of a UDT that consumes memory during run time.
    - E.g. like a variable *i* or the literal 42 represent instances or objects of the type int.
  - Consider the UDT *Coordinates*:

```cpp
// Blue print:
struct Coordinates {
	int x;
	int y;
};
```

```cpp
// A concrete instance of the blue print
Coordinates location = {15, 20};
// that consumes memory:
std::size_t size = sizeof(location);
std::cout<<size<<std::endl;
// >8
```

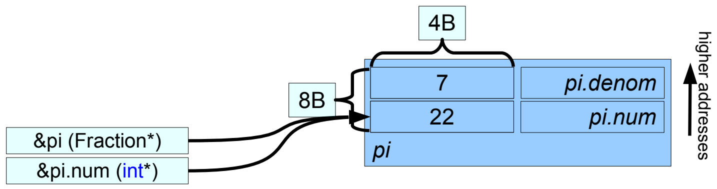
C++11 – get the size of a field

```cpp
std::size_t size = sizeof(Coordinates::x);
```

Memory Representation of Structs – PODs

```
struct Fraction {
    int num;
    int denom;
};
Fraction pi;
```

```
pi.num = 22;
pi.denom = 7;
```



- UDTs only containing fields are often called plain old datatypes (PODs).
  - PODs have special features concerning memory we're going to analyze now.

- Similar to arrays, a struct's fields reside in memory as a sequential block.
  - The *Fraction pi* is created on the stack, i.e. all its fields are stored on the stack as well.
  - The first field of *pi* (*pi.num*) resides on the lowest address, next fields on higher addresses.
  - The address of the struct instance (*&pi*) is the same as of its first field (*&pi.num*).
    - Without context, the address of *pi* could be the address of a *Fraction* instance or an int.
  - Assigning 22 to *pi.num* does really assign to the offset of 0B to *pi*'s base address.
    - As *pi.denom* is 4B above *pi.num*'s address, the 7 is stored to *&pi* plus an offset of 4B.

- Definitions: *num* := numerator, *denom* := denominator
- A very stringent definition of PODs: A POD contains only fields of fundamental type; this makes a PODs self-contained. Self-contained PODs have no dependencies to other UDTs that makes using them very straight forward (this is esp. relevant for tool support).
- Meanwhile C++11 provides a less stringent definition of PODs.
- The necessity of a struct's fields being arranged at sequential addresses (incl. the first field having the lowest address, representing the address of a struct instance) doesn't hold true, if a UDT applies access specifiers (private, protected and public).

Memory Representation of Structs – Natural Alignment

- Many CPU architectures need objects of fundamental type to be naturally aligned.
  - This means that the object's address must be a multiple of the object's size.
  - So, the natural alignment (n.a.) of a type is the multiplicator of its valid addresses.

- In a UDT, the field with the largest n.a. type sets the n.a. of the whole UDT.
  - This is valid for record-types (e.g. structs), arrays and arrays of struct instances.
  - The resulting n.a. is called the n.a. of the UDT.
  - Here some examples (LLVM GCC 4.2, 64b Intel):

```
struct A {
    char c;
};
sizeof(A): 1
Alignment: 1
```

```
struct B {
    char c;
    char ca[3];
};
sizeof(B): 4
Alignment: 1
```

```
struct C {
    double d;
    char* pc;
    int i;
};
sizeof(C): 24
Alignment: 8
```

C++11 – get the alignment of a type

```
std::size_t alignment = alignof(A);
std::cout<<alignment<<std::endl;
// >1
```

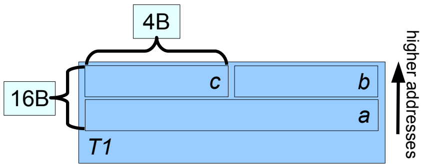
C++11 – control the alignment of a type

```
alignas(double) char array[sizeof(double)];
```

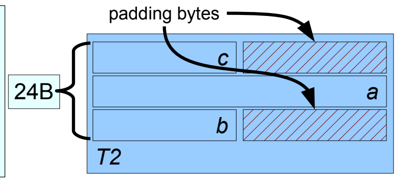
- On a closer look, the size of an instance of *C* seems to be too large! Let's understand why...

Memory Representation of Structs – Layout

```
struct T1 {
    double a;
    int b;
    int c;
};
sizeof(T1): 16
Alignment: 8
```




```
struct T2 {
    int b;
    double a;
    int c;
};
sizeof(T2): 24
Alignment: 8
```



- Field order of *T1*: the memory layout is sequential, rectangular and contiguous.
  - The field with the largest n.a. (the double *a*) controls *T1*'s n.a. (8B).
  - Two int-fields can reside at word boundaries and will be packed (2x4B "under" *a*'s 8B).
  - The packed fields lead to a size of 16B for a *T1* instance.
- Field order of *T2*: the memory layout is sequential, rectangular but not contiguous.
  - The field with the largest n.a. (the double *a*) controls *T2*'s n.a. (also 8B).
  - The n.a. of the fields *b* and *c* (ints at word boundaries) causes gaps, this is called padding.
  - This field order leads to a size of 24B for a *T2* instance.
- => The order of fields can influence a UDT's size.

- In .NET the memory representation of structs (value types) can be controlled with the custom attribute *StructLayout*.

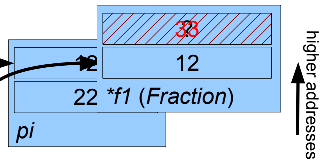
Memory Representation of Structs – Cast Contact Lenses

```
struct Fraction {
    int num;
    int denom;
};
```

```
Fraction pi;
pi.num = 22;
pi.denom = 7;
```

&pi.denom (int*)

```
Fraction* f1 = reinterpret_cast<Fraction*>(&pi.denom);
```



- The *Fraction* object *pi* is created on the stack.
- The &*pi.denom* is really pointing to an int, after the cast it is seen as &*Fraction*!
  - In other words: &*pi.denom* is seen as an address of another complete *Fraction* (**f1*).
  - The &*pi.denom* is now seen as the *num* field of the overlapping *Fraction* instance (**f1*)!
  - Writing *f1->num* will really write to *pi.denom* due to the overlap (&*pi.denom* + 0B)

```
f1->num = 12;
```

  - Writing *f1->denom* will really write to foreign memory (&*pi.denom* + 4B)!

```
f1->denom = 33;
```

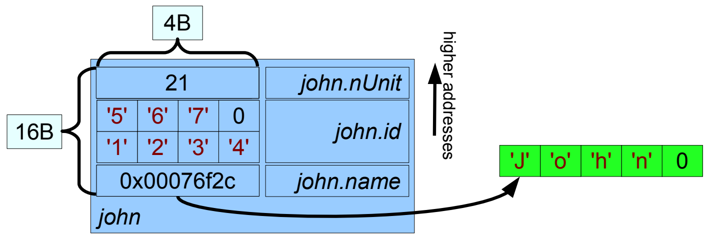
  - We do not own the memory at &*pi.denom* + 4B!

- Pointer arithmetics only work with arrays. This feature is exploited for STL operations.

Memory Representation of Structs – Pointer-Fields to Heap

```
struct Student {
    char* name;
    char id[8];
    int nUnit;
};

Student john;
```




```
const char* name = "John";
john.name = static_cast<char*>(std::malloc(sizeof(char) * (1 + std::strlen(name))));
std::strcpy(john.name, name);
```

```
std::strcpy(john.id, "1234567");
```

```
john.nUnit = 21;
```

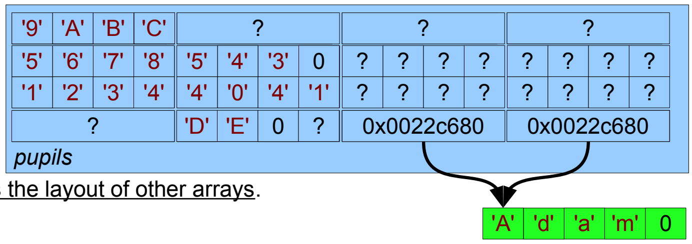
- Memory facts of *john*:
  - The address of *john.name* (as it is the first field) is also the address of *john*.
  - The content of *john.name* is not contained in *john*, *john.name* points to a cstring in the heap.
  - The field *john.id* occupies 8B, stores a cstring of seven chars and is contained in *john*.
  - The field *john.nUnit* stores an int that is also contained in *john*.

16

- In which kind of memory is *jon.name* stored? And where is the memory, to which *jon.name* is pointing to?
  - The field *jon.name* is special, as it only contains a pointer to the "real" cstring (the pointer (the "cord") resides on the stack). The memory occupied by this cstring (the "balloon") resides somewhere in the heap. The address stored in *jon.name* is also the address of the cstring's first char.
  - It also means that the memory at *jon.name* needs to be freed after we're done with *jon*!
- Why has the char* a size of 4B?
  - The size of all pointer types is the size of the architecture's address space. On a 32-bit system: 4 = sizeof(int*), on a 64-bit system: 8 = sizeof(int*), etc.
- The address of *jon* could be a *Student** or char**.

Memory Representation of Structs – Arrays of Structs

```
Student pupils[4];
pupils[0].nUnit = 21;
pupils[2].name = strdup("Adam");
```



- The memory layout of a struct-array is the same as the layout of other arrays.

```
pupils[3].name = pupils[2].name;
```

- Assigning a pointer to another pointer makes both point to the same memory location.

```
std::strcpy(pupils[1].id, "4041543");
```

- Assigning a cstring leads to occupation of adjacent bytes incl. the 0 termination.

```
std::strcpy(pupils[0].id, "123456789ABCDE"); // Oops!
```

- Assigning a cstring that overlaps reserved memory overwrites adjacent bytes!
  - This is "ok" in this case: we overwrite only bytes within the array. We are the owner!
  - But the values in the affected *Student* instances (*pupils[0]* and *pupils[1]*) were corrupted.

17

- In which kind of memory is the array *pupils* stored?
- Why is the function *strdup()* not prefixed with *std*?
  - This function is not in the namespace *std*, because it is no C++ standard function. But it is defined in the POSIX "standard".

```
// No standard C/C++, but defined in POSIX:
char* strdup (const char* str) {
	char* dynStr = str
		? static_cast<char*>(std::malloc((1 + std::strlen(str)) * sizeof(char)))
		: 0;
	return dynStr ? std::strcpy(dynStr, str) : 0;
}
```

- What is POSIX?
  - The Portable Operating System Interface or Portable Operating System Interface based on UNIX (POSIX) is an IEEE- and Open Group-standard for a platform-independent API to operate with the OS.
- If two pointers point to the same location in the heap, the memory must only be freed from one of those pointers!

Thank you!