
Introduction to Programming with C++

Aaron Naiman

Jerusalem College of Technology

naiman@jct.ac.il

<http://jct.ac.il/~naiman>

Copyright ©2011 by A. E. Naiman

UNIX Intro (optional)

- ⇒ Help
 - File System
 - Emacs
 - Mail

From PC Environment

- VMWare, Virtual PC, etc. — for the entire UNIX GUI
- Cygwin, MinGW — for *emulating* the UNIX environment
- Secure Shell Client (SSH), PuTTY, QVTNet — include
 - * telnet
 - * ftp

Plenty of methods for UNIX access from Windows.

UNIX Help

- `homedir:/usr/u/public/unixintro/` —
 - * better: use the Internet (“basic unix commands”)
- `apropos concept` (alt., `man -k`), e.g.,
 - \$ `apropos copy`
 - `cp (1)` - copy files
 - `cpio (1)` - copy file archives in and out
 - `pg (1V)` - page through a file on a soft-copy terminal
 - `rcp (1C)` - remote file copy
 - `tcopy (1)` - copy a magnetic tape
- `man command`
- `man intro`
- email to `problems(@mail)`

Know the help system!

UNIX Intro (optional)

- Help
- ⇒ File System
- Emacs
- Mail

UNIX File System

- directories: `mkdir`, `rmdir`, `cd`, `pwd`, `ls`, `ls -lt`, `ls *.C`, `ls *.C*`
- files: `cp` (e.g.: `cp a1 b1`), `mv` (e.g.: `mv a1 b1`, `mv a1 dir1/`), `rm`
- examination: `more`, `less`
- modification: `emacs [-nw] big_database.C`
- compilation: `make big_database` or
`g++ -Wall -o big_database big_database.C`
- debugging: `gdb` (see man page)
- some file types:
 - * `big_database.C`: C++ source
 - * `big_database.C~`: old C++ source
 - * `#big_database.C#`: interrupted editing of C++ source
 - * `big_database.o`: object file (binary, not readable, big)
 - * `big_database`: executable file (binary, not readable, big)

UNIX Intro (optional)

- Help
- File System
- ⇒ Emacs
- Mail

Emacs

- help: `^h(a|t)`
- move: `^f`, `^b`, `^p`, `^n`, `^a`, `^e`, `^v`, `M-v`, `M-x line<RETURN>`
- search: `^s`, `^r`
- files and buffers: `^x^s`, `^x^f`, `^xi`, `^xb(<SPACE>)`
- windows: `^x2`, `^xo`, `^x1`
- cut/paste: `^<SPACE>`, `^w`, `M-w`, `^y`, `^_`
- quit command: `^g`
- exit: `^x^c` (or `^z` and `fg`)

UNIX Intro (optional)

- Help
 - File System
 - Emacs
- ⇒ Mail

Electronic Mail

- emacs: M-x mail, ^c^c
- mail fred@poor.students.edu < some-money.C
- pine
- elm
- Gmail — of course!

Certainly you are not mailing homework.C !

Fundamental Types

- ⇒ Introduction
- Basic Types and Variables
 - Defining Constants
 - References

The Basic Program

```
#include <iostream>
using namespace std;

int main(void) {
    cout << "Hello world!" << endl;

    return 0;
}
```

- The compiler reads character by character, line by line

When reviewing or debugging, simulate the compiler.

The Basic Elements

- `#include` *actually* includes the header file
- All functions communicate with caller (usually a function)
 - * Functions by default (and `main()` always) return an `int`
 - * `void` == the function takes no arguments
- `{}` delineates a code block
- output:
 - * a string (how long?)
 - * a newline (`'\n'`) character
 - * a flushing (since output is buffered)
- A 0 is returned to the operating system

Preliminary Debugging

- Error *and warning* messages supply useful information ⇒
Do not fear or ignore them!
- Debug syntax in compiler order ⇒
start with personal header files (later)
- Trace run irregularities (*before* you run to me), e.g.
 - * premature termination
 - * “stuck” program—probably an infinite loopby sprinkling `cerrs` to locate problem
`cerr << "foo" << endl; // endl: clarity (and flushing for cout)`
- Debug program logic by inspection ⇒
add `cerrs` with relevant information
- Fix from the start and recompile/relink
 - * if necessary: divide and conquer!

Read, don't run!

