C++ was designed at AT&T Bell Labs by Bjarne Stroustrup in the early 80's—nearly 30 years ago! C++ is a mostly upwardly compatible extension of C that provides:

- Stronger typechecking
- Support for data abstraction
- Support for object-oriented programming
- Support for generic programming

Many native host machine compilers now exist—e.g., C++ Builder, clang, Compaq C/C++, GCC, Intel C++, etc. However, this was difficult to debug and potentially inefficient. The original Cfront translated C++ into C for portability.
C++ Design Goals

- As with C, run-time efficiency is important
- `&` separate compilation complicates inlining due to difficulty of re-entrancy
- Software engineering
  1. Compiler optimization
  2. Storage & garbage collection

- In C++, pointers to arbitrary memory locations can complicate register allocation
- Note that there is no language-specific support for concurrency.
  - Unlike other languages (e.g., Ada, Java, C#), etc.)
  - As with C, run-time efficiency is important
  - Note, language-specific support for concurrency has been required for C++

C++ Design Goals (cont’d)
Major C++ Enhancements

1. C++ supports data abstraction & encapsulation
   - e.g., the class mechanism & name spaces
2. C++ supports object-oriented programming features
   - e.g., abstract classes, inheritance, & virtual methods
3. C++ supports generic programming
   - e.g., the class mechanism & name spaces
4. C++ supports sophisticated error handling
   - e.g., exception handling
5. C++ supports parameterized types
   - e.g., Run-Time Type Identification (RTTI)

Important Minor C++ Enhancements

- C++ enforces type checking via function prototypes
- C++ supports identifying an object's type at runtime
  - e.g., Run-Time Type Identification (RTTI)
- C++ supports sophisticated error handling
- C++ supports parameterized types
- C++ supports generic programming
- C++ supports data abstraction & encapsulation
An Overview of C++

Useful Minor C++ Enhancements

- The name of a `struct`, class, enum, or union is a type name
- Static data initializers may be arbitrary expressions
- Allows user-defined conversion operators within a block
- Variable declarations may occur anywhere statements may appear
- Operator `&` function overloading
- Default values for function parameters
- References allow „call-by-reference“ parameter modes

Questionable C++ Features

- Default values for function parameters
- Operator & function overloading
- Variable declarations may occur anywhere statements may appear
- Allows user-defined conversion operators
- Static data initializers may be arbitrary expressions

Useful Minor C++ Enhancements

- New boolean type
- New mutable type qualifier
- Several different commenting styles
- „Function call”-style cast notation
- New type-safe extensible I/O streams I/O mechanism
- References allow „call-by-reference“ parameter modes
- The name of a `struct`, class, enum, or union is a type name
An Overview of C++ Douglas C. Schmidt

1. Concurrency
   - "Concurrent C++" by Gehani
   - Actor++ model by Lavender & Kafura

2. Persistence
   - Object Store, Versant, Objectivity
   - Exodus system & E programming language

3. Garbage Collection
   - Exodux system & E programming language
   - “Concurrent C++” by Gehani

4. Distribution
   - CORBA, DDS, COM+, SOAP, etc.
   - GNU C++
   - USENIX C++ 1994 paper by Ellis & Detlefs

Strategies for Learning C++

• Focus on concepts & programming techniques
  – Don’t get lost in language features
  – Don’t have to know every detail of C++ to write a good C++
  – Learn C++ gradually
  – Learn C++ to become a better programmer

Vanderbilt University
An Overview of C++

Stack Example

- Begin with C & evolve up to various C++ implementations.

First, consider the "bare-bones" implementation:

```c
typedef int T;

#define MAX_STACK 100 /* const int MAX_STACK = 100; */

T stack[MAX_STACK];
int top = 0;

T item = 10;
stack[top++] = item; // push
...

item = stack[--top]; // pop
```

Obviously not very abstract...

