Overview of C++: Design Goals

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Recognize the key components of C++
• Know strategies for learning C++
• Understand C++ design goals
C++ Design Goals
C++ Design Goals

• As with C, run-time efficiency is important
C++ Design Goals

- As with C, run-time efficiency is important
- Zero-overhead abstraction
  - e.g., classes with constructors & destructors, inheritance, generic programming, functional programming techniques, etc.

See [www.youtube.com/watch?v=G5zCGY0tkq8](www.youtube.com/watch?v=G5zCGY0tkq8)
C++ Design Goals

• As with C, run-time efficiency is important
  • Zero-overhead abstraction
• Direct mapping to hardware
  • e.g., no virtual machine overhead for instructions & native data types
C++ Design Goals

- As with C, run-time efficiency is important
  - Zero-overhead abstraction
  - Direct mapping to hardware
- No complicated run-time libraries, managed environments, or virtual machines
  - Unlike other languages, e.g., Ada, Java, C#, etc.

See [en.wikipedia.org/wiki/Gordian_Knot](en.wikipedia.org/wiki/Gordian_Knot)
C++ Design Goals

• As with C, run-time efficiency is important
  • Zero-overhead abstraction
  • Direct mapping to hardware
  • No complicated run-time libraries, managed environments, or virtual machines
• No language-specific support for persistence, garbage collection, or networking in C++
C++ Design Goals

• As with C, run-time efficiency is important
  • Zero-overhead abstraction
  • Direct mapping to hardware
  • No complicated run-time libraries, managed environments, or virtual machines
• No language-specific support for persistence, garbage collection, or networking in C++
  • Additional support for threading, synchronization, & parallelism was added beginning w/C++11

C++ Design Goals

• As with C, run-time efficiency is important
  • Zero-overhead abstraction
  • Direct mapping to hardware
  • No complicated run-time libraries, managed environments, or virtual machines
• No language-specific support for persistence, garbage collection, or networking in C++
  • Many libraries exist that provide these capabilities

See www.dre.vanderbilt.edu/ACE & www.boost.org
C++ Design Goals

• Compatibility w/C libraries & traditional development tools is emphasized
C++ Design Goals

- Compatibility w/C libraries & traditional development tools is emphasized, e.g.,
  - Object code reuse
    - e.g., the storage layout of structs is compatible with C
C++ Design Goals

- Compatibility w/C libraries & traditional development tools is emphasized, e.g.,
  - Object code reuse
    - e.g., the storage layout of structs is compatible with C
  - Supports the standard ANSI C library, UNIX & Windows system calls via extern blocks, etc.

See [www.dre.vanderbilt.edu/~schmidt/ACE/book1](http://www.dre.vanderbilt.edu/~schmidt/ACE/book1)
C++ Design Goals

• Compatibility w/C libraries & traditional development tools is emphasized, e.g.,
  • Object code reuse
  • C++ works with the “make” family of (re)compilation build tools

See www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html
C++ Design Goals

• An initial design goal was for C++ to be “as close to C as possible, but no closer”
C++ Design Goals

• An initial design goal was for C++ to be “as close to C as possible, but no closer”
• i.e., C++ is not a proper superset of C
• Backwards compatibility with C is not entirely maintained
C++ Design Goals

- An initial design goal was for C++ to be “as close to C as possible, but no closer”
- i.e., C++ is not a proper superset of C
- Backwards compatibility with C is not entirely maintained

```cpp
void *ptr;
/* Implicit conversion from void* to int* */
int *i = ptr;

/* Implicit conversion from void* to int* */
int *j = malloc(5 * sizeof *j);
```

C++ Design Goals

• An initial design goal was for C++ to be “as close to C as possible, but no closer”
  • i.e., C++ is not a proper superset of C
  • Backwards compatibility with C is not entirely maintained

```cpp
void *ptr;
/* Implicit conversion from void* to int* */
int *i = ptr;
/* Implicit conversion from void* to int* */
int *j = malloc(5 * sizeof *j);
```

