Coding guidelines

*These are guidelines for CSCI121 Spring Session 1997. They do **not** have "departmental status". Lecturers for other subjects require different styles. Confirm with lecturer when starting a subject.*

These guidelines contain internal cross references indicated by underlining.

Program

"Physically" a program is built by compiling and linking together:

- a file with the main program (usually named main.cpp)
- a number of separate modules (X.h/X.cpp file pairs)
- functions taken from standard libraries.

Module

A module is a separately compilable part of a C/C++ program. A module will consist of two files – the "header" file and the "implementation" file. The header file defines the services that the module provides to the rest of the program. The implementation file contains the code that is used to supply those services.

There are different styles of module. These suit different approaches to program design.

A module for a program designed using top-down functional decomposition:

The header file will contain:

- A list of function prototypes and comments summarizing the functions provided (using a term from another programming language, you could say that these are the functions that are "exported" by the module).
- Constants and typedefs might also be in the header file; and sometimes there may even be struct declarations. However, these should be minimized.

The implementation file will contain:

- Declarations of any structs used only by functions defined in that module.
- Definitions of any constant data used by those functions.
- Definitions of any static (filescope) variables used to retain state information between successive calls to functions "exported" by the module.
- Definitions of the functions listed in the module's header file (the "exported" functions).
- Definitions of any static auxiliary functions needed to simplify implementation of "exported" functions.

A module for a program designed using an object based (or object oriented) approach

A module for an object based program will either define a single class, typically one representing an abstract data type such as a "PriorityQueue" that may bet used in many different programs, or a group of closely collaborating classes whose instances are always used together (a simple example would be a module that defines class List and class ListIterator).

The header file would contain:

- class declaration(s),
- Definitions of "inline functions",
- Declarations of any "free functions" related to the class;
- and, of course, some comments explaining the role that the class's instances fulfil;

The implementation file would contain:

- Declarations of auxiliary structs or even a declaration and definition of an auxiliary class (e.g. class BinaryTree might have class TreeNode declared and defined in its implementation file);
- Definitions of all member functions of the class(es) apart from those defined by inline functions in the header;
- Definition of any associated free functions;
- Definition of any class data members (static data members)

Definition and declaration

In C++:

- a "forward" declaration
  introduces the name of a function, struct, enum or class.
- a declaration
  introduces a variable, or specifies the members of a struct or class
- a definition
  allocates space for a variable, provides code for a function, introduces name and allocates space and value for a constant.

By default, a variable declaration is also its definition. The following statements all instruct a compiler to record a variable name in its symbol table and arrange allocation of storage space. (If the variable is a global or filescope variable, that really does mean arrange allocation of storage space. If the variable is a local variable of a function, it means recalculating the size of the "stack frame" for that function and allocation of a relative position within the stack frame where the variable will be located.)

```
int  gCount;
int  gStartXCoord = 30;
char gMsg[] = "Hello World";
static int sFrequencyCount;
static char *sErr = "Incorrect number of arguments on command line";
```

If you want to declare that a variable, e.g. gWorld, does exist but is *not* to be defined in the current module, you must write an extern declaration:

```
extern int    gWorld;
```

The appropriate use of "extern" variable declarations is explained later in the note concerning "global variables".

Variables can only be defined once in any scope.

A function can be declared more than once, just write down its prototype. It should have only one definition.

```
void DoThis();       // declaration
void DoThis();       // declared again, never mind
void DoThis();       // compiler getting bored reading this
void DoThis();

void DoThis() { … }  // finally the definition
```

Make sure that the prototype in a declaration matches the definition; otherwise the compiler will think that you are talking about different functions.

The name of a struct or class type may be declared any number of times, but there must be only one specification of its members. Declarations like the following ("forward declarations") simply tell the compiler that the various structs and classes will be fully declared and defined properly *somewhere*; meantime the compiler is to accept the name as the name of a data type and deal with any code that involves pointers to or references to data of these types. (If the compiler has simply seen a name declared, it can't deal with any code that tries to make use of variables of that type. The compiler must have read a complete declaration in order to deal with code that declares instances of a class or struct type, attempts to declare the form of another structure with a data member of a type for which only the name is known, or invokes member functions or accesses data members via a pointer or reference to a variable of that type.)

```
struct Thing1;              // all these are "forward declarations"
struct Thing2;
class MyClass;
class MyTree;
class MyTreeNode;
```

A struct or class is declared properly when you specify its members:

```
struct Thing1 { int fXPos; int fYPos; char *fName;  };
class MyThing {
public:       MyThing ();
     int Add(void* data, int key);
     …
private:
     …
     static int      sMyThingClassCounter;
};
```

Note that a class declaration does not define any static (class) data members. These must be explicitly defined in the implementation file:

```
int MyThing::sMyThingClassCounter = 0;
```