Define the interface to a Stack of integers in C in Stack.h:

```c
/* Type of Stack element. */
typedef int T;

/* Stack interface. */
int create (int size);
int destroy (void);
void push (T new_item);
void pop (T *old_top);
void top (T *cur_top);
int is_empty (void);
int is_full (void);
```

The following slides examine several alternative methods of implementing a Stack Example.
Data Hiding Implementation in C (cont'd)

Main problems:

1. The programmer must call create() first & destroy() last!
2. There is only one stack & only one type of stack
3. Name space pollution...
4. Non-reentrant

#include "stack.h"

void foo (void) {
    T i;
    push (10); /* Oops, forgot to call create! */
    push (20);
    pop (&i);
    free (void*) stack_;
}

int is_empty (void) { return top_ == 0; }
int is_full (void) { return top_ == size_; }

Vanderbilt University
An Overview of C++
Douglas C. Schmidt
An Overview of C++ Douglas C. Schmidt

Data Abstraction Implementation in C

• An ADT Stack interface in C:

```c
typedef int T;
typedef struct { size_t top_; size_t size_; T *stack_ } Stack;

int Stack_create (Stack *s, size_t size);
void Stack_destroy (Stack *s);
void Stack_push (Stack *s, T item);
void Stack_pop (Stack *, T *item);
/* Must call before pop'ing */
int Stack_is_empty (Stack *);
/* Must call before push'ing */
int Stack_is_full (Stack *);
/* ... */
```

• An ADT Stack implementation in C:

```c
#include "stack.h"

int Stack_create (Stack *s, size_t size) {
    s->top_ = 0; s->size_ = size;
    s->stack_ = malloc (size * sizeof (T));
    return s->stack_ ? 0 : -1;
}

void Stack_destroy (Stack *s) {
    free ((void *) s->stack_); s->top_ = 0; s->size_ = 0; s->stack_ = 0;
}

void Stack_push (Stack *s, T item) {
    s->stack_[s->top_] = item;
}

void Stack_pop (Stack *s, T *item) {
    *item = s->stack_[--s->top_];
}

int Stack_is_empty (Stack *s) { return s->top_ == 0; }
int Stack_is_full (Stack *s) {
    return s->stack_ == (s->stack_ + s->size_ - 1); // Stack is full
}

/* ...
int Stack_is_empty (Stack *);
int Stack_is_full (Stack *);
... */
```

Vanderbilt University
Main problems with Data Abstraction in C

1. No guaranteed initialization, termination, or assignment
2. Too much overhead due to function calls
3. Still only one type of stack supported
4. No generalized error handling...
5. The C compiler does not enforce information hiding e.g.,

...and more...
We can get encapsulation and more than one stack.

```cpp
struct Stack {
    // Constructor
    Stack (size_t size); 
    Stack (const Stack &s); 
    void operator= (const Stack &s); 
    ~Stack (void); 

    void push (const T &item); 
    void pop (T &item); 
    bool is_empty (void) const; 
    bool is_full (void) const; 

private:
    size_t top_, size_; T *stack_; 
};
```

Manager operations

**Data Abstraction Implementation in C++ (cont'd)**
Data Abstraction Implementation in C++ (cont'd)
An Overview of C++

Benefits of C++ Data Abstraction Implementation

1. Data hiding & data abstraction, e.g.,
   ```c++
   Stack s1(200);
   s1.top_ = 10  // Error flagged by compiler!
   ```
2. The ability to declare multiple stack objects
   ```c++
   Stack s1(10), s2(20), s3(30);
   ```
3. Automatic initialization & termination
   ```c++
   Stack s1(1000);  // constructor called automatically.
   ```
   ```c++
   // ...
   ```
   ```c++
   // Destructor called automatically
   ```

Drawbacks with C++ Data Abstraction Implementation

1. Error handling is obtrusive
   - Use exception handling to solve this (but be careful!
2. The example is limited to a single type of stack element (int in this case)
   - We can use C++'s "parameterized types" to remove this limitation
3. Function call overhead
   - We can use C++'s "inline functions" to remove this overhead

Vanderbilt University
Exception Handling Implementation in C++ (cont'd)

Stack::Stack (size_t s): top_ (s), size_ (s), stack_ (new T[s]) {

}

Stack::Stack () { delete [] stack_; }

void Stack::push (const T &item) throw (Stack::Overflow) {
    if (is_full ()) throw Stack::Overflow ();
    stack_[top_++] = item;
}

void Stack::pop (T &item) throw (Stack::Underflow) {
    if (is_empty ()) throw Stack::Underflow ();
    item = stack_[--top_];
}

Exception Handling Implementation in C++ (cont'd)
• scoped-array extends auto-ptr to destroy built-in arrays

```cpp
template <typename T> class scoped_array {
public:
    explicit scoped_array (T *p = 0) : ptr_ (p) {}
    ~scoped_array () { delete [] ptr_; }
    T &operator[](std::ptrdiff_t i) const { return ptr_[i]; }
    T *get() const { return ptr_; }
    void swap (T *&b) { T *tmp = b; b = ptr_; ptr_ = tmp; }
    void swap (scoped_array<T> &b) { T *tmp = b.ptr_; b.ptr_ = this->ptr_; this->ptr_ = tmp; }

private:
    T *ptr_; // Copy constructor and assignment operator are defined.
}
```

