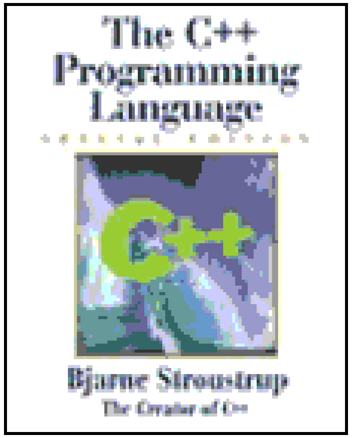
A Condensed Crash Course on C++

ECE 417/617:
Elements of Software Engineering

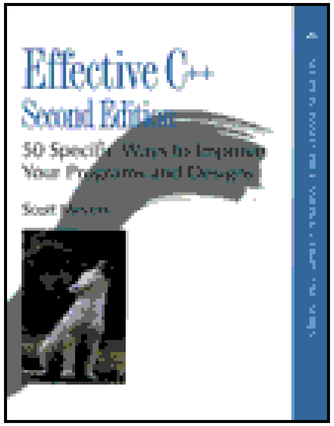
Stan Birchfield
Clemson University

Recommended C++ resources

- **Bjarne Stroustrup,**
  The C++ Programming Language



- **Scott Meyers,**
  Effective C++



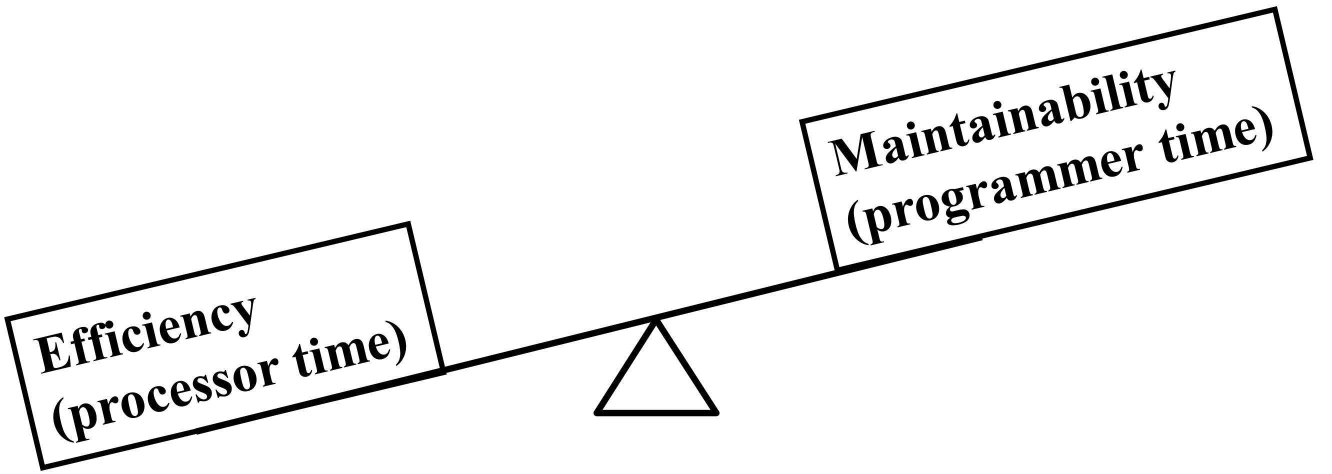
Why C++?

- **Popular and relevant (used in nearly every application domain):**
  - **end-user applications (Word, Excel, PowerPoint, Photoshop, Acrobat, Quicken, games)**
  - **operating systems (Windows 9x, NT, XP; IBM's K42; some Apple OS X)**
  - **large-scale web servers/apps (Amazon, Google)**
  - **central database control (Israel's census bureau; Amadeus; Morgan-Stanley financial modeling)**
  - **communications (Alcatel; Nokia; 800 telephone numbers; major transmission nodes in Germany and France)**
  - **numerical computation / graphics (Maya)**
  - **device drivers under real-time constraints**
- **Stable, compatible, scalable**