**Valid in C, but not in C++**

```cpp
void *ptr;
int *i = (int *)ptr;
int *j = (int *)
    malloc(5 * sizeof *j);
```

**Valid in C++ & C**

See [en.wikipedia.org/wiki/Compatibility_of_C_and_C++]
C++ Design Goals

- An initial design goal was for C++ to be “as close to C as possible, but no closer”
- i.e., C++ is not a proper superset of C
  - Backwards compatibility with C is not entirely maintained

**Preferred in C++**

```cpp
void *ptr;
auto i =
    reinterpret_cast<int *>(ptr);
auto j = new int[5];
```

**Valid in C, but not in C++**

```cpp
void *ptr;
/* Implicit conversion from void* to int* */
int *i = ptr;

/* Implicit conversion from void* to int* */
int *j =
    malloc(5 * sizeof *j);
```

**Valid in C++ & C**

```cpp
void *ptr;
int *i = (int *)ptr;
int *j = (int *)
    malloc(5 * sizeof *j);
```
C++ Design Goals

• An initial design goal was for C++ to be “as close to C as possible, but no closer”
  • i.e., C++ is not a proper superset of C
    • Backwards compatibility with C is not entirely maintained
  • Typically not a problem in practice...
C++ Design Goals

- Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively
C++ Design Goals

- Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively
- Generic programming generalizes software components so that they can be easily reused in many situations

See [www.boost.org/community/generic_programming.html](http://www.boost.org/community/generic_programming.html)
C++ Design Goals

• Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively
  • Generic programming generalizes software components so that they can be easily reused in many situations
  • C++ templates enable generic programming since they generalize without sacrificing efficiency

int a[] = {1, 2, 3, ...};
vector<int> v = {1, 2, 3, ...};

copy(a, a + sizeof(a)/sizeof(*a), ostream_iterator<int>(cout));
copy(v.begin(), v.end(), ostream_iterator<int>(cout));

See www.boost.org/community/generic_programming.html
C++ Design Goals

• Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively
  • Generic programming generalizes software components so that they can be easily reused in many situations
  • The C++ core guidelines are a set of idioms documented to help developers efficiently and consistently write type & resource safe C++ programs

C++ Core Guidelines

“Within C++ is a smaller, simpler, safer language struggling to get out.” – Bjarne Stroustrup

The C++ Core Guidelines are a collaborative effort led by Bjarne Stroustrup, much like the C++ language itself. They are the result of many person-years of discussion and design across a number of organizations. Their design encourages general applicability and broad adoption but they can be freely copied and modified to meet your organization’s needs.

The aim of the guidelines is to help people to use modern C++ effectively. By “modern C++” we mean C++11 and C++14 (and soon C++17). In other words, what would you like your code to look like in 5 years’ time, given that you can start now? In 10 years’ time?

The guidelines are focused on relatively higher-level issues, such as interfaces, resource management, memory management, and concurrency. Such rules affect application architecture and library design. Following the rules will lead to code that is statically type safe, has no resource leaks, and catches many more programming logic errors than is common in code today. And it will run fast - you can afford to do things right.

See isocpp.github.io/CppCoreGuidelines
C++ Design Goals

• Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively
  • Generic programming generalizes software components so that they can be easily reused in many situations
  • The C++ core guidelines are a set of idioms documented to help developers efficiently and consistently write type & resource safe C++ programs

R.11: Avoid calling `new` and `delete` explicitly

**Reason** The pointer returned by `new` should belong to a resource handle (that can call `delete`). If the pointer returned by `new` is assigned to a plain/naked pointer, the object can be leaked.

**Note** In a large program, a naked `delete` (that is a `delete` in application code, rather than part of code devoted to resource management) is a likely bug: if you have N `delete`s, how can you be certain that you don’t need N+1 or N-1? The bug may be latent: it may emerge only during maintenance. If you have a naked `new`, you probably need a naked `delete` somewhere, so you probably have a bug.

**Enforcement** (Simple) Warn on any explicit use of `new` and `delete`. Suggest using `make_unique` instead.

See [isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html#Rr-newdelete](isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html#Rr-newdelete)
End of C++
Design Goals