Class Declaration

A class declaration should have the following standard form (note, not allowing for inheritance and "object oriented" styles):

```
class MyClass {
public:
     // First, constructors
     //  (some classes have none --- or more accurately they use a default
     //              constructor provided by the compiler)
     // Most classes have several --- corresponding to different ways
     //   to initialize instances of the class
     MyClass();
     MyClass(const MyClass& other);
     …
     // Possibly a "destructor"
     //   not all classes require a destructor
     ~MyClass();
     // Public member functions grouped in categories
     //    access functions --- that simply get info from instance
     //      of class --- should be declared const
     int      NumItems() const;
     const char *Name() const;
     …
     void ChangeName(const char* newName);
     …
     // Standardize names of functions for input and output of class
```

```
     //  instance
     void     ReadFrom(istream& input);
     void     WriteTo(ostream& output) const;
     …
     // You may have "static" (class) public member functions,
     //    but probably you won't
     static int Counter();
     // If your class insists on having friends, name them here
     friend class MyOtherClass;
private:
     // private auxiliary functions that help implement the public
     //    functions
     int      AuxDoAdd(...)
     …
     // data members
     int      fIdentifier;
     char     *fName;
     …
     // possibly some static class data
     static char *sNameInfo[];
};

// inline functions here
const char *MyClass::Name() const { return fName; }

// declare (maybe define) any associated free functions
inline istream& operator>>(istream& input, MyClass& myc)
     { myc.ReadFrom(input); return input; }
inline ostream& operator<<(ostream& output, const MyClass& myc)
     { myc.WriteTo(output); return output; }
```

Inline functions are primarily a minor optimisation feature. As a general rule, only define a function as inline if it is a "one-liner" such as a function providing read access to a private data member.

Function prototype

A function prototype specifies:

- name of function
- type of result
- types of arguments
- if the function is a member of a class, there may be an additional const qualifier (indicating whether the function can change the instance for which it was invoked)
- if exceptions are being used, there may be an additional exception specification (exceptions are not covered in CSCI121).

Some lecturers insist that function prototypes should give solely the types of arguments:

```
void MoveTo(int, int);
```

Others like the function prototypes to identify the arguments:

```
void MoveTo(int horiz, int vert);
```

Check your lecturer's preferences, or lose marks. For CSCI121, you will specify the names of arguments in function prototypes.

Argument names given in a function declaration don't have to match those used in the actual definition. So you could, if you were stupid, declare MoveTo() with arguments horiz, and vert then define it to use arguments x and y:

```
void MoveTo(int x, int y) { … x … … y }
```

See also "function interface" and "functions that return error codes" for further comments on functions.

Header file

As noted earlier, a header file defines the "services" (functions, classes, whatever) provided by a separately compilable module. Header files are #included in modules that are "clients" for those services. If you want to use a function defined in a different module (possibly one of the standard library modules) you must #include the appropriate header file in your own module (possibly in your .h header file, possibly in your .cpp implementation file – a little more advice given later).

In general, a header file can contain:

- #include directives on other header files
- function prototypes
- inline function definitions
- struct declaration
- class declaration
- const definitions
- enum declarations
- typedef statements
- extern variable declarations

Although legal and often quite appropriate, const enum typedef and extern variable declarations can all cause problems with large projects and their use should be minimized.

A header file must NOT contain:

- variable definitions – either global or static
- function definitions other than those explicitly or implicitly declared as inline

A variable definition in a header file is always an error. The compiler/linker system will pick up the error if the variable has global scope (it will get multiply defined, once for each implementation file that #includes the header). A variable declared in a header as having static scope is a more subtle problem. It won't be picked up by the compiler or linker. Each compiled module that #includes the header will have its own separate copy of the variable. The programmer probably thinks that there is a common shared copy and consequently can't understand why the program doesn't run correctly.

The appearance of a function definition in a header file will result in multiple copies of the code being generated and a link error. (Of course, inline function definitions are an exception to this rule.)

Although all the elements listed above can appear in any header file, the type of elements mainly used will vary according to whether the header is for a program designed by top down functional decomposition or for an object based program. A header file should follow the standard format; these are shown below for both styles (along with an example of how not to write a header file).

Example 1:

This is for a module in a program designed using top-down functional decomposition. It had a component that stores data records in a disk file. It defines a set of functions that can be used by the rest of the program and has a structure declaration that specifies the kind of data that should be passed to these functions. Note how the author has adopted a consistent naming style.

datamanager.h

```
#ifndef DATAMAN_H
#define DATAMAN_H
/*
Header for datamanager component that keeps
  records in name filed.  Provides access to keyed records
  as described in project docs.
*/
struct dtmn_data {
     void     *p_data;
     int      datasize;
```

```
};

struct dataman; // for "opaque" pointer to a datamanager

// open returns NULL if couldn't open the file, otherwise
//    it is a pointer to the datamanager that works with the named file
dataman *dtmn_open(const char *filename);
void dtmn_close(dataman *manager);
// "get" tries to load record with specified key,
//   it allocates space on heap for actual data and puts
//   size and address in record passed as argument
//   return 0 if all OK, non-zero error codes see main docs.
int dtmn_get(dataman *manager, long key, dtmn_data& record);
// "save" puts data described by record into file
//   if there is already an entry with that key, the old entry is replaced
void dtmn_save(dataman *manager, long key, const dtmn_data& record);
// "delete", remove entry, with specified key, from file
void dtmn_delete(dataman *manager, long key);

#endif
```

Example 2

The following illustrates **poor** style for a module.