Exception Handling Implementation in C++ (cont'd)
An Overview of C++ Douglas C. Schmidt

Exception Handling Implementation in C++ (cont'd)

• There's a better way to implement exception-safe Stack:

```c++
class Stack { // ...
private: // ...
    scoped_array<T> stack_; // ... };

Stack::Stack (const Stack &s) : top_ (s.top_), size_ (s.size_), stack_ (new T[s.size_]) {
    for (size_t i = 0; i < s.size_; ++i) stack_[i] = s.stack_[i];
    std::swap (t, s); // Check for self-assigment
    ... // Old way (cont'd)
}
```

And yet an even better way to implement exception-safe Stack:

```c++
void Stack::operator= (const Stack &s) {
if (this == &s) return; // Check for self-assigment
Stack t (s);
std::swap (t, *this); // Check for self-assigment
... // Old way (cont'd)
}
```
And yet an even better way to implement `operator=()`:

```
void Stack::operator= (const Stack &s) {
    if (this == &s) return; // Check for self-assignment
    Stack temp_stack (s);
    swap (temp_stack);
}
```

This solution is easy to generalize!

```
#include "Stack.h"

void foo (void) {
    Stack s1 (1), s2 (100);
    try {
        T item;
        s1.push (473);
        s1.push (42); // Exception, push'd full stack!
        s2.pop (item); // Exception, pop'd empty stack!
        s1.push (473);
        s1.push (42);
        s2.pop (item);
    } catch (Stack::Underflow) { /* Handle underflow... */ }
    catch (Stack::Overflow) { /* Handle overflow... */ }
    catch (...) { /* Catch anything else... */ throw; }
} // Termination is handled automatically.
```

```
A parameterized type Stack class interface using C++:

```
template <typename T> class Stack {
public:
    Stack (size_t size);
    Stack (const Stack<T> &rhs);
    void operator= (const Stack<T> &rhs);
    ~Stack (void);
    void push (const T &item);
    void pop (T &item);
    bool is_empty (void) const;
    bool is_full (void) const;
private:
    size_t top_, size_; scoped_array<T> stack_;
    void swap (scoped_array<T> &t);
};
```

A parameterized type Stack class implementation using C++ (cont'd):

```
template <typename T> inline Stack<T>::Stack (size_t size)
    : top_ (0), size_ (size), stack_ (new T[size]) { }
template <typename T> inline Stack<T>::~Stack (void) { /* no-op! */ }
template <typename T> inline void Stack<T>::push (const T &item) { stack_[top_] = item; }
template <typename T> inline void Stack<T>::pop (T &item) { item = stack_[--top_]; }
```

A parameterized type Stack class interface using C++:

```
// ...
```

A parameterized type Stack class implementation using C++ (cont'd):

```
};
```

Vanderbilt University
Temple Implementation in C++ (cont'd)

• Note minor changes to accommodate parameterized types

```cpp
#include "Stack.h"

void foo (void) {
  Stack<int> s1 (1000);
  Stack<float> s2;
  Stack<Stack<Activation_Record> *> s3;

  s1.push (-291);
  s2.top_ = 3.1416; // Access violation caught!
  s3.push (new Stack<Activation_Record>);
  Stack<Activation_Record> *sar;
  s3.pop (sar);
  delete sar; // Termination of s1, s2, & s3 handled automatically
}
```

Another parameterized type Stack class

```cpp
template <typename T, size_t SIZE> class Stack {
  public:
    Stack (void);
    ~Stack (void);
    void push (const T &item);
    void pop (T &item);
  private:
    size_t top_, size_;
    T stack_[SIZE];
};
```

No need for dynamic memory, though SIZE must be const, e.g.,
```cpp
Stack<int, 200> s1;
```
• Problems with previous examples:
  – Changes to the implementation will require recompilation & relinking
  – Extensions will require access to the source code
  – Solutions:
    – Combine inheritance with dynamic binding to completely decouple interface from implementation & binding time
    – By using „pure virtual methods“, we can guarantee that the compiler won't allow instantiation!