C vs. C++

- **C++ is C incremented**
  (orig., "C with classes")
- **C++ is more *expressive***
  (fewer C++ source lines needed than C source lines for same program)
- **C++ is just as *permissive***
  (anything you can do in C can also be done in C++)
- **C++ can be just as *efficient***
  (most C++ expressions need no run-time support;
  C++ allows you to
  - **manipulate bits directly and interface with hardware without regard for safety or ease of comprehension, BUT**
  - **hide these details behind a safe, clean, elegant interface)**
- **C++ is more *maintainable***
  (*1000 lines of code* – even brute force, spaghetti code will work;
  ***100,000 lines of code* – need good structure, or new errors will be introduced as quickly as old errors are removed)**

Efficiency and Maintainability



90/10 rule: 10% of your program will take 90% of the processor time to run

→ optimize what needs to be optimized, but no more

→ focus on design

Design goals of C++

- **Backward compatibility with C**
  (almost completely – every program in K&R is a C++ program – but additional keywords can cause problems)
- **Simplicity, elegance**
  (few built-in data types, e.g., no matrices)
- **Support for user-defined data types**
  (act like built-in types; N.B. Standard Template Library (STL))
- **No compromise in efficiency, run-time or memory**
  (unless "advanced features" used)
- **Compilation analysis to prevent accidental corruption of data**
  (type-checking and data hiding)
- **Support object-oriented style of programming**

Compatibility with C

How is C++ *not* backward compatible with C (C89)?

C++ does not allow

- **old-style C function declarations**
```
void f(a) int a; {}
```
- **generic function declarations**
```
void f();
void g() { f(2); }
```
- **setting enum to int**
```
enum Dir {Up, Down};
Dir d=1;
```
- **multiple declarations**
```
int i;  int i;
```
- **assigning to void ***
```
int* p = malloc(10);
```
- **"implicit int"**
```
signed a = 7;
```

Other differences:

- **const global variables have internal linkage in C++, external in C**
- **extra keywords in C++**
```
void main()
{ int catch = 5; }
```
- **bizarre comments**
```
int f(int a, int b)
{
   return a//**/b
;
}
```

(For these, C++ *is* backward compatible with C99)

Purpose of a programming language

- Programming languages serve two purposes:
  - vehicle for specifying actions to be executed "close to the machine"
  - set of concepts for thinking about what can be done
    "close to the problem being solved"
- Object-oriented C++ excels at both

Learning C++

- **Goal: Don't just learn new syntax, but become a better programmer and designer; learn new and better ways of building systems**
- **Be willing to learn C++'s style; don't force another style into it**
- **C++ supports *gradual learning***
  - **Use what you know**
  - **As you learn new features and techniques, add those tools to your toolbox**
- **C++ supports variety of *programming paradigms***

Programming paradigms

- ***procedural* – implement algorithms via functions (variables, functions, etc.)**
- ***modular* – partition program into modules (separate compilation)**
- ***object-oriented* – divide problem into classes (data hiding, inheritance)**
- ***abstract* – separate interface from implementation (abstract classes)**
- ***generic* – manipulate arbitrary data types (STL: containers, algorithms)**

What is object-oriented?

- Encapsulation
  "black box" – internal data hidden

© SDC

- Inheritance
  related classes share implementation
  and/or interface
- Polymorphism
  ability to use a class without knowing its type

"C++ is an object-oriented language" =
C++ provides mechanisms that support object-oriented style of programming

Some C++ concepts

- constructor / destructor / copy constructor
- initialization list
- inheritance
- exceptions
- overloading operators (e.g., assignment operator)
- namespace
- const
- virtual function
- pure virtual (abstract) function
- friend
- template
- standard template library (STL)
- pass by value, pass by reference
- composition versus derivation

A simple C++ program

```
#include <iostream>  // std::cout
#include <cstdio>   // printf

int main()
{
      int a = 5; // 'a' is L-value
      float b = 0.9f;
      printf("Hello world %d %3.1f \n", a, b);
      std::cout << "Hello world" << a << " "
            << b << " " << std::endl;
      return 0;
}
```

Declarations and definitions

- **Declaration:**
  - `extern char c;`
  - `struct User;`
  - `double sqrt(double);`