The header describes a set of apparently unrelated functions. The module lacks the conditional bracketing that should always be present. The naming styles are inconsistent. The compiler will not be able to deal with this header if the iostream header is not #included before this one. The function signatures show lack of consistency with respect to passing of arguments.

stuff.h

```
const int RecordSize = 53;
const int SizeRecrd = 34;
struct Record { int l;
char name[71]; double DataValue;
long Intvalue;       char thing[SizeRecrd];
     char Thing[RecordSize];};
int StopIt(int, int, int, int);
void WhenReady(double);
void PrntRecord(ostream&, Record);
void Readrec(Record&, istream&);
int chck_rcrd(Record*);
void Matrixmultply(double m[38][38], double n[38][38],
double a[38][38]);
void i(double l[38][38]);
typedef long Quendop;
const max = 103;
const int SzeRecrd = sizeof(Record);
#define speedygonzales
Quendop Fudge(Quendop);
void foo(int*, int&, int i=max);
```

Don't ever write anything like that! A module is supposed to represent a recognizable component part of a program, not a heap of junk and odd and ends.

Example 3:

Here is a header for a module that defines a single class, MyClass. A unique name, here MYCLASS_H, shown in bold should be made up to identify the module:

myclass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H

/*
Block comment explaining what this class provides to rest of program.
```

```
*/

#ifndef _STRING_H
#include <string.h>
#endif

class ostream;
class MyClass {
public:
     …
     int       CheckName(char* str) const;
     void      WriteTo(ostream& out) const;
     …
private:
     char      *fName;
     …
};

inline int  MyClass::CheckName(char *str) const { return 0==strcmp(str,fName); }
#endif
```

The entire contents of a header file should be bracketed by the #ifndef, that tests whether this header has already been seen by the compiler, and matching #endif (essentially this prevents mistakes that would occur if the same header file was #included more than once by a client module).

Here there is a conditional include of <string.h> (the compiler will read the system file string.h at this point if this file has not previously been read). This is necessary because later, with the inline function definition for CheckName(), we have something in the header file that depends on the contents of string.h.

Next, there is a simple "forward" declaration of the class name ostream. This is necessary as there are references to ostream objects in the function prototypes of MyClass. The compiler must be assured that ostream is a valid name of some type of data object. We don't need to #include <iostream.h> as there is nothing in this header that depends on any property of an ostream object (we simply need to know that they exist). But, of course, if we don't #include iostream.h here, the implementation file for MyClass will need to #include it so that the compiler can check any output operations that we invoke in the code of MyClass::WriteTo().

Inline function definitions, if any, should come at the end of the header file.

Leave some blank lines after the #endif (this avoids the possibility of an obscure bug that can otherwise occur when shifting files from PC or Mac to Unix).

Implementation file

As with header files, there are some variations in the prototypical form for an implementation file for a program using functional decomposition and one using classes.

Implementation file for top-down functional decomposition

datamanager.cpp

```
/*
Block comment
     more details about module
*/

// next get includes on header files
#include <stdlib.h>
#include <fstream.h>
#include …
#include "datamanager.h"

// watch out for "header ate my file" bug --- a header that has a mistake
// like an incorrectly matched block comment bracket, or even a missing
// blank line after final #endif, can cause compiler to ignore all subsequent
```

```
// input to end of file!

// next declare any const values,
// then if there are auxiliary structs used solely within this
// module get them declared

const int kBUFFERSIZE = 256; // max allowed for input line
…

struct dataman {
     fstream  fMainFile;
     fstream  fIndexFile;
     …
};

// Globals??????
//    Any Gobals should be defined in
//      a special "globals" module with a header file that has a
//      set of extern data declarations and an implementation file
//      that has just a set of variable definitions.
// In rare cases, where there are only one or two global variables
//   these can be define in main.cpp and referenced via explicit
//   external declarations in other modules.

extern int gErrorNumber;  // defined in main.cpp

// In even rarer cases, a module other than main.cpp or globals.cpp can
//   define global variables that are referenced from other modules
//   The header file for the defining module should contain the extern
//   declaration for these "exported" variables
// int aValueWeMightExport;  // put extern declaration in header
                     // if it is really needed as global


// Filescope variables
//   These are variables that maintain state data between successive calls
//  to functions in this module.

static int sCountOfFilesNowOpen = 0;

// If you will be defining auxiliary static filescope functions
// then list their declarations here (definitions later)

static long ComputeHashValue(const char *source);
static long KeyToAddress(long key);

// Now get the function definitions
//   Order?  Just something reasonably logical
// Repeat the static linkage specifier on any functions that are supposed
// to be filescope.

dataman *dtmn_open(const char *filename)
{
     …
}

…

static long KeyToAddress(long key)
{
     …
}
```

Implementation file for classes

MyClass.cpp

```
/*
Block comment
     more details about class
*/

// next get includes on header files
#include <stdlib.h>
#include <iostream.h>
…
#include "MyClass.h"


// const values,

const int kMSGSTRING[] = "Hello World";
…

// then if there are auxiliary structs (or classes) used solely within this
// module get them declared (& defined if a class with member functions)

struct KeyPtrPair { void *fdata; int fkey; };

// Globals??????
//    Gobals in an object based program – no.
//   There are conventions that you will learn later, such as
//   singleton classes, that you can use in place of globals.
//   For now, don't even think about them in class based modules

// Filescope variables????
//   Rare, mostly they are replaced by static class members
//   but there are occasions when filescopes can be used.
//   For now, forget them in class based  modules


// If this module is for a class with "static" class data members, define them
// here.  Note, in this case you do NOT repeat the keyword static.

char *MyClass::sNameInfo[] = { "Tues", "Wed", "Thurs" };

// Auxiliary static filescope functions (not class members)
// You may have such functions even if you are using classes.  For example
// you may need code to format a string in a particular way; this not really
// a "behaviour" of your class, simply something that you need to do in
// several parts of your class's code.

static void ReformatString(const char *source, int control, char buff[], int
     maxl);

// Now get the function definitions
//  Do NOT use static qualifier if defining a static member
// function of a class.