```cpp
// Define an abstract base class in C++

template <typename T>
class Stack {
public:
  virtual ~Stack (void) = 0; // Need implementation!
  virtual void push (const T &item) = 0;
  virtual void pop (T &item) = 0;
  virtual bool is_empty (void) const = 0;
  virtual bool is_full (void) const = 0;
  void top (T &item) { /* Template Method */ pop (item); push (item); }
};
```

Object-Oriented Implementation in C++ (cont'd)
An Overview of C++

Object-Oriented Implementation in C++ (cont'd)

Vanderbilt University
38

included to create a specialized stack implemented via an STL vector.
Inheritance can also create an linked list stack:

```cpp
template <typename T> class Node; // forward declaration.
template <typename T> class L_Stack : public Stack<T> {
public:
    enum { DEFAULT_SIZE = 100 }
    L_Stack (size_t hint = DEFAULT_SIZE);
    ~L_Stack (void);
    virtual void push (const T &new_item);
    virtual void pop (T &top_item);
    virtual bool is_empty (void) const { return head_ == 0; }
    virtual bool is_full (void) const { return 0; }
private:
    // Head of linked list of Node<T>'s.
    Node<T> *head_;
};
```

Note that the use of the "Cheshire cat" idiom allows the library writer to completely hide the representation of class 'Stack'.

```
friend template <typename T> class L_Stack;
```

### Node class implementation

```cpp
template <typename T> class Node {
    friend template <typename T> class L_Stack;
public:
    Node (T i, Node<T> *n = 0): item_ (i), next_ (n) {};
private:
    T item_; // Item.
    Node<T> *next_; // Next.
};
```

Note that the use of the "Cheshire cat" idiom allows the library writer to completely hide the representation of class 'Stack'.

Inheritance can also create an linked list stack:
• class L_Stack implementation:
  template <typename T> L_Stack<T>::L_Stack (size_t): head_ (0) {}
  template <typename T> void L_Stack<T>::push (const T &item) {
    Node<T> *t = new Node<T> (item, head_); head_ = t;
  }
  template <typename T> void L_Stack<T>::pop (T &top_item) {
    top_item = head_->item_; Node<T> *t = head_; head_ = head_->next_; delete t;
  }
  template <typename T> L_Stack<T>::~L_Stack (void) {
    for (T t; head_ != 0; pop (t)) continue;
  }

Using our abstract base class, it is possible to write code that does not depend on the stack implementation, e.g.,

```cpp
Stack<int> *make_stack (bool use_V_Stack) {
  return use_V_Stack ? new V_Stack<int> : new L_Stack<int>;
}
```

```cpp
void print_top (Stack<int> *stack) {
  std::cout << "top = " << stack->top () << std::endl;
}
```

```cpp
int main (int argc, char **) {
  std::auto_ptr <Stack<int>> sp (make_stack (argc > 1));
  sp->push (10);
  print_top (sp.get ());
}
```

Object-Oriented Implementation in C++ (cont'd)
Moreover, we can make changes at run-time without modifying existing code via dynamic linking: char stack_t::filename[MAXFILENAME] = "file.ttt";
char stack_t::template_stack[template_stack_len];

Note, no need to stop, modify, & restart an executing application!

An Overview of C++

The Road Ahead: C++11

The modifications for C++ involve both the core language and the standard library:
- Improve C++ to facilitate systems and library design, rather than to extend existing core language
- Prefer introduction of new features through the standard library, rather than by providing safer alternatives to current, unsafe techniques;
- Increase type safety by providing safer alternatives to current, unsafe techniques;
- Introduce new features only useful to specific applications;
- Increase performance and the ability to work directly with hardware;

Implement “zero-overhead” principle (additional support required by implementations must be used only if the utility is used).

Objective-Oriented Implementation in C++ (cont’d)
C++11 Example

```cpp
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>

int main () {
  std::vector<int> v ({10, 20, 30, 40});
  for (int &i : v) std::cout << i << std::endl;
  auto total = 0;
  std::for_each (v.begin (), v.end (),
                  [&total](int x) { total += x; });
  std::cout << total << std::endl;
  return 0;
}
```

Summary

- A major contribution of C++ is its support for defining abstract data
types (ADTs) for generic programming.
- For some systems, C++'s ADT support is more important than using
  the OO features of the language.
- For other systems, the use of C++'s ADT support is essential to build
  highly flexible and extensible software.
- e.g., classes, parameterized types, exception handling.
- e.g., inheritance, dynamic binding, RTTI.

For some systems, C++'s ADT support is more important than using
the OO features of the language.

A major contribution of C++ is its support for defining abstract data

classes, parameterized types, exception handling.