  → **Okay to have many**

- **Definition:**
  - `char c;`
  - `int count = 1;`
  - ```
    double abs(double a) {
      a>0 ? a : -a;
    }
    ```

  → **Must have exactly one**
    ```

Fundamental types

INTEGRAL

ARITHMETIC

USER-DEFINED

- **bool (true ←→ 1, false ←→ 0)**
- **char (could be signed or unsigned – implementation-defined)**
- **int (signed by default)**
- **double**
- **void ("pseudo-type")**
- **enum**
- **class**
- **also short, long, struct, float, wchar_t, etc.)**

Do not rely on sizes of these!
(Implementation-dependent)

Macros

- **Dangerous:**
  - **compiler never sees them**
    source code → translation unit
  - **global**
- **Instead, use**
  - **const**
  - **inline**
  - **template**
  - **enum**
- **Ok to use for *include guards* ("header wrappers")**
- **If you must use a macro, give it a long ugly name with lots of capital letters**

Example:

```
template<typename T>
inline T max(T t){
   t>0 ? t : -t;
}
```

Memory allocation

- "on the stack"
  - block delimited by {}
  - object alive till it falls out of scope
  - calls constructor / destructor
- "on the heap"
  - new and delete replace malloc, calloc, free
  - object exists independently of scope in which it was created
  - also "on the free store" or "allocated in dynamic memory"
  - be careful: new → delete, new[] → delete[]
  - for safety, same object should both allocate and deallocate
- "local static store"

```
void foo() {
        static int i=0;
}
```

Global variables

- **Built-in types initialized to 0**
  (but local variables uninitialized)
- **Initialized before main() invoked**
- **Initialization order:**
  - **within translation unit, same as definition**
  - **between translation units, arbitrary order**

Bad!
No guarantee that `twopi` will be initialized correctly

<u>**file1.cpp**</u>

```
double pi = 3.14;
```

<u>**file2.cpp**</u>

```
double twopi = 2*pi;
```

A class

member functions (methods)

member variables

```
class Date {
public:
        enum Month {Jan, Feb, Mar, ...}
        Date(int year, Month month, int day);
        int GetDay() const;
        void SetDay(int day);
        Date& operator+=(int days);
private:
        Month m_month;
        int m_year, m_day;
};
```

Struct vs. class

- In C++, no difference b/w struct and class (except default public vs. private)
- In C++, struct can have
  - member variables
  - methods
  - public, private, and protected
  - virtual functions
  - etc.
- Rule of thumb:
  - Use struct when member variables are public (just a container)
  - Use class otherwise

OO in C

- **In C, a struct can have both member variables and methods:**

```
void CreateFoo()
{
}

struct Foo
{
  void (*Construct)();
  int m_data;
};

int main()
{
  struct Foo a;
  a.Construct = &CreateFoo;
  a.Construct();
}
```

- **In C++, syntax is simpler:**

```
struct Foo
{
  Foo()
  int m_data;
};

Foo::Foo()
{
}

int main()
{
  Foo a;
}
```

Names

- **Maintain consistent naming style**
  - **long names for large scope**
  - **short names for small scope**
- **Don't start with underscore; reserved for special facilities**
- **Avoid similar-looking names: l and 1**
- **Choosing good names is an art**

Access control

- `Public:` visible to everyone
- `Private:` visible only to the implementer of this particular class
- `Protected:` visible to this class and derived classes
- Good rule of thumb:
  - member functions (methods):
    - if non-virtual, then `public` or `protected`
    - if virtual, then `private`
  - member variables should be `private` (except in the case of a struct)

The big four

- **By default, each class has four methods:**
  - **constructor** `Date();`
  - **destructor** `~Date();`
  - **copy constructor**
    `Date(const Date& other);`
  - **assignment operator**
    `Date& operator=(const Date& other);`
- **These call the appropriate functions on each member variable**
- **Be careful: If this is not what you want, then either override or disallow (by making private)**

Constructor and destructor

- (Copy) constructor creates object
- Destructor destroys ("cleans up") object
- Be aware of *temporary objects*

```
class Matrix {
        Matrix(const Matrix& other);
        Matrix operator+(const Matrix& other) const;
        Matrix& operator=(const Matrix& other);
};

void foo() {
        Matrix a, b, c, d;
        a = b + c + d;
}
```

What functions get called?

(Note: There are ways to speed this up while preserving the syntax)

Example

Suppose we have a simple class.

```
class A {
public:
   A()                                { printf("con\n");     }
   A(const A& other)                  { printf("copycon\n"); }
   ~A()                               { printf("des\n");     }
   A& operator=(const A& other)       { printf("assign\n");
                                        return *this;
                                      }
};
```

Example 1

What is the output of the following program?

```
01 {
02      A a;
03      A* b = new A();
04      *b = a;
05      delete b;
06      A c = a;
07 }
```

```
02 con
03 con
04 assign
05 des
06 copycon
07 des
07 des
```

Example 2

What is the output of the following program?

```
01 void F(const A& f, A* g, A h)
02 {
03     *g = f;
04 }

05 {
06     A a, b;
07     F( a, &b, a);
08 }
```

```
06 con
06 con
01 copycon
03 assign
04 des
08 des
08 des
```

Example 3

What is the output of the following program?

```
01 A F()
02 {
03     A tmp;
04     return tmp;
05 }

06 {
07     A a = F();
08 }
```

```
(VC++ 6.0 -- Windows)
03 con
07 copycon
05 des
08 des
```

```
(g++ 3.4.3 -- Linux)
07 con
08 des
```

Avoid new and delete

- Whenever possible, avoid ‘new’ and ‘delete’
- Instead create object on stack
- Automatic destructor makes things easier
- No need to worry about forgetting to delete the object (memory leak) or deleting the wrong object (crash)
- If you must use ‘new’, then try to keep the ‘delete’ nearby
- This helps code maintenance – otherwise it is hard to keep track of the new/delete pairs

When to use new and delete

- Sometimes you have to use new and delete
- And sometimes the pair cannot be close together
- Oh well
- The next slide shows an example where we need to break both of these rules

An example of new/delete

- **You have a base class:**
```
class Command { virtual DoSomething(); };
```
- **You have several derived classes:**
```
class CommandAdd : public Command {};
class CommandMove : public Command {};
class CommandSet : public Command {};
```
- **You have a list of objects whose types are unknown at compile time (polymorphism):**
```
std::vector<Command*> undo_list;
```
- **Must put pointers in list – not actual objects – because the objects may be of different sizes (among other reasons)**
- **Someone creates the object and puts its pointer on the list:**
```
undo_list.push_back( new CommandAdd() );
```
- **Later the object is removed from the list and deleted:**
```
Command* com = undo_list.back();
undo_list.pop();
com->DoSomething();  // call a virtual method
delete com;
```

Initializer lists

Assign values inside constructor:

```
Matrix::Matrix(const Matrix& other)
{
        m_n = 0;
        m_a = 0;
}
```

Use initializer list:

```
Matrix::Matrix(const Matrix& other)
        : m_n(0), m_a(0)
{
}
```

Concrete classes

- **A *concrete class***
  - **does a single, relatively small thing well and efficiently**
  - **hides data members (encapsulation)**
  - **provides clean interface**
  - **acts like a built-in type**
  - **is a “foundation of elegant programming” – Stroustrup**
- **Don’t underestimate the importance of this basic C++/OO feature!**

Class relationships

- **OK:**
  - **A calls function from B**
  - **A creates B**
  - **A has a data member of type B**
- **Bad:**
  - **A uses data directly from B (without using B's interface)**
- **Even worse:**
  - **A directly manipulates data in B**

Pointers, arrays, references

- Use `0`, not `NULL` (stronger type checking)
- Name of array is equivalent to pointer to initial element
- Access array using `*` or `[]`; same efficiency with modern compiler
- use `std::vector`, not built-in array, when possible
- Reference is like a pointer

References

- **Reference: alternate name for an object (alias)**
- **There is no null reference**
- **No reference to a temporary**
- **Syntax confusing**
- **Basically a const dereferenced pointer with no operations**

`int b; int &a = b;`

(Now use ‘a’ as ‘b’)

~~`int &a;`~~

~~`int& a = 1;`~~

`int* c = &a;`

“get address of” (not a reference)

Confusing syntax

```
int a, b;

int c = a * b;

int* d = &a;

int e = *d;

int& f = a;
```

* means

- multiplication, *or*
- pointer, *or*
- dereference pointer

& means

- get address of, *or*
- reference

Same symbol, different meanings!

Pass by X

pass by value

pass by pointer

pass by reference

```
void f(int a, int* b, int& c)
{
        // changes to a are NOT reflected outside the function
        // changes to b and c ARE reflected outside the function
}
```

