Why does the arrow (->) operator in C exist?

The dot (`.`) operator is used to access a member of a struct, while the arrow operator (`->`) in C is used to access a member of a struct which is referenced by the pointer in question.

The pointer itself does not have any members which could be accessed with the dot operator (it's actually only a number describing a location in virtual memory so it doesn't have any members). So, there would be no ambiguity if we just defined the dot operator to automatically dereference the pointer if it is used on a pointer (an information which is known to the compiler at compile time afaik).

So why have the language creators decided to make things more complicated by adding this seemingly unnecessary operator? What is the big design decision?

I'll interpret your question as two questions: 1) why `->` even exists, and 2) why `.` does not automatically dereference the pointer. Answers to both questions have historical roots.

Why does `->` even exist?

In one of the very first versions of C language (which I will refer as CRM for "C Reference Manual", which came with 6th Edition Unix in May 1975), operator `->` had very exclusive meaning, not synonymous with `*` and `.` combination

The C language described by CRM was very different from the modern C in many respects. In CRM struct members implemented the global concept of *byte offset*, which could be added to any address value with no type restrictions. I.e. all names of all struct members had independent global meaning (and, therefore, had to be unique). For example you could declare

```
struct S {
  int a;
  int b;
};
```

and name `a` would stand for offset 0, while name `b` would stand for offset 2 (assuming `int` type of size 2 and no padding). The language required all members of all structs in the translation unit either have unique names or stand for the same offset value. E.g. in the same translation unit you could additionally declare

```
struct X {
  int a;
  int x;
};
```

and that would be OK, since the name `a` would consistently stand for offset 0. But this additional declaration

```
struct Y {
  int b;
  int a;
};
```

would be formally invalid, since it attempted to "redefine" `a` as offset 2 and `b` as offset 0. And this is where the `->` operator comes in. Since every struct member name had its own self-sufficient global meaning, the language supported expressions like these

```
int i = 5;
i->b = 42;  /* Write 42 into `int` at address 7 */
100->a = 0; /* Write 0 into `int` at address 100 */
```

The first assignment was interpreted by the compiler as "take address `5`, add offset `2` to it and assign `42` to the `int` value at the resultant address". I.e. the above would assign `42` to `int` value at address `7`. Note that this use of `->` did not care about the type of the expression on the left-hand side. The left hand side was interpreted as an rvalue numerical address (be it a pointer or an integer).

This sort of trickery was not possible with `*` and `.` combination. You could not do

```
(*i).b = 42;
```

since `*i` is already an invalid expression. The `*` operator, since it is separate from `.`, imposes more strict type requirements on its operand. To provide a capability to work around this limitation CRM introduced the `->` operator, which is independent from the type of the left-hand operand.

As Keith noted in the comments, this difference between `->` and `*+.` combination is what CRM is referring to as "relaxation of the requirement" in 7.1.8: *Except for the relaxation of the requirement that `E1` be of pointer type, the expression `E1->MOS` is exactly equivalent to `(*E1).MOS`*

Later, in K&R C many features originally described in CRM were significantly reworked. The idea of "struct member as global offset identifier" was completely removed. And the functionality of `->`operator became fully identical to the functionality of `*` and `.`combination.

Why can't `.` dereference the pointer automatically?

Again, in CRM version of the language the left operand of the `.`operator was required to be an *lvalue*. That was the *only* requirement imposed on that operand (and that's what made it different from `->`, as explained above). Note that CRM did *not* require the left operand of `.` to have a struct type. It just required it to be an lvalue, *any*lvalue. This means that in CRM version of C you could write code like this

```
struct S { int a, b; };
struct T { float x, y, z; };

struct T c;
c.b = 55;
```

In this case the compiler would write 55 into an int value positioned at byte-offset 2 in the continuous memory block known as c, even though type struct T had no field named b. The compiler would not care about the actual type of c at all. All it cared about is that c was an lvalue: some sort of writable memory block.

Now note that if you did this

```
S *s;
...
s.b = 42;
```

the code would be considered valid (since s is also an lvalue) and the compiler would simply attempt to write data *into the pointer sitself*, at byte-offset 2. Needless to say, things like this could easily result in memory overrun, but the language did not concern itself with such matters.

I.e. in that version of the language your proposed idea about overloading operator . for pointer types would not work: operator .already had very specific meaning when used with pointers (with lvalue pointers or with any lvalues at all). It was very weird functionality, no doubt. But it was there at the time.

Of course, this weird functionality is not a very strong reason against introducing overloaded . operator for pointers (as you suggested) in the reworked version of C - K&R C. But it hasn't been done. Maybe at that time there was some legacy code written in CRM version of C that had to be supported.

(The URL for the 1975 C Reference Manual may not be stable. Another copy, possibly with some subtle differences, is here.)