MyClass::MyClass()
{
     …
}

…

static long ReformatString(const char*source, int control, char buff[], int maxl)
{
     …
}
```

Inline function

Define a function (in a header file) with the qualifier "inline". The compiler then has the option of coding a separate function that gets called at each point it is used, or arranging to expand the call into the actual data manipulation steps.

Primarily a minor optimisation to gain speed on simple functions. Main use will be for functions used to access values held in data members of objects.

Free function

A function that is not a member of a class is a "free function". Examples would include all the functions in the maths and string libraries. Generally, you should try to minimize the number of free functions that you define with external (global) scope.

Static auxiliary function

In a procedural style program, a "static auxiliary function" fulfils much the same role as a private auxiliary function in a class.

The functions exported by a function-style module (analogues of the public functions of a class) may be too complex to implement as single functions. Consequently, you will invent auxiliary functions that do part of their work. These auxiliary functions should only be used from within your principal functions, so you want to hide them.

You hide them by giving them static linkage, making them filescope rather than global scope. Their names disappear before your code reaches the linker – so code from other modules cannot possibly call them.

Local variables

Local automatic variables are defined within the body of a function or block. Their scope extends from the point of definition to the end of the block where they are defined. They are allocated on the stack. Their lifetime is the lifetime of the enclosing scope (block or function).

Some lecturers insist that all variables introduced in a block be defined at the start of that block. Others follow standard C++ practice and introduce variables at the point where they are first used. (Different preferences can easily cost you 0.5 marks in a 5 mark assignment.)

The following is standard C++ style (it may violate your lecturer's preferences):

```
int FillBuffer(istream& input, char buffer[], int maxlen, int& errno)
{
     /*
     Fill buffer with characters from input.
     Input must be in correct message form, starting with stx code then
     …
     Return 0 if successful.
     Return 1 if error (in which case set reference variable errno to
       error number, 1 = EOF encountered,
                  2 = incorrect message structure, 3 =
       …
     */
     errno = 1; // most likely error is end of file
     int ichar;      // use int for character, simplify eof check
     ichar = input.get();

     if(ichar == EOF) return 1;

     const char STX = 3;
     if(ichar != STX) { errno = 2; return 1; }
```

```
      // Next two characters represent length
      ichar = input.get();
      if(ichar == EOF) return 1;
      int highdigit = ichar;

      ichar = input.get();
      if(ichar == EOF) return 1;
      int lowdigit = ichar;

      int msglen = 256*highdigit + lowdigit;

      if(msglen > maxlen) { errno = 2; return 1; }

      // have to do checksum on data, initialize to 0
      long checksum = 0;

      // Use ptr when filling in destination array
      char *ptr = buffer;
      // read characters
      for(int i = 0; i<msglen; i++, ptr++) {
        int nxtch = input.get();
        if(nxtch == EOF) { return 1; // end of file error }
        *ptr = nxtch;
        checksum += nxtch;
        }
      …
}
```

Here variables (and constants) are being introduced as needed. Scope of ichar (i.e. places where you can reference this variable) is almost the whole body of the function. Scope for msglen is from point of declaration to end of function. Scope of nxtch is just the little block where it is defined.

Scope of 'i' ? A problem! Most compilers don't comply with C++ standard. See comments on Limited scope variables

Limited scope variables

The proposed standard for C++ suggests:

- loop control variables, introduced in the initialization part of a for statement should have a scope that includes only the for statement and any block it controls;
- a variable can be introduced in an if statement – its scope is the code guarded by that if;

Most compilers don't yet comply; some do.

The most likely problem that you will encounter is the following:

```
for(int i=0; i < kLEN; i++) { … DoSomething(i);  … x[i] … }
…
for(int i=0; v[i] != 0; i++) { … }
```

Strictly that is legal C++. Most compilers will give an error at the second for loop "double definition of variable i".

Since you are likely to be trying to move code around (PC, Mac, Unix) and may be using more than one compiler on any single platform, it is probably better to play safe until the compilers are better standardised. Either define a reusable loop control variable separately from any loop:

```
int i;
for(i=0; i < kLEN; i++) { … DoSomething(i);  … x[i] … }
…
for(i=0; v[i] != 0; i++) { … }
```

or use a distinct loop control variable in each loop:

```
for(int i1=0; i1 < kLEN; i1++) { … DoSomething(i1);  … x[i1] … }
…
```

```
for(int i2=0; v[i2] != 0; i2++) { … }
```

The specification involving if statements should allow things like the following:

```
if((int ch = in.get()) != EOF) { … = ch … }
…
char ch;
…
```

Best to avoid such styles until all compilers agree on the meaning.

Filescope variables

Variables declared in an implementation file, and not within the body of any function, that have a "static" qualifier can only be referenced by functions declared within the rest of that file.

Such variables are used:

- when implementing an abstract data type using simply the C base language without the C++ class extensions
- to preallocate some scratch pad work area e.g. a buffer for reading characters from an input stream or doing some formatting.

Global variables

Variables declared in implementation files, outside of any function, (and not qualified by the static linkage qualifier) possess "external" or "global" scope. Such variables can be accessed and manipulated by any of the code of a program.

Although such variables may appear to be "hidden" in an implementation file, any other part of the code can simply name them in "external" data declaration statements. The linker will then dutifully tie the reference in the "foreign" module to the variable in the module where the variable is defined. This tying together of separate modules through some global variables is something you should try to avoid.

Such ties between modules are not obvious when reading the code. But they mean that when debugging or extending the code, you can't restrict your attention to a single module – the variables that you want to work with may be being changed from any other part of the program.

You should aim to minimize the number of globals that you use.

You should group all globals in a single "globals" module with a header that names the variables in extern data declarations and an implementation file that defines the variables. (Files globals.h, globals.cpp)

Later, you will learn ways of using classes to better encapsulate and control access to globals.

Constants

Use named constants rather than "magic numbers" in your code.

Define constants at the start of a file (header or implementation *as appropriate*) after any #include statements.

Adopt a consistent way of naming constants. The following convention is recommend:

small letter c or k
rest of name capitalised

e.g.

```
const int kLENGTH = 56;
const char kFILENAME[] = "input.txt";

const char *kMONTHS[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                     "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
```

What does he mean "header or implementation" as appropriate? Do your clients need to know the value of the constants? If yes put them in the header, if no put them in the implementation.

Enums

Define enums with names that make them distinctive; following illustrate preferred style:

```
enum EFontType { eTIMES, eHELVETICA, eCOURIER };
```

Main use for enums should be making function interfaces more type secure. Often you will have a function one of whose arguments is essentially an integer from a limited range, as in the following definition for a print function:

```
const int TFont = 0;
const int HFont = 1;
const int CFont = 2;

void Print(const char *str, int fonttype);
```

Operation of Print() is only defined if second argument is 0, 1, 2. But if declared like this, the compiler can't spot errors like

```
Print("hello world", 72);
```

If you changed from an integer argument to an enumerated type, the function interface would be more secure:

```
void Print2(const char* str, EFontType aFontType);
```

Compiler will complain if Print2() given anything other than eTIMES etc.

Type names

Classes and structs must be given meaningful names. An isolated code fragment given as an example can use struct thing1 or class MyClass. Your code better have struct KeyRecord and class PhoneDirectory or whatever. Typenames corresponding to structs and classes should have names that start with a capital letter; as with variables, if the name consists of multiple words you should follow a convention like capitalisation of initial letter of each word or underscore separators between words (see later in section on variable names).

You can use typedef to introduce synonyms for existing types. You will find that there are many synonyms for integers etc defined in various system header files (following examples are from Unix's stddef.h and sys/types.h):

```
typedef unsigned int        size_t;
typedef long                whcar_t;
typedef int                 ptrdiff_t;
typedef unsigned char       uchar_t;
typedef long                daddr_t;
typedef char*               caddr_t;
```

Note the systematic naming conventions, these synonym types all end with _t.

Such synonyms have a limited use; they help to clarify role of arguments to a function, e.g. void WriteTo(daddr_t position, caddr_t data, long len) is clearer than WriteTo(long, char*, long).

However they are not new types. The compiler won't object if you add a wchar_t variable to a daddr_t variable.

Function names

Functions that you define should have names that explain their role in the program (or, if member functions, the behaviour that they define for the class); generally, the names should start with capital letter and follow the convention of capitalising embedded words.

Choosing a good function name?

- A few are conventional. In a program that accepts commands from a user you should expect a central "HandleCommand()" or "MenuSelect()" function; the functions that it calls to process the inputs will have names that correspond to the user commands, such as: DoAddNewRecord(), DoFindRecord(), DoSave(), etc. A class for an abstract data type will have names that relate to the services it provides; so for example, a class Stack will have member functions Push() and Pop(). Use consistent names when implementing classes – so in all your classes where the instances can read and write their data you should have members ReadFrom() and WriteTo().

- A function (member, static auxiliary, or global free function) that returns a value should be named for the value it produces – Identifier(), Name(), Salary(). Some people like the convention that functions that return boolean values all have names that end with _p (for predicate) – so they would prefer Empty_p() to Empty() etc; if you adopt such a convention, stick to it throughout all the code of a program (don't have some functions with _p and others without).

- Names that combine a verb and a noun are often the best for conveying the role of a procedure (void function).

- A suffix or prefix Aux can be useful in identifying (private) auxiliary helper functions for a principal function (public member of class or exported function of function style module) – e.g. AuxAdd() for an Add() function.

Variable names

Conventions:

First letter(s)	Variable type
g	Global
s	Static filescope variable
f	(Instance) data member of class
s or **sc**	static data member of class
a or **the**	arguments
p or **ptr**	pointers
other lower case	local variables of functions

Local variables and arguments generally have names that start with a lower case letter. For arguments, it is common to have names starting with 'a' or 'the' e.g. void DoUpdate(SalesRec& theSalesRec, const char *aName, int theCount). If the variable is a pointer, it is wise to adopt a style where you use a consistent prefix (or suffix) such as p, p_, ptr_ that flags this fact.

Some programmers like a convention that is really more the style of the Ada language than C++. In this convention, if the value of the argument is used but not changed by a function, then the argument name starts with in_ (C++ pass by value or const reference). If the function ignores any initial value, but passes a result back (implies a C++ reference argument) then the argument's name should start with out_. If the function reads and changes the value of an argument then its name should start inout_ (again, pass by reference).

After the first letter (or multi-letter prefix) the rest of name made up with words, if more than one word, capitalise first letter of each word – gRecordCount, sTreeRoot, fXCoord, …. (You can use underscore as an alternative to capitalization, e.g. g_record_count; but stick to one style in a program.)

Names should clarify role of variable in program. It is easier to understand code that involves manipulations of "gRecordCount" than code that manipulates "x".

There are some commonly used names: i, j, k, ndx are typically used as for-loop control variables; ch is often used as name of char variable whose scope is the current block or function. A variable called 'temp' had better be defined in a block with a scope of at most half a dozen lines!

Function Interface

Miscellaneous points:

- How many arguments?
  Not many! Functions with multiple arguments (>3) are indicative of bad design.
  In old style C programs, and procedural C++ programs, you may have to pass around several arguments because they represent related data. With C++ classes, a lot of related data are packaged together as the data members of an instance of the class. The member functions of that class don't have to have these passed as arguments – these data are already directly accessible!

- Default values specified in declarations?
  Don't over do this; they can cause more trouble than benefit.

- Pass by value, pass by reference, or pass by pointer?
  Value – for things like simple integers, doubles, or anything else ≤ 8 byes.
  const reference for things that you would have passed by value but which were larger than the 8 byte limit.
  reference or pointer ? Can't be completely definitive, but here are some suggestions.

Remember a pointer and a reference are to all intents and purposes the same thing – the address of the argument. In most circumstances you could have either type as an argument. There is one difference, you can pass a NULL pointer, you can't pass a NULL reference. So, if a NULL argument is meaningful, the function must use pointer arguments.

If arguments are passed by reference, the code of the function is easier to read (you don't have all those pointer dereferencing steps):

Reference

```
void Swap(double& d1, double& d2)
{
     double temp = d1;
     d1 = d2;
     d2 = temp;
}
```

Pointer

```
void Swap(double* pd1, double *pd2)
{
     double temp = *d1;
     *d1 = *d2;
     *d2 = temp;
}
```

Most programmers find the reference style much more intelligible.

Those who favour a pointer style argue from the perspective of the calling code. They claim that the pointer style helps flag calls where arguments can be changed. The call will involve specifically passing an address rather than a value. So, if you see the address of a variable being passed to a function, you should suspect that it could get changed.

I generally favour use of pass by reference (the function prototype should use const qualifier to identify cases where the argument will not be changed).

Note. If your function is defined to take a reference, and you find you need to call a function that requires a pointer, that is easy – use the "address of" operator

```
void OtherFunction(Record* aRec);

void MyFunction(Record& theRecord)
{
     …
     OtherFunction(&theRecord);
```

```
}
```

If your function is defined to take a pointer, and you find you need to call a function that requires a reference, that is easy – "dereference" the pointer:

```
void ThatFunction(Record& dataRec);

void FirstFunction(Record *p_Record)
{
     …
     ThatFunction(*p_Record);
}
```

Functions that return error codes

There are at least four problems here – a) passing responsibility back to a calling function that can do nothing about the error, b) consistency of style, c) an unfortunate habit of programmers of returning something that could either be a result or an error flag, d) there is no way that you can make sure that someone who calls your function is going to check the error code you return.

Why are you returning an error code? You encountered some problem – have you any reason to expect that your client can do something about it? If the problem is essentially a fatal error (e.g. you tried to read data from a file and got garbage) the program will have to stop. So you might as well stop it with code that prints a message to cerr and calls exit().

Other examples of where you should just stop the program with an error exit would be a sort function being asked to sort -5 items, an empty stack being asked to Pop(), a list being asked for the 13th entry when it only has 12 items; obviously in all these cases the program has got its data corrupted (or is in some other way just plain buggy) and there is no point in continuing.

You may find it useful to define a function like:

```
void DieOnError(const char* msg) { cerr << msg << endl; exit(1); };
```

you set a breakpoint in this, then you can use the debugger to work back up the stack looking at calling sequence and data while you try to sort out why things went wrong.

If you really do encounter an error that the calling code could do something about, then it is appropriate to return an error indicator (e.g. you have been asked to load a particular record from a file and find that the record specified doesn't exist – most likely the user typed a wrong number and should be allowed another go).

There are several conventions:

- return 0 if all went well, return 1 if failed – possibly setting some global "errno" variable to a value that explains the specific failure mode (lots of system functions do this)
- return 0 if all went well, return a non-zero error number if function failed
- return 0 if all went well, return 1 if failure with an extra error variable (passed by reference) being set to a value that identifies the specific failure mode
- return 1 (true) if succeeded, 0 (false) if failed (opposite to first convention)

OK, you can't avoid these inconsistent styles. Just make sure that all the functions you write for a program adopt the same style.

Mixing error flags and results is a common practice (all those system functions that return NULL pointers or valid addresses are examples of practice). Again, try to minimize the extent that you do this. It is better to have a function that returns a success or failure indicator and sets a reference parameter to hold the result.

If you believe that the code that calls your function really should pay attention when you report an error, you had better learn how to use exceptions. (Put that off till next year.)

bool type

C++ has a boolean type bool with constants true and false. Most current compilers don't support this. Endless mess as you will find headers with #define FALSE 0, #define TRUE ~0 and typedef int boolean (and all kinds of variations on this theme).

Unfortunately, for now you will just have to use int instead of boolean, following the convention that 0 is false, any non-zero value is true.

Assertions

Use assertions to validate data passed to principal functions.

Public member functions of a class, and any "exported" functions from a function style module, should check their data. Private member functions, and static functions within modules shouldn't need to repeat the checks (because you will have had to reach them via a public function that has done the checks) but you might have them while still debugging your own class.

You have to #include <assert.h>. Then you can have checks like the following:

```
void Sort(int data[], int low, int high)
{
     assert(low >= 0);  // no negative subscripts please
     assert(high >= low); // I don't know how to sort upside down
     …
}
```

NEVER include any effective code from a function within an assert statement. The following is wrong:

```
void Something(char *datastring)
{
     char buff[kMAX];
     int len;
     assert( (len = strlen(datastring)) < kMAX);
     …
     for(int i=0; i < len; i++) …
```

A compiler can be told to omit the assert statements! If there is effective code in an assert, the resulting optimized code is incorrect. In this example, variable len would not be initialized. The correct (and more natural) way of coding this example is:

```
     int len = strlen(datastring);
     assert( len < kMAX);
```

Functional decomposition

If you can point to a few lines of code and state its purpose, e.g. "this computes …", "this opens the file if it exists already, otherwise it creates it", you've probably found a new function. Cut out that code, promote it to an auxiliary function (private member function if a class, static auxiliary function if a function style module).

Functions should be small.

Functions should do one thing.

Functions that take an argument and use it to select a subset of statements really should be a set of separate functions.

Be aggressive at decomposing functions into smaller auxiliary functions.

Case statements

Make sure that you have:

- constructed each clause correctly, with a "break" to prevent "fall through" into next clause (if you mean fall through to occur, this should be clearly commented in the code)
- have considered a default clause (possibly even put one in with no statements but simply a comment explaining why no default action needed).

If statements

Always double check that you don't need an else clause for an if.

That expression being tested in the if() – should it be promoted to a function? If the expression involves several && and || operators you really should be thinking about a boolean function.

If you have several clauses combined, make sure you are using the boolean && and || operators not the bitwise & and | operators.

Expressions

Use parentheses even if they are redundant. Most of us can't remember the rules of precedence of the operators in C++.

Don't combine autoincrement (autodecrement) operators into expressions (sure such usage is legal, sure all the great C hackers do it, it is also very error prone). A simple example:

```
while(sum < 50) { sum += data[i]; i++; }
```

rather than

```
while(sum < 50) { sum += data[i++]; }
```

Don't be smart alecky and mix arithmetic and boolean operations; only nerds enjoy code like

```
LongStringCount += strlen(stringdata[i++]) > 15;
```

Pointers

A pointer variable holds the address of another variable. In C++, almost always a pointer should hold the address of a dynamically allocated data structure that you created with the new operator. The only time that you should be using the & "address of" operator to get the address of something is when supplying an argument to the low-level read/write binary data transfer functions associated with input/output streams or, occasionally, when supplying an argument to one of the old C library functions that gets used from C++.

Pointers are a major cause of problems.

Pointers that point somewhere they shouldn't:

These result from careless coding. You have used an uninitialized pointer, or you've used a pointer despite the fact that its value is NULL (you should have checked), or you've used a pointer to a data structure that you once created with new but which you have since deleted.

On Macs and PCs, such misuse of pointers may pass unnoticed (your program's output is quite probably wrong, but you never check it that carefully do you). On Unix, the misuse is picked up at run time and your program is killed. The plaintive excuse "Well, it worked on my PC", is often heard by markers (who just laugh). Your program has a bug. Unix was nice enough to "point" this out.

Memory leaks:

You create a structure with new and, in an assignment statement, store its address in a pointer. You use it for a while. Then you forget about it. Your pointer goes out of scope or is set to point somewhere else. The data structure you created remains sitting in the heap like a kind of zombie (dead but still hanging around).

You have a memory leak. You also have a mark leak.

Pointer sillies

Markers often see code like the following:

```
char *ptr_char;
ptr_char = new char;
…
*ptr_char = 'a':
…
…  = *ptr_char …
…
delete ptr_char;
```

or worse:

```
char *ptr_char;
ptr_char = new char;
cin >> ptr_char;  // read name
…
```

The first is stoooopid, but not incorrect. The programmer wanted a single character variable. Instead of just using char ch (and getting an automatic), this programmer arranges to have a pointer variable, sets this pointer to a newly created dynamic data structure intended to hold one character, and then laboriously works by dereferencing the pointer to get at the data. The overhead is immense. Operators new and delete aren't cheap. There is a storage allocation overhead; the data structure allocated in the heap is probably 12 or 16 bytes in size (not one). All the pointer dereferencing makes the code difficult to read.

The second example is wrong. It says create space for one character. Now read in an arbitrary length string. The characters in the string will destroy the "housekeeping information" associated with the dynamically created data structure (and other data structures nearby in the heap). The program will die in some horrid way shortly afterwards. The marker will wish a similar fate befall the programmer.

standard libraries

There are about 14 standard C libraries that are also used from C++. C++ also adds its own libraries. You need to know about a few of these. If you have a module that uses any of the functions from these libraries, you must #include the header file describing the corresponding library. (Caution. The Integrated Development Environments used on Macs and PCs often "help" you by effectively #including these headers without your having to ask. Unfortunately, your program then gets compilation errors when you transfer it to Unix.)

The "C" libraries have all been updated so that they work for both C and C++. You will probably use the libraries described in the following header files:

- assert.h
  The assert "macro" that you use to check arguments.
- ctype.h
  The "functions" like isalpha(char), isdigit(char) used to check characters.
- limits.h
  Mainly a set of constants defining things like the size of the largest short integer, long integer etc.
- math.h

Prototypes for standard mathematical functions for sine, cosine, logarithm etc. (Some versions also define lots of useful constants, e.g. PI; unfortunately, this isn't standardized.)

- stddef.h
  Mainly typedefs. (Including a spare definition of NULL just in case you didn't include stdlib.h.)
- stdlib.h
  Always #include this. Odds and ends you need all the time (like the definition of NULL for work with pointers). Look at its contents sometime --- lots of useful functions like atoi() which converts a string of digits into an integer value, rand() which gives you a pseudo random number, exit() which allows your program to stop when all is lost, …
- string.h
  The functions like strcpy(char*, const char*) (copying a string), strlen(const char*).
- time.h
  Functions for measuring time, and functions for converting times and dates into strings.

You will probably only use a few of the C++ specific libraries. The ones that you use will be those providing input/output functions:

- iostream.h
  The main streams classes, cin, cout, cerr.
- fstream.h
  The extensions that allow you to read and write files.
- strstream.h
  An extension that allows you to use an array of characters as if it were an input or output stream (sometimes useful for formatting messages).
- iomanip.h
  A few add ons for the iostream.h stuff, provides the "manipulators" that help do formatting.

code layout

The integrated development environment that you use on a Mac or PC will attempt to arrange your code in a sensible manner.

Read your 111 notes or the textbooks for recommendations on layouts. There are different styles for indentation and placement of { "begin" and } "end" brackets. Just adopt one style and stick to it consistently in your code.

Functions

There are a number of function libraries that you will use; some of the more commonly used functions and their libraries are identified here.

Finding out more about a function (arguments, result type etc):

On Macintosh:

Use the "Think Reference Program". This has a set of "databases", select SLR (standard library reference) or IOStream (input output) ("Databases" button will show a selection menu). Program displays list of all functions, select the one you require.

On PC:

Use Help system, entering name of function. Hypertext style cross references can help you find useful related functions whose names you don't know.

On Unix:

Use the man (manual) command with name of function (bit patchy with iostreams, as although has details on all the functions only a few are listed in the man pages index).

`Selection of more commonly used functions:`

Function name	Library header	Use
abs	stdlib	absolute value (see also labs, fabs)
assert	assert	test value? terminate program
atoi	stdlib	convert string digits to integer
clock	time	time since program started
cos	math	cosine
ctime	time	time to date - time string
exit	stdlib	terminate program
fabs	math	absolute value of float
isalnum	ctype	is character letter or digit?
isalpha	ctype	is character letter?
isdigit	ctype	is character digit?
islower	ctype	is character lowercase?
isupper	ctype	is character uppercase?
memset	string	initialize block of memory (array)
memcpy	string	copy array
rand	stdlib	random number
sin	math	sine
sleep	"unix"	delay (*not standard, library varies*)
srand	stdlib	initialize random number generator
strcat	string	append string
strcmp	string	compare strings
strcpy	string	copy one string into space of other
strlen	string	length of string
strncpy	string	copy part of string
strtok	string	break string into "tokens"
time	time	get time
toupper	ctype	convert character to upper case
tolower	ctype	convert character to lower case

Stream i/o

OK, this does get moderately complicated with a complete hierarchy of related classes. The following descriptions are simplified, focussing just on those aspects that you are most likely to use.

There is a further problem. The standardisation process for C++ is still not finalized. Some changes relating to the streams library are proposed. If these are adopted, they will be introduced into compilers at different rates. Consequently, there will be inconsistencies between system.

iostream

The iostream header defines the basic input stream, istream, and output stream, ostream, functionality and also the specialized standard streams cin, cout, and cerr.

An istream object can:

- "give to" any of the standard builtin data types:

  ```
  in >> x; // 'x' can be int, long, double, character, or character string
           // or an instance of a class that you defined and for which you
           // have provided an associated istream& operator>>(…) function
  ```

- report its status
  in.eof(), in.good(), in.bad(), in.fail()

- input a complete line (defined by maximum number of characters, or end of line marker)
  in.getline(…);

- skip over characters (limit number or end marker)
  in.ignore(…);

- read a character (unlike the >> operator), this version does not skip whitespace
  in.get(ch);

- read a block of bytes (use this only for record structured files)
  in.read(…);

- set or report byte position (only useful for record structured files)
  in.seekg(…);
  in tellg()

Similarly, an ostream object can:

- "take from" any of the standard builtin data types:

  ```
  out << x; // 'x' can be int, long, double, character, or character string
            // or an instance of a class that you defined and for which you
            // have provided an associated ostream& operator<<(…) function
  ```

- report its status
  out.good(), etc

- write a block of bytes (use this only for record structured files)
  out.write(…);

- set or report byte position (only useful for record structured files)
  out.seekp(…);
  out tellp()

An ifstream is an istream that you can attach to a file on which you can then do operations like open(), or close().

Similarly, an ofstream is an ostream that you can attach to a file.

An fstream is for files used for both input and output. It can do anything an ifstream can do as well as anything an ofstream can do.

A function that specifies that it wants to be passed an istream can be given an ifstream (or an fstream) once the file opening has been done.
Similarly, a function that uses an ostream can be passed an ofstream (or an fstream).

---

strstream

The strstream group allows you to use a character array in memory as a "stream". Most frequently used to convert a value to a string that you want to pass around as a string.

---

iomanip

The iomanip part of the library defines a few extra "manipulators" that can simplify formatting.