```
main()
{
        int a, b, c;
        f(a, &b, c);
}
```

DOES make a copy

does NOT make a copy

PBP and PBR are *different* syntax for the *same* functionality

Argument passing / return

- **Pass / return by value**
  - **calls copy constructor**
  - **ok for built-in types**
    ```
    int foo(int a) { return 0; }
    ```
  - **performance penalty for structs and classes (*temporary objects*)**
- **Pass / return by reference or pointer**
  - **does not call copy constructor**
  - **pass inputs by const reference**
  - **never pass inputs by "plain" reference**
    ```
    void update(int& a);  update(2);  // error
    ```
  - **pass outputs by pointer**
    ```
    int x = 1;  next(x);  // should not change x
    int x = 1;  next(&x);  // may change x
    ```
  - **ok to return a ref, or const ref**

C++ function mechanisms

- **Overloaded function names**
  - **Cleaner and safer**
    ```
    print(int);
    print(float);
    ```
  - **But beware**
    ```
    print(int); print(int*); print(0);
    ```
- **Default parameters**
  ```
  void print(int a, int b=0, int c=0);
  ```
- **Operator overloading**
  ```
  Matrix& operator+=(const Matrix& other);
  ```
- **Implicit conversion operator**
  ```
  operator int() const {}  // converts to int
  ```
  - **Provides convenient syntax, but potentially dangerous so use sparingly**

Opaque pointers

- **An *opaque pointer* is used to hide the internal implementation of a datatype**
- **Also called Pimpl (pointer to implementation) idiom, or Cheshire Cat**
- **Example: The *d-pointer* is the only private data member of the class and points to an instance of a struct defined in the class' implementation file**

```
class Handle {
private:
    struct CheshireCat;   // Not defined here
    CheshireCat *smile;   // Handle

public:
    Handle();             // Constructor
    ~Handle();            // Destructor
    // Other operations...
};
```

```
#include "handle.h"

struct Handle::CheshireCat {
    ...     // The actual implementation can be anything
};

Handle::Handle() {
    smile = new CheshireCat;
}

Handle::~Handle() {
    delete smile;
}
```

http://en.wikipedia.org/wiki/Opaque_pointer

Explicit type conversion

- **C++ casts**
  - `static_cast` **between 2 related types (int/float, int/enum, 2 pointers in class hierarchy)**
  - `reinterpret_cast` **between 2 unrelated types (int/ptr, pointers to 2 unrelated classes)**
  - `const_cast` **cast away constness**
  - `dynamic_cast` **used for polymorphic types Run-time type info (RTTI)**
- **Avoid casts, but use these instead of C casts**
  - **e.g., compiler can perform minimal checking for** `static_cast`**, none for** `reinterpret_cast`

Namespaces

- **Namespace expresses logical grouping**
- **`using` declaration**
  - **Don’t use global `using` except for transition to older code**
  - **Ok in namespace for composition**
  - **Ok in function for notational convenience**
- **Namespaces are open**
- **Unnamed namespaces restrict code to local translation unit**
- **Aliases (namespace ShortName = LongName;)**

Const

- **Const prevents object from being modified (orig., `readonly`)**
- **Avoid magic numbers**
```
char a[128];
const int maxn = 128;
char a[maxn];
```
- **Logical constness vs. physical constness**
- **Const is your friend; use it extensively and consistently**
- **can cast away constness, but be sure to use `mutable`**
- **const pointers:**
  - **const int * const ptr = &a[0]; // const ptr to a const int**
  - **int const * const ptr = &a[0]; // ”**
  - **int * const p2 = &a[0]; // const ptr to an int**
  - **const int * p1 = &a[0]; // ptr to a const int**
  - **int const * p2 = &a[0]; // ”**

Assert macro

- Assert allows the programmer to explicitly type assumptions about expected inputs and values
- Use assert generously; it is your friend
- Assert helps to catch bugs early during development
- Assert is removed by precompiler before final release, so no run-time penalty
- Use assert only to check values; do not change values!!!

```
#include <assert.h>
int GetValue(int index)
{
    assert(index >= 0 && index < array.size());
    if (index < 0 || index >= array.size())
        return -1;  // value should make sense
    return array[index];
}
```

If performance is not a concern, then it is okay to augment (but not to replace) assert with an extra check that will remain in the final version.

Inheritance

- **Subclass derived from base class**
- **Two classes should pass the “ISA” test:**
  <u>derived class</u> is a <u>base class</u>

```
class Shape {
};
class Circle : public Shape {
};
```

- **Class hierarchy: means of building classes incrementally, using building blocks (subclass becomes base class for someone else)**
- **Facilitates code reuse**

Inheritance vs. composition

- **Inheritance: “is a”**

```
class Circle : public Shape {
};
```

- **Composition: “has a”**

```
class Circle {
private:
     Shape m_shape;
};
```

- **Decision should be based on commonality of interface**

Virtual functions

- **Function of derived class is called even if you have only a pointer to the base class**

File.h

```
class Shape
{
  virtual void Draw();
};

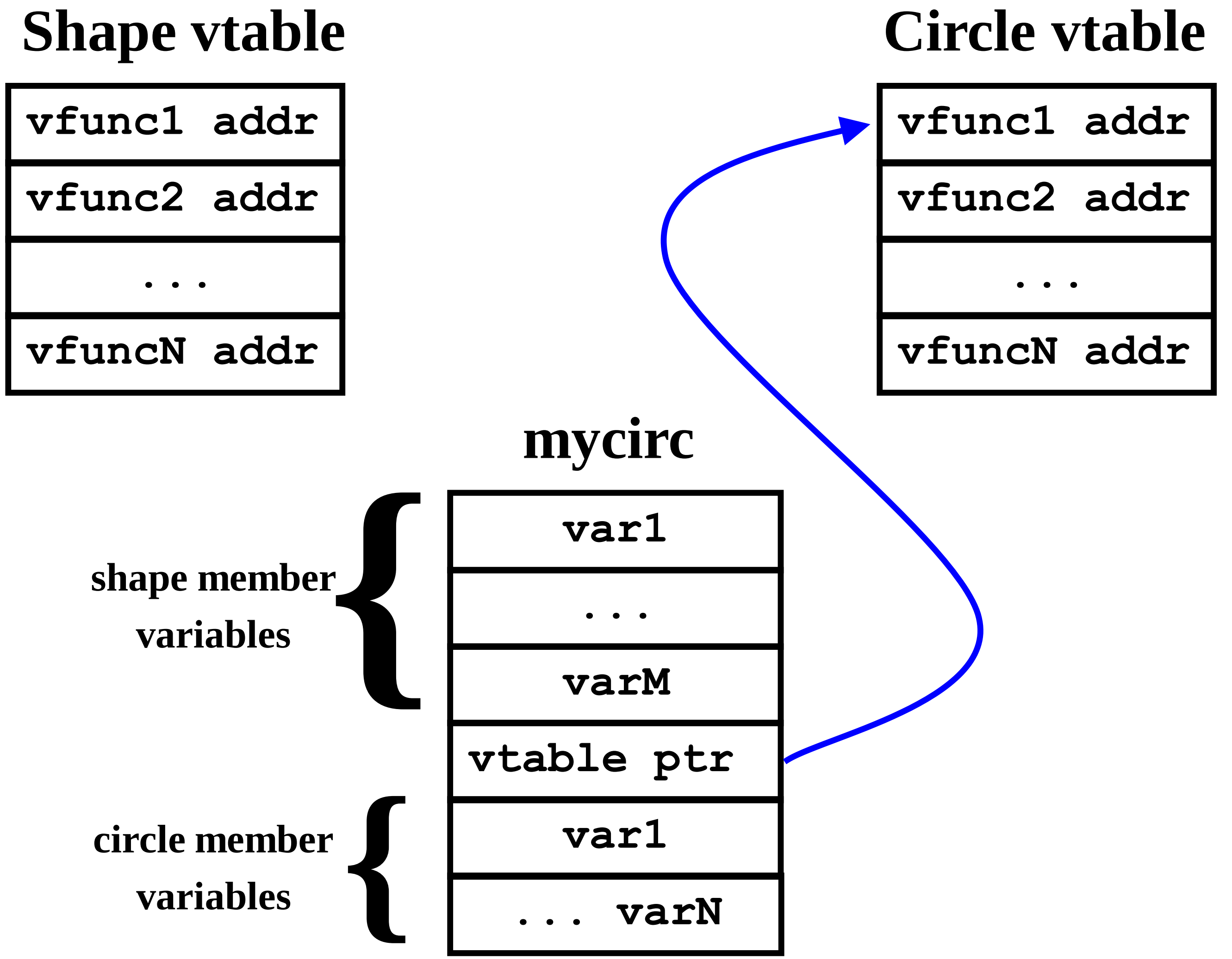
class Circle : public Shape
{
  virtual void Draw();
};
```

File.cpp

```
void Func1()
{
  Circle mycirc;
  Func2(&mycirc);
}

void Func2(Shape* s)
{
  s->Draw(); // calls Circle::Draw()
}
```

How a virtual function works



What is the penalty of a virtual function?

- **Space:**
  - **one vtable per class with virtual function(s)**
  - **one pointer per instance**
- **Time:**
  - **one extra dereference if type not known at compile time**
  - **no penalty if type known at compile time (ok to `inline` a `virtual` function)**

Pure virtual function

- **Pure virtual function**
  - **Function intentionally undefined**
  - **Same penalty as regular virtual function**
- ***Abstract class***

```
class Shape {
  virtual void Draw() = 0;
};
```

  - **Contains at least one pure virtual function**
  - **Cannot instantiate; must derive from base class and override pure virtual function**
  - **Provides an interface (separates interface from implementation)**
- **Advice: virtual functions should always be pure virtual**
  - **i.e., “Make non-leaf classes abstract” (Scott Meyers, Item 29)**
  - **Also, “Don’t derive from concrete classes” (Herb Sutter, p. 137)**
- **More advice: Make virtual functions private** (Herb Sutter, p. 134). This separates the override implementation details from the public interface.

Multiple inheritance

- C++ allows you to inherit from multiple base classes
- Works best if
  - exactly one base class passes ISA test
  - all other base classes are interfaces
- Advanced feature that is rarely needed

```
class MyDialog :
   public CDialog, Observer {};
```

MyDialog is a CDialog

MyDialog needs a single method from Observer (lightweight class) (see MVC architecture)

Polymorphism

- **Polymorphism**
  - **"ability to assume different forms"**
  - **one object acts like many different types of objects (e.g., `Shape*`)**
  - **getting the right behavior without knowing the type**
  - **manipulate objects with a common set of operations**
- **Two types:**
  - **Run-time (Virtual functions)**
  - **Compile-time (Templates)**

Exceptions

- **Error handling in C:**
  - **Half of code is error handling**
  - **Dangerous: Easy for programmer to forget to check return value**

```
void Func() {
  int ret;
  ret = OpenDevice();
  if (ret != 0)  error("Unable to open device");
  ret = SetParams();
  if (ret != 0)  error("Unable to set params");
}
```

Exceptions (cont.)

- **Error handling in C++:**
  - try-catch blocks safer
  - separate “real code” from error handling code

```
void Func() {
        try {
                OpenDevice();
                SetParams();
        } catch (const MyException& e) {
                e.ReportToUser();
        } catch (...) {
                abort(1);
        }
}

void OpenDevice()
{
        if (bad)  throw MyException(“Cannot open device”);
}
```

Templates

- **Define a class or function once, to work with a variety of types**
- **Types may not be known until future**

```
template<typename T>
T Max(T a, T b) { return a>b ? a : b; }

template<typename T>
class Vector {
  Vector(int n, T init_val);
  T* m_vals;
};
```

- **Better type checking and faster (cf. `qsort`)**
- ***Specialization* can be used to reduce code bloat**
- **Templates support *generic programming***

Generic programming

- **Drawbacks of qsort in <stdlib.h>**
  - **requires a compare function, even if trivial**
  - **loss of efficiency b/c dereferencing pointer**
  - **lost type safety b/c void***
  - **only works with contiguous arrays**
  - **no control over construction / destruction / assignment; all swapping done by raw memory moves**

Standard Template Library (STL)

- **Containers:**
  - **Sequences**
    - **vector – array in contiguous memory (replaces realloc)**
    - **list – doubly-linked list (insert/delete fast)**
    - **deque (“deck”) – double-ended queue**
    - **stack, queue, priority queue**
  - **Associative**
    - **map – dictionary; balanced tree of (key,value) pairs like array with non-integer indices**
    - **set – map with values ignored (only keys important)**
    - **multimap, multiset (allow duplicate keys)**
  - **Other**
    - **string, basic_string – not necessarily contiguous**
    - **valarray – vector for numeric computation**
    - **bitset – set of N bits**

STL (cont.)

- **Algorithms (60 of them):**
  - **Nonmodifying**
    - **find, search, mismatch, count, for_each**
  - **Modifying**
    - **copy, transform/apply, replace, remove**
  - **Others**
    - **unique, reverse, random_shuffle**
    - **sort, merge, partition**
    - **set_union, set_intersection, set_difference**
    - **min, max, min_element, max_element**
    - **next_permutation, prev_permutation**

std::string

- **Example:**

```
#include <string>

void Func()
{
  std::string s, t;
  char c = 'a';
  s.push_back(c);  // s is now “a”;
  const char* cc = s.c_str();  // get ptr to “a”
  const char dd[] = "afaf";
  t = dd;  // t is now “afaf”;
  t = t + s;  // append “a” to “afaf”
}
```

std::vector

- **Example:**

```
#include <vector>
void Func()
{
  std::vector<int> v(10);
  int a0 = v[3];  // unchecked access
  int a1 = v.at(3);  // checked access
  v.push_back(2);  // append element to end
  v.pop_back();  // remove last element
  size_t howbig = v.size();  // get # of elements
  v.insert(v.begin()+5, 2);  // insert 2 after 5th element
}
```

std::vector (cont.)

- **Example:**

```
#include <vector>
#include <algorithm>
void Func()
{
  std::vector<int> v(10);
  v[5] = 3;  // set fifth element to 3
  std::vector<int>::const_iterator it
               = std::find(v.begin(), v.end(), 3);
  bool found = it != v.end();
  if (found) {
    int three = *it;
    int indx = it - v.begin();
    int four = 4;
  }
}
```

Iterators

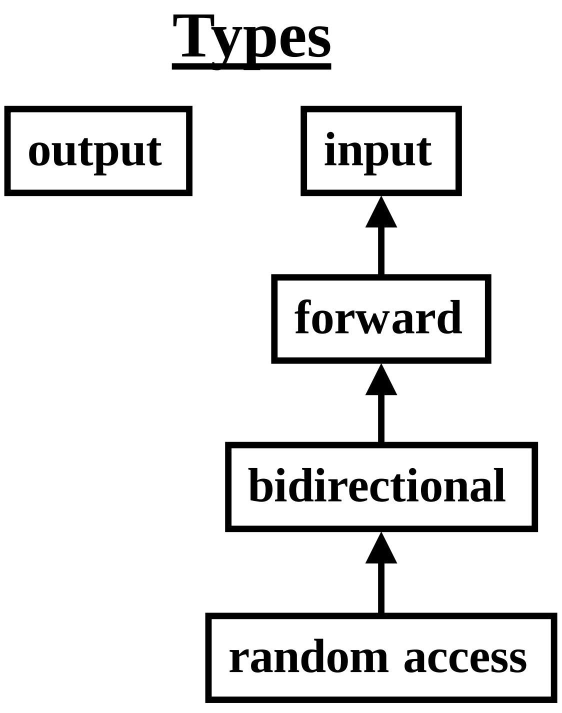
- ***iterator* – generalized pointer**
- **Each container has its own type of iterator**

```
void Func() {
  stl::vector<int> v;
  stl::vector<int>::const_iterator it = v.begin();
  for (it = v.begin() ; it != v.end() ; it++) {
    int val = *it;
  }
}
```

Types of iterators

```
template<class InputIterator, class Type>
   InputIterator
   find( InputIterator _First,
         InputIterator _Last,
         const Type& _Val );
```

- **Each container provides a different type**



Allocators

- **STL written for maximum flexibility**
- **Each container has an *allocator***
- **Allocator is responsible for memory management (new/delete)**

```
template < class Type,
           class Allocator = allocator<Type> >
class vector {
   ...
};
```

- **Advice: Ignore allocators**

Streams

- **C**
  - **flush, fprintf, fscanf, sprintf, sscanf**
  - **fgets, getc**
- **C++**
  - **cout, cin, cerr**

Buffer overrun

- **Never use *sprintf*!**
- **Use snprintf instead to avoid buffer overrun**
- **Or use std::stringstream**

Numerics

- **valarray**
  - **matrix and vector (not std::vector)**
  - **slices and gslices**
- **complex**
- **random numbers**