Issues in Moving from C to C++ (PANEL)

David R. Reed, *Saber Software Inc.*, (moderator)
Marty Cagan, *Interactive Development Environments*
Ted Goldstein, *Sun Laboratories*
Barbara Moo, *AT&T*

Background

C++ is rapidly emerging as the most widely used object oriented language. One major reason is that C++ is derived from the popular C programming language. Because C++ is considered the successor to C, C programmers wanting the benefits of the object oriented paradigm look to C++ as a logical step toward object oriented programming.

This panel will focus on the issues in moving from C to C++ and will try to present different points of view on how the C programmer should make the transition. Simply stated, one view promotes an incremental style in adapting both the language and the paradigm, the other more radical teaching, takes the view that the C programmer must adopt both a new paradigm and language at the onset.

Other related topics include:

- To what extent can existing systems be extended and enhanced using object-oriented programming techniques, and what are the issues involved and the solutions available.
- Many existing C programs were designed and modeled using conventional structured design techniques. What are the issues relating to front-end design of systems written in a mix of C and C++.
- Does using technologies associated with C++ such as object-oriented database technology require extensive knowledge of the object-oriented paradigm.
- Is it practical to start a new project using C++ simply as a better C, and then gradually employ object-oriented programming techniques as the project progresses? What would be the impact of this learn-as-you-go approach on program quality and team productivity.

The panel will be made up of both end users and technology providers that have experience in making the transition from C to C++, some with a particular focus such as front-end design methodologies.

Marty Cagan

At IDE we build integrated environments to support both C and C++ development, as well as combinations of the two. Like ourselves, many of our customers have been migrating from C to C++.

The migration from structured development to object-oriented development presents several interesting issues from the perspective of design support environments. A long-standing limitation of structured development has been the gap between the various levels of specification and implementation. Object-oriented development holds the promise of reducing this gap by providing a smooth transformation between specification and implementation.

The question then arises as to the best way for an organization to move from a structured development process to an object-oriented one. At this panel we will discuss experiences of groups that have moved to object-oriented design and implementation, but have remained with structured analysis, groups that have moved to object-oriented analysis methods yet remained with conventional structured design and implementation languages, and groups that have moved completely to object-oriented analysis, design and implementation.

Barbara Moo

The path to Object Oriented Happiness: Immediate or Gradual?

My position is based on the experiences of AT&T R&D projects using C++ and Object Oriented Programming over the past 5 years. The issues in moving to OOP are primarily learning curve issues: how to design and think about problems in a new way.

It is much harder to convince a skeptical development manager to try a new technology if they have to start off with a large investment before seeing any return. C++ allows us to "learn while doing"; we can capitalize on the large investment we already have in C knowledge and gradually evolve systems and people to use the data abstraction and OOP facilities of the language to dramatically improve productivity. Our approach:

- In the first release, keep it simple, using the "better C" facilities of C++. Senior members of the team define and implement a few central data abstractions that model the application for use by all on the project.
- In subsequent releases more opportunities for class design will be clearer and opportunities for reuse and inheritance from the original classes will present themselves.

The first release gets staff comfortable with the language and with concentrating on the data and builds both a human and a code base for more aggressive application of the technology in later releases. Also, management confidence is increased since even the first release shows benefits from stronger type checking and modest reuse benefits.

This process has been successfully applied to projects ranging in size up to 70 developers. The benefits of the approach:

- lower risk in taking in a new technology
- training gets spread across a release rather than being concentrated up front
- learning, we find, is more effective if coordinated with practice
- gives time for local experts to evolve

The benefit of spreading out the learning costs is especially important when migrating larger projects to a new technology. We have had some successes with the direct leap approach as well. These tend to come on small (1 - 5 in staff size) projects without rigorous schedule constraints. These projects, because they are small, can learn as a group, and because they are not over constrained with deadlines can afford to change design as learning progresses. This is infeasible both for communication reasons and for schedule on larger projects.

Ted Goldstein

Once upon a time, the "Software crisis" was the backlog in application development by the corporate MIS department. Structured programming and artificial intelligence was the solution proposed to solve this first software crisis. Complex systems such as user interfaces, databases, and network services have brought about a new crisis of ever more interrelated and complex software. Recently, object-oriented programming has been proposed as the solution to both crises.

Unfortunately, object-oriented programming is not a panacea. There is considerable risk involved in making any technology transition, and object-oriented technology is no exception. A similar situation occurred during the 1980's to AI programming. Many promises made, but few were kept. In the early '80's, artificial intelligence was supposed to solve the "software crisis". Reality set in and many problems turned out to be harder than expected. People became disillusioned and even those AI techniques which had validity were dismissed. This had led to the current state of an "AI winter."

There are a number of issues involving languages, systems, and libraries which could easily bring on "an object-oriented winter." It is important for managers and practitioners of object-oriented programming to recognize exactly what the technology can accomplish. Myths concerning productivity improvement, language choice, dynamic versus statically typed languages, performance results, and testing must be countered with the reality of experience in object-oriented programming. The reality behind object-oriented programming is more about making a commitment to well designed software, and making the capital investment in training and tools to accomplish the goal of software quality and overcome the software crisis.