Hi, my name is Yuriy and I would like to present the work we did with Dr. Dos Reis and Dr. Stroustrup on implementing open and efficient type switch for C++. 
Often times we have to deal with heterogeneous object graphs of many different node kinds. Traditional functional programming way to do this is pattern matching over algebraic data types, which is known to be simple, elegant and fast - properties that stem from the fact that algebraic data types are closed, while their variants are disjoint. Traditional object-oriented approach is to use visitor design pattern, which is known to be complicated, verbose, hard to teach and use. On the positive side visitors are reasonably fast, and bring function extensibility while maintaining a limited form of class extensibility.

Our type switch combines the benefits of both, while avoiding their limitations. In particular it is as simple and elegant as pattern matching, open to new sub-classes and does not limit the set of case clauses.
By type-switch we mean a multi-way branch statement on object’s dynamic type, which hypothetically can look as following in a program given a compiler implementation.

By an open type switch we will mean any type switch implementation that does not make closed world assumption and thus allows adding new functions and classes modularly, without changes to existing code, including at run-time in the presence of dynamic linking.
Consider an interpreter (or calculator) of a simple expression language represented by this grammar.

In functional languages, a term in the expression language will typically be represented with a recursive algebraic data type with variants corresponding to production rules. Algebraic data types are closed (no variants can be added later) and disjoint (value belongs to only one variant). Because of this, various operations on expression terms (like evaluation, printing or transformation) can be implemented by simple case analysis on the variants and the compiler can check the exhaustiveness of it.

Notice that in functional languages it is easy to add new operations on expression terms, but adding new variants (i.e. productions here) requires modification of the algebraic data type and all the existing functions on it.
In object-oriented languages, a term in the expression language will usually be represented with an abstract base class with derived classes representing the corresponding production rules. We then introduce a virtual function per each operation into the base class. The function is then overridden by each derived class to implement the logic of the operation on that specific variant.

Unlike algebraic data types, classes are extensible and hierarchical. Extensibility means that new sub-classes, which correspond to variants of algebraic data types for this example, can be added later, including at run-time in the presence of dynamic linking. The fact that classes are hierarchical means that a given value may belong to numerous classes.

The situation is thus dual for object-oriented languages – it is easy to add new variants, but adding new operations require modification of the base class and all the existing variants. This duality is known as the “expression problem” and have been extensively studied in the literature.
Visitor design pattern was an attempt to combine the benefits of both worlds and it certainly had few advantages:

- a solution to extensibility of functions
- an overhead of only 2 virtual function calls
- and being a library solution

Unfortunately it is intrusive (because we have to inject some virtual functions), specific to class hierarchy, introduces control inversion and, most importantly, hinders extensibility of classes by fixing the set of variants. You can see on the left that it requires a lot of boilerplate code to be written and is surprisingly hard to teach to students.
The following code shows the complete evaluator of a term in our expression language, written in OCaml. Other functional languages have very similar syntax, which functional programming community finds intuitive and capturing only the essence of computation.
Experimental C++ Notation

C++ with type switch library (Mach7)

```cpp
int eval(Expr& e) {
    Match(e)
    Case(Value& x) return x.value;
    Case(Plus& x) return eval(x.e1) + eval(x.e2);
    Case(Minus& x) return eval(x.e1) - eval(x.e2);
    Case(Times& x) return eval(x.e1) * eval(x.e2);
    Case(Divide& x) return eval(x.e1) / eval(x.e2);
    EndMatch
}
```

Logically equivalent to functional programming notation
We could improve the syntax if we modified a compiler

Compare it to the logically equivalent code in C++ based on our Mach7 library. The syntax could be improved further if we modified a compiler.
Unlike the visitor design pattern, our solution does combine the benefits of both worlds! The solution we offer retains all the advantages of the visitor design pattern, but also gets rid of its disadvantages. In particular, our solution is:

- much more intuitive, concise and direct and is thus easier to teach and use
- it is non-intrusive and can be retroactively applied to any polymorphic classes
- it avoids control inversion typical of visitors
- provides for extensibility of classes and functions
- in its current form it is still a library solution
- it is general in that it can be used for other object-oriented languages and fully supports multiple inheritance of C++

Unlike the visitor design pattern, our solution does combine the benefits of both worlds! The solution we offer retains all the advantages of the visitor design pattern, but also gets rid of its disadvantages. In particular, our solution is:

- Easy to teach and use
  - Non-intrusive
  - No control inversion
- Extensible
  - Functions and classes
- Fast
- General
  - Not specific to hierarchy
- Library solution
  - We use several industrial compilers for experimentation and measurement

- Slower on the 1st call for each dynamic type passed
Existing approaches to type switching are either efficient or open, but not both. Efficient approaches rely on closed world assumption and then implement the switch with jump table over small consecutive integers as demonstrated in chart with “C Switch on integers”.

Open approaches rely on constant-time subtype tests combined with decision trees, as demonstrated in chart with “Fast Dynamic Cast”, “Cohen’s Algorithm” and “Binary Matrix”. Unfortunately most of the known constant-time type tests are not suitable for repeated multiple inheritance and require non-trivial computations at load time in order to keep the solution open. Independently of these, for Match statements with many case clauses the time to uncover the type is not constant and increases with the case number because of the underlying decision tree. It quickly surpasses the time of known workaround for even small number of case clauses.

Our solution provides amortised constant time discovery that is faster than visitor’s constant time discovery.
The solution we offer relies on the following key points:

• A mechanism for mapping values to executions paths
• A property of the C++ object model that lets us distinguish particular sub-objects within an object
• Non random structure of the v-table pointers that helps us make the solution efficient
Instead of starting with a known efficient solution and trying to make it open, we start with a known open solution and try to make it efficient. The cascading-if at the bottom implements the first-fit semantics of the type switch on the top in truly open fashion. This happens because the implementation of dynamic_cast operator is open to class extensions.

Unfortunately the solution takes thousands of cycles for large case clauses, even when optimized with a decision tree, which is disproportionately larger than only 20 cycles used by the visitor design pattern. This happens because on one side we repetitively check the same dynamic type, while on the other – we do not learn anything from it.
Consider a slightly more general problem: execute the first statement, whose guard predicate is true. Providing the predicates do not have side effects, the same clause will be executed for the same value of \( x \). We can thus memoize the clause enabled for a particular value and re-execute it directly on subsequent runs with the same \( x \).

The actual control structure, which we call “Memoization Device” is automatically generated by macros and represents an interleaving of a cascading-if statement with the switch-statement. The mapping of values to case clauses is stored in a global hash table, which default initializes elements to 0 upon first access. Values not yet in the hash table will fall into the default clause and trigger the sequential execution of the cascading if-statement. The first predicate that evaluates to true will save its clause number in the corresponding element of the hash table for subsequent calls.

The obvious drawback of this solution is the size of the hash table that will be proportional to the number of values seen, however often times the values can be grouped into equivalence classes with values in the same class rendering the same outcomes for all the predicates, allowing us to store the equivalence class instead.
We use sub-objects as an equivalence class that does not change the outcome of dynamic_cast and map them to a pair of offset between the source and target sub-objects and the jump target. We use vtbl-pointers to uniquely identify sub-objects in our implementation.

On first entry (sequential execution of cascading if) we memoize offset and jump target

On subsequent runs we simply jump directly to memoized target and adjust this-pointer with memoized offset
The intermediate solution works, but as is it is still about 50% slower than visitors. We use the structure of v-table pointers to narrow the gap.

We look at a set of v-table pointers that come through a given Match-statement and identify the bits they are different in. The idea is to use a cache of $2^k$ entries and a simple hashing function that gets rid of common bits on the right, however the size of such cache can be too large to justify the use. We thus fix $k$ to be not larger than twice the closest power of 2 for the number of actual v-table pointers seen. We then let $l$ be chosen in such a way as to minimize the probability of conflict (which is the same as to maximize the number of occupied cache entries).

For each such set $V$ we:

- Use cache of $2^k$ entries addressed by:

$$H_{kl}(v) = \frac{v}{2^l} \mod 2^k$$

We choose $k$ and $l$ to minimize the number of conflicts in cache.
We now compare the performance of our approach to the performance of visitor design pattern. Numbers in blue indicate cases when type switch was faster than visitors while numbers in red indicate cases when visitors were faster than type switch. The 3 usage scenarios represent various call patterns that we found in different applications. As can be seen on the right, our library allows users to benefit from cases when the corresponding class hierarchy is known to be closed.

The following chart visually represents the speedups for the most typical scenario of case analysis on all derived classes.
Presence of forwarding indicates a common technique in visitor design pattern where default implementation of visit methods for derived classes forwards the call to visit of its base class. This is equivalent to matching objects of derived class against their base classes in case clauses. Visitors become slower in the presence of forwarding due to additional virtual function call, while type switch generally becomes faster due to smaller jump tables.
Finally, we did comparison of our technique against performance of pattern matching on closed algebraic data types of OCaml and Haskell and you can see that our solution is almost as efficient, while maintaining openness.
Since the performance of our solution relies in part on the structure of v-table layout by the compiler, we wanted to make sure that we do not make oversimplifying assumptions that result in that property. We thus took a class hierarchy benchmark that was used before to evaluate the speed of various run-time dispatching techniques. The most important property of this benchmark for us was that all the classes were written by humans. There was about 15K classes in the benchmark with about 64K paths (see sub-objects).

Type switch as well as visitor design pattern are generally applied to non-leaf classes only. 71% of the classes in the entire benchmarks suite were leaf classes. Out of the 4369 non-leaf classes, 36% were spawning a sub-hierarchy of only 2 classes (including the root), 15% – a sub-hierarchy of 3 classes, 10% of 4, 7% of 5 and so forth, which can be seen in the following chart. Turning this into a cumulative distribution, a% of sub-hierarchies had more than b classes in them, which reflects the percentage of real-world use cases with given number of case clauses.
For each of the 4369 non-leaf classes we generated a type-switch and a visitor class performing case analysis of all possible derived classes. The tests as well as classes were distributed among multiple translation units.

We put a dot in the picture based on the number of actual sub-objects (vtbl-pointers) that came through a given type switch and the computed probability of conflict that was achieved by that type switch using the optimal parameters k and l of its hashing function. We used shadow to show points where numerous experiments had the same outcome.

The most important observation here is that in 87.5% of cases the optimal hash function rendered no conflicts at all – these are the experiments lying on X axis. In other words only in 12.5% of cases the optimal functions had any conflicts at all.
Related Work

- N. Glew. *Type dispatch for named hierarchical types*. 1999
- M. Zenger, M. Odersky. *Independently extensible solutions to the expression problem*. 2005
- M. Homer, J. Noble, K. Bruce, A. Black, D. Pearce. *Patterns as Objects in Grace*. 2012
- N. H. Cohen. *Type-extension type test can be performed in constant time*. TOPLAS 1991
THANKS!

Mach7
- Matches the gold standard for notation
- Matches the gold standard for performance
- Handles both open and closed cases

Special Thanks To:
- Xavier Leroy
- Luc Maranget
- Gregory Berkolaiko
- Suhasini Subba Rao
- Jaakko Järvi
- Peter Pirkelbauer
- Andrew Sutton
- Abe Skolnik
- Karel Driesen

http://parasol.tamu.edu/mach7/
Our work makes the following contributions:

- **We provide a technique for implementing efficiently a type switch on hierarchical extensible data types of object-oriented languages.**
- **The technique comes close to the performance of case analysis on closed algebraic data types and matches or outperforms the visitor design pattern while getting rid of its limitations.**
- **We provide a fully functional library implementation that we plan to use to gather experience for a future language extension.**
- **We provide a way of partitioning objects based on sub-objects they are referred by, which can be used for other optimization techniques.**
Expression Problem

\[ \text{expr} ::= \text{val} | \text{expr} + \text{expr} | \text{expr} - \text{expr} | \text{expr} \times \text{expr} | \text{expr} / \text{expr} \]

<table>
<thead>
<tr>
<th>Functional Languages</th>
<th>Object-Oriented Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type expr =</code></td>
<td><code>class Expr { }</code></td>
</tr>
<tr>
<td>`</td>
<td>Value of int`</td>
</tr>
<tr>
<td>`</td>
<td>Plus of expr * expr`</td>
</tr>
<tr>
<td>`</td>
<td>Minus of expr * expr`</td>
</tr>
<tr>
<td>`</td>
<td>Times of expr * expr`</td>
</tr>
<tr>
<td>`</td>
<td>Divide of expr * expr`</td>
</tr>
</tbody>
</table>

let rec eval e =
  match e with
  Value v -> v
| Plus (a,b) -> (eval a) + (eval b) | int Value::eval() { return value; }
| Minus (a,b) -> (eval a) - (eval b) | int Plus ::eval() { return e1.eval()+e2.eval(); }
| Times (a,b) -> (eval a) * (eval b) | int Minus::eval() { return e1.eval()-e2.eval(); }
| Divide(a,b) -> (eval a) / (eval b) | int Times::eval() { return e1.eval()*e2.eval(); }
|                        | int Divide::eval(){ return e1.eval()/e2.eval(); }

Easy to add new functions
Adding new variants is intrusive

Easy to add new variants
Adding new functions is intrusive

Type switching is related to a well known expression problem that can be demonstrated on the following example. Imagine that you would like to implement an interpreter (or calculator) of a simple expression language represented by this grammar.

In functional languages, a term in the expression language will typically be represented with a recursive ADT with variants corresponding to production rules. In object-oriented languages, a term in the expression language will usually be represented with an abstract base class with derived classes representing the corresponding variants.

Functional languages then implement various operations on expression terms (like evaluation, printing or transformation) by simple case analysis on the variants. Object-oriented languages, on the other hand, will introduce a virtual function per each operation into the base class. The function is then overridden by each variant (derived class) to implement the logic of the operation on that specific variant.

One may notice that it is easy to add new operations on expression terms in functional languages, but adding new variants (i.e. productions here) requires modification of the ADT and all the existing functions on it. The situation is dual for object-oriented languages – it is easy to add new variants, but adding new operations require modification of the base class and all the existing variants.

This duality is known as the “expression problem” and have been extensively studied in the literature.
Problem of Type Switching in C++

- Classes are:
  - Extensible
    - Important: Separate compilation
    - Important: Dynamic linking
  - Hierarchical
    - Multiple Inheritance
    - Up-, down- and cross-casts
    - Cast is not a nu-up
    - Ambiguities
- Existing approaches
  - Closed world: jump tables
    - Unrealistic for modern C++ use
  - Open world: constant-time subtype tests + decision trees
    - Most are not suitable for repeated multiple inheritance
    - Most require computations or runtime code generation at load time
    - Time increases with case number

There are several problems to efficient implementation of type switching in object-oriented languages.

Unlike ADT, classes are extensible and hierarchical. Extensibility means that new variants can be added later, including at run-time in the presence of dynamic linking. The fact that classes are hierarchical means that variants are not necessarily disjoint and that the solution should be able to match more specific variants against more general ones in the case clause. The general multiple inheritance of C++ complicates the matter even further. The type inclusion tests as well as the outcomes of related type casts in such scenario become dependent on the notion of sub-object and the sub-object graph induced by different kinds of multiple inheritance present in C++.

Existing approaches to type switching are either efficient or open, but not both. Efficient approaches rely on closed world assumption, which is unrealistic for modern use of C++. Open approaches rely on constant-time subtype tests combined with decision trees. Unfortunately most of the known constant-time type tests are not suitable for repeated multiple inheritance and require non-trivial computations at load time in order to keep the solution open. Independently of these, the time increases with the case number because of the underlying decision tree, which quickly surpasses the time of known workaround for even small number of case clauses.
To be able to use memoization device for our purposes we need to be able to identify not only the dynamic type of an object, but also a sub-object within it, since an object may have several different sub-objects of the same static type. The following diagram shows a typical layout of object of type D under repeated multiple inheritance, which is a recursive composition of layouts of its base classes as well as its own members. While C++ does not require to use any particular technique to implement virtual functions, most of the compilers use virtual tables and embed a v-table pointer at a known offset into every class with virtual functions. Let us look at what those pointers are pointing to in what is called a “Common Vendor ABI”.

At positive offsets from the v-table pointer we have function pointers implementing virtual functions. At negative offsets we have a pointer to RTTI, offset to top of the object, offsets to virtual base classes etc. Since the content of a v-table is read-only, the compiler may unify parts of the table when they are the same. This is a common optimization technique when the use of RTTI is disabled. We can see now that the same v-table is reused by several different classes and sub-objects of the same class.

Nevertheless, we show in the paper that under the requirements of a common vendor ABI and in the presence of RTTI the v-table pointers are guaranteed to be unique for sub-objects of the same static type. This is easy to see from the picture because objects of different dynamic types will have to have different RTTI pointers in their v-tables, while different sub-objects of the same object will have different offset to top.
V-Table Pointers Facts

- Are unique per same static type only
  - can be shared with primary base class
- Can be many for same sub-object
  - e.g. numerous copies of the same v-table in DLLs
- May change during [de]construction
  - affects outcome of a type switch in constructors and destructors
  - is in line with C++ semantics for virtual function calls
- Are at fixed offset within the dynamic type
  - we can memoize offsets obtained on one instance
  - and reapply them to another instance
    - of the same dynamic type
    - from the same sub-object

Not sure whether to keep this slide.
We now compare the performance of our approach to the performance of visitor design pattern. Numbers in blue indicate cases when type switch was faster than visitors while numbers in red indicate cases when visitors were faster than type switch. The tests were made in 3 usage scenarios:

- **Repetitive** – can be seen in particle simulations where we have large number of objects of the same dynamic type.
- **Sequential** – we sequentially use object of each dynamic type only once before we move onto next one.
- **Random** – we use objects randomly in a random order.

Presence of forwarding indicates a common technique in visitor design pattern where default implementation of visit methods for derived classes forwards the call to visit of its base class. This is equivalent to matching objects of derived class against their base classes in case clauses.

Performance of our library solution benefits significantly from branch hinting, which can be indicated to G++ with some intrinsic functions. Unfortunately Visual C++ does not have a similar facility and they recommend to use Profile Guided Optimizations to achieve the same effect. This is why we provide the numbers for Visual C++ with and without Profile Guided Optimizations.

To give an idea about absolute timing, the following chart shows the comparison of absolute values leading to these relative ones. The actual bars represent the timings of visitors and type switch without forwarding while the black lines show where the corresponding bar would be in the presence of forwarding. It is easy to see that in case of repetitive benchmark both timings are significantly smaller because of the hardware cache. Visitors become slower in the presence of forwarding due to additional virtual function call, while type switch generally becomes faster due to smaller jump tables.

Finally, our library allows users to benefit from cases when the corresponding class hierarchy is known to be closed.
Sizes of Class Hierarchies

% of hierarchies with that number

% 100.00% 90.00% 80.00% 70.00% 60.00% 50.00% 40.00% 30.00% 20.00% 10.00% 0.00%

Classes in Hierarchy

1 4 16 64 256 1024 4096
Minimization of Conflicts

Pivot’s classes ranked by frequency of type-switching on them

Probability
Effect of Conflicts Minimization