Error Messages Right at the Start

- “method” = function, “parse” = syntax
- Bad header file name (and no spaces in angle brackets)
`foo.C:1: iostream,h: No such file or directory`
- Introduction of where to look
`foo.C: In function ‘int main()’:`
- Undeclared (and undefined) variable (note line number)
`foo.C:6: ‘i’ undeclared (first use this function)`
`foo.C:6: (Each undeclared identifier is reported only once`
`foo.C:6: for each function it appears in.)`
- Due to missing `#include` (also for functions)
`foo.C:8: ‘cout’ undeclared (first use this function)`

Typos and Syntax Errors

- Mixing types: `int i = 4.5;`
`foo.C:8: warning: initialization to 'int' from 'double'`
- Missing usage, possible typo
`foo.C:6: warning: unused variable 'int j'`
- Parse error = something is not C++ kosher
 - * missing a ";" on (the) previous line
 - * specific code pattern required: `for (j = 0; j < i, j ++)`
`foo.C:10: parse error before ')'`
 - * bad function argument, "&" of reference in wrong place
`"list.h", line 21: error(3112): expected a ")"`
`int foo(const &char a, const double& b);`

Fundamental Types

- Introduction
- ⇒ Basic Types and Variables
- Defining Constants
- References

Basic Data Types

- Computer only contains 0's and 1's \Rightarrow
 - * need to convey how to relate, e.g., 4-byte integers, 8-byte floating point numbers, etc.
 - Integral types [standard workstations]
 - * char (1 byte), short [2], int [4], long [4]
 - * Monotonic nondecreasing
 - * signed (default, except maybe char) or unsigned
 - * E.g.: $-128 \leq \text{char} \leq 127$, $0 \leq \text{unsigned char} \leq 255$
 - Floating point types [standard workstations]
 - * float [4], double [8], long double [12]
 - * Monotonic nondecreasing
 - * Components: sign, mantissa and exponent
- Aggregate types are built on these.

The char Type

- char: just a (small) int number
- There are: unsigned char, signed char, and (implementation dependent) char
- signed types use “twos-complement”
- signed char:

1	==	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1			
127	==	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1			
-1	==	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1			
-128	==	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0			

char I/O

- All chars have special I/O based on ASCII
- E.g., note following definitions, with initializations:

```
int  i_of_i = 7,      // all 4 take _integer_ values
    i_of_c = '7';    // ASCII: character -> value (55)
char c_of_i = 7,     // ditto
    c_of_c = '7';

cout << i_of_i << " " << i_of_c << " " <<
    // ASCII: value -> character
    c_of_i << " " << c_of_c << endl;
```

produces:

```
7 55 7
```

note invisible chars: <BELL> and <RETURN> (where?)

To input a char number—input to an int.

Constant Values

- Integral bases: 68, 043, 0x3B
- Integral suffixes: -45L, 88u, 4ul
- Floating point suffixes (default is double): 45.f, .34L, 0.87E-2
- Characters: 'a', '\n', '\"'
 - * ASCII order: ..., '0', '1', '2', ..., '9', ..., 'A', 'B', 'C', ..., 'Z', ..., 'a', 'b', 'c', ..., 'z', ...
- Strings (null-terminated): "", "I am a string", "\n", "tab here:\t"

Constants values are (obviously) non-addressable.

Variables

- Avoid “magic” values (numbers, file names, etc.)
- Use sensible, descriptive names: `buffer_size`, `stringLength`,
....
- All elements of C++ (and C) are case-sensitive
- Definition and initialization (match up types *exactly*):

```
float weight, light_speed(3e8f);  
char bell = 7, input_char;
```
- Declaration, i.e., variable “lives” elsewhere:

```
extern float weighted_average;
```
- Memory allocation: definition \checkmark , declaration \times
- Type/variable size (in bytes), e.g., for UNIX:

```
sizeof(float) == sizeof(weight) == 4
```

In general, define and declare variables *within* functions.

Fundamental Types

- Introduction
- Basic Types and Variables
- ⇒ Defining Constants
- References

Symbolic Constants

```
const int max_buffer = 512;  
const float e_approx = 2.718f;
```

- Much preferred to “magic” values (here: numbers)
 - * What does “512” mean?
 - * Localized if need to change value
 - * What if “512” means a number of things?!
 - * Later: same for strings, e.g., file names
- Memory is allocated
- Must be initialized (because ...)
- Cannot be changed, i.e., is “read-only”
- Actual values not seen anywhere else in code
- Preferred to `#defines`

Enumeration

```
enum {fail, pass};    // 0 is default start value
enum boolean {false, true, sheker = 0, emet};
boolean test_started = false;    // memory allocation
enum menu_op {quit, input_data, analyze, print_report};
enum suit {clubs, diamonds, hearts, spades};
enum color {red = -1, green, blue};    // prefer to "colors"
color jacket = blue;    // memory allocation (short/int)
```

- blue is a constant and jacket a variable of type color
- *Just like* for “int i = 3;”: 3 is a constant and i a variable of type int
- error: color jacket = 0;
- Actual values not seen *anywhere* in code

Good for strong typing and self-documentation (esp. menus).

Fundamental Types

- Introduction
 - Basic Types and Variables
 - Defining Constants
- ⇒ References

Reference Types

```
short score;
```

```
short& score_ref = score;
```

- Used as an alias for another *existing* variable
- No additional memory allocation
- `score_ref` and `score` have same usage and syntax
- Move “&” to left to disambiguate from address-of operator (later)
- Like a `const`, association is permanent
 - * must be associated immediately (because ...)
 - * association cannot be changed

Primarily for arguments and return values of functions.

const References

```
long l;  
const long cl = l;  
long& l_ref = l;           // same as earlier  
// warning: conversion from 'const long' to 'long &' discards const  
long& cl_ref = cl;  
const long& c_l_ref = l;   // new restricted access to "l"  
const long& c_cl_ref = cl; // can associate to a constant value
```

- Will not let you change `c_l_ref` or `c_cl_ref`
- Restricted alias: `l = 6L; ✓`, `c_l_ref = 6L; ✗`
- Analogy: take Bob, who may be also called “Son”, “Honey” and “Dad”—someone who refers to him as
 - * “Bob”, “Son” or “Honey” can (and do)
 - * “Dad” cannotfollow up with “... You’re an idiot!”

Pointers and Arrays

- ⇒ Pointers
 - Arrays
 - Compare and Contrast
 - Multidimensional Arrays
 - Typedefs

Pointers

- Every variable—even a `const`—has an:
 - * address (“lvalue”)
 - * value (“rvalue”)
- Which are used where?: `int a = 4; a ++;`
- Definition, initialization and assignment (check the types!):

```
int  some_info, *i_ptr = &some_info, **int_buffer_ptr;  
char ch, *char_index;    // allocation for _any_ pointer:  
char_index = &ch;       // (sizeof(int)) bytes (why?)
```
- Note: “*” in initialization, but not in assignment (why?)

Pointer Dereferencing and Arithmetic

- Note: a calculation has no lvalue
 - * e.g.: `j + 9, &a`
 - * not found in the memory (where then?)
- Dereferencing and pointer arithmetic:

```
*i_ptr += 2;    // == some_info += 2;  
i_ptr += 2;    // == i_ptr increases by 2 * sizeof(int);
```
- Dereferencing ∴ valid pointer states
 - * point to some variable, or
 - * set to 0 (what does this signify?)

at all times

Primarily for free store, unnamed memory allocation.

Pointer Arithmetic — Example

```
double *p, *q;  
p = (double *) 100;    // p == 100  
q = p + 10;           // q == 180
```

code	value
$q - p$	10
$p - q$	-10
$q - 10$	100
$q + 10$	260
$10 + q$	260
$10 - q$	not defined
$q + p$	not defined
$q - (\text{int} *) p$	not defined

Constant Pointers

```
float *fp;
// fcp is a const pointer, to a float
float *const fcp = &some_float;
const float *cfp;    // cfp points to a "const float"
const float *const cfcp = &some_const_float;
```

- Note: (fp → fcp) as (int → const int); cfp is different type

	can change what it points to	can point to a const float	needs initialization (& cannot be changed)
fp	✓	×	×
fcp	✓	×	✓
cfp	×	✓	×
cfcp	×	✓	✓

Pointers to const are mainly for function arguments.

Pointers and Arrays

- Pointers
- ⇒ Arrays
- Compare and Contrast
 - Multidimensional Arrays
 - Typedefs



Arrays

```
const int num_students = 23, pixels_per_side = 512;
int grades[num_students], teacher_iqs[] = {58, 93, 81};
double image[pixels_per_side][pixels_per_side],
    *deviations = &image[5][10];
```

```
int student;    // note: init. outside "for", see later
for (student = 0; student < num_students; student++)
    grades[student] = 0;    // :-)
```

- Array size must be
 - * a `const`, and
 - * known at *compile* time
- `{ ... }`: for array initialization only, *not* assignment
- Multidimensional: last index iterates fastest
- Pointer deviations allocates `sizeof(double *)` bytes

Array Definition Errors

- `int jj[];`
`foo.C:6: array size missing in 'jj'`
- `int jj[4.5];`
`foo.C:6: size of array 'jj' has non-integer type`
- Recall, standard requires
 - * `const` array size \Rightarrow
: g++ does not warn
 - * known at compilation \Rightarrow
: g++ sometimes more lenient

Stick to the standards.

Character Strings

- string = a NULL-terminated array of char
- "bar"
 - * type: `char[]` (this easy form, for char only)
 - * advanced: *not* const, so this will work:

```
char *foo = "bar";
```
 - * for initialization, equivalent to: `{'b', 'a', 'r', 0}`, automatic with double quotes

```
char greeting[] = {'H', 'i', '!'}, // 3 chars, vs.:
fun_question[] = "Are we having fun yet?";
```
 - * other, poor usages
 - ★ assignment to pointer (memory leak)
 - ★ function argument ("magic", prefer variables)

Strings are very popular and common.

Strings and Arrays

- Note: "info.dat" \neq " info.dat "
- Strings vs. other arrays
 - * do not forget extra byte
 - * string length can be calculated (count to NULL)
 - * no need to "shlep" around length
 - * therefore: `strlen()`, `strcpy()`, ...

- String arrays

```
char *stooges[] = {"Larry", "Moe", "Curly"};
cout << stooges[1];          // type: char *, prints: Moe
cout << *(stooges[2]);      // type: char,  prints: C
cout << (int *) (stooges[1]); // type: int *,
                             // prints address of 'M' (alt.: (unsigned))
```

Always know the type.

Pointers and Arrays

- Pointers
- Arrays
- ⇒ Compare and Contrast
- Multidimensional Arrays
- Typedefs

Arrays and Pointers

```
long id_numbers[5];  
long *id_num_ptr = &(id_numbers[3]);
```

- Similarities

- * Both tags alone refer to address of its first element

```
if (id_num_ptr == id_numbers) { /* ... */ }
```

- * ⇒ both can use array and pointer syntax

```
id_num_ptr[1] = *(id_numbers + 2);  
// copying third element into fifth
```

- Differences

- * `id_numbers` only has an rvalue: *refers* to address of beginning of array and cannot be changed

- * `id_num_ptr` also has an lvalue: an extra (`long *`) is allocated and can be set to address of a long

- * e.g., try: `cout << ptr << *ptr << &ptr;`

Pointers and Arrays

- Pointers
- Arrays
- Compare and Contrast
- ⇒ Multidimensional Arrays
- Typedefs

1-D Pointer and Array Definitions

```
double *dp = 1000, da[5];
```

- Assume allocated at 4000 and 5000, resp.
- Bytes allocated
 - * `dp`: `sizeof(void *)` [4]
 - * `da`: `5 * sizeof(double)` [40]
- `sizeof(dp)` and `sizeof(da)` return these numbers
- Note: initialization to 1000 is a bad idea (why?); better:
 - * `dp` gets set to address of double, e.g., `&(da[3])`
 - * `dp` gets return value of `new()` (later)
 - * `dp` gets set to 0 (why?)
- Given `da` above, what are: (*hint*: what are the *types*?)
 - * `da[10]`
 - * `&(da[3]) - &(da[-4])`

1-D Pointers

syntax	type	lvalue	rvalue	comments
dp	double *	4000	1000	name alone
dp + 2	double *	—	1016	pointer arithmetic
Rule: in bytes: $dp + n * sizeof(double)$ (i.e., remove 1 *)				
*dp	double	1000	$r(1000)$ [□]	pointer dereferencing
Rule: add a */[] of dereferencing \Rightarrow drop a */[] of the type				
dp[3]	double	1024	$r(1024)$ [□]	array syntax
Rule: $dp[n] == *(dp + n)$ (a combination)				

[□] $r(n)$ means whatever resident at memory location n

1-D Arrays

syntax	type	lvalue	rvalue	comments
da	double[]	—	5000	name alone
Rule: array without []: <i>rvalue</i> is shorthand of <code>&(da[0])</code>				
Rule: <code>double[]</code> is <i>like</i> a <code>double *</code> (but not exactly)				
..... same as pointer				

- Note: the size of 5 doubles was *only* used for
 - * initial allocation
 - * future invocations of `sizeof()`

Now for 2-D arrays

2-D Pointer and Array Definitions

```
double **dpp = 2000, dm[7][11];
```

- Assume allocated at 6000 and 7000, resp.
- Bytes allocated
 - * dpp: `sizeof(void *) [4]`
 - * dm: `7 * 11 * sizeof(double) [616]`
- `sizeof(dpp)` and `sizeof(dm)` return these numbers
- Again, initialization to 2000 is a bad idea
- Now for same analysis, with (almost) same rules

2-D Pointers

syntax	type	lvalue	rvalue	comments
dpp	double **	6000	2000	name alone
dpp + 2	double **	—	2008	pointer arithmetic
*dpp	double *	2000	$r(2000)$ [□]	pointer dereferencing
**dpp	double	9788	$r(9788)$	multiple dereferencing
dpp[3]	double *	2012	$r(2012)$ [†]	array syntax
Rule: <code>dpp[n] == *(dpp + n)</code>				
dpp[3][5]	double	8884	$r(8884)$	multi-array syntax
Rule: <code>dpp[n][m] == (*(dpp + n) + m)</code>				

[□]say, 9788; [†]say, 8844

2-D Arrays

syntax	type	lvalue	rvalue	comments
<code>dm</code>	<code>double[][11]</code>	—	7000	name alone
Rule: compiler needs to know how to jump, e.g., <code>dm[1][0]</code>				
Rule: array type retains all dimensions except first				
<code>dm + 2[□]</code>	<code>double[][11]</code>	—	7176	partial ptr. syn.
Rule: <code>dm + n == &(dm[n][0])</code> (n rows later); but different types				
<code>dm[2][□]</code>	<code>double[]</code>	—	7176	partial array syn.
Rule: without all []s: <code>dm[n] == &(dm[n][0])[†]</code> (<code>!= dpp[n]</code>)				
<code>dm[2][4]</code>	<code>double</code>	7208	<code>r(7208)</code>	full array syn.

[□]note: same values, different types (and \therefore arithmetic, try: `+ 3`)

[†]note: supply [0]s for remaining dimensions; like in 1-D

2-D Arrays — Example

- Given `dm` above, first determine the *types*.

code	value
<code>(unsigned long) dm[0]</code>	3221222656
<code>(unsigned long) (dm + 2)</code>	3221222832
<code>(unsigned long) dm[2]</code>	3221222832
<code>(unsigned long) (dm[0] + 2)</code>	3221222672
<code>&(dm[2][0]) - &(dm[0][0])</code>	22
<code>(dm + 2) - dm</code>	2
<code>dm[2] - dm[0]</code>	22
<code>(unsigned long) (dm + 2) - (unsigned long) dm</code>	176

Pointers and Arrays

- Pointers
 - Arrays
 - Compare and Contrast
 - Multidimensional Arrays
- ⇒ Typedefs

Motivations and Examples

- To introduce type synonyms, not new types

```
typedef double weight;    // documentation
typedef char *string;     // readability
typedef short id_num;     // portability
```

```
// ... usages:
```

```
weight student_a, *p_student;
string str1, *p_str, str_a[5];
const id_num last_student = 78;
```

Can be very useful.

Basic Operations

- ⇒ Operators and Assignments
 - Type Conversions
 - Statements
 - Input/Output
 - The Free Store

Arithmetic and Relational Operators

- Arithmetic: +, -, *, /, %

`7 / 3 * 3 == 6`

- For boolean logic/question: yes/true/1 vs. no/false/0

- Equality and inequality: ==, !=

- Relative inequality: >, <, >=, <=

- Conditional:

```
int a, b, max;  
max = (a > b) ? a : b;
```

Beware of precedence—use ()s.

Logical and Bitwise Shifts

- Logical (true/false): `&&`, `||`, `!`

```
int n = 7 || 0;    // n == 1
```

- One's complement: `~`

```
n = ~4;    // all bits on except third to last
```

- Bitwise shifts: `<<`, `>>`

```
n = 12 >> 2;    // n == 3
```

- * standard set for unsigned, integral types only
- * bit-wrap, 0-fill or 1-fill is direction and compiler dependent

Bitmasks

- Bitwise masks, logically bit-by-bit: $\&$, $|$, \wedge
- Zeroing out last 4 bits: $n = n \& \sim 017;$

017	$==$	<table border="1"><tr><td>0</td><td>0</td><td>...</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	0	...	0	1	1	1	1
0	0	...	0	1	1	1	1			
~ 017	$==$	<table border="1"><tr><td>1</td><td>1</td><td>...</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	1	...	1	0	0	0	0
1	1	...	1	0	0	0	0			
n	$==$	<table border="1"><tr><td>1</td><td>0</td><td>...</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	1	0	...	1	0	0	1	1
1	0	...	1	0	0	1	1			
$n \& \sim 017$	$==$	<table border="1"><tr><td>1</td><td>0</td><td>...</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	0	...	1	0	0	0	0
1	0	...	1	0	0	0	0			

- Note: better than $n \& 0177760$ which assumes ≥ 16 bits

Assignment

- Simple, with constant or other variables:

```
float salary, initial_salary, raise;
salary = 9500.f;    // lhs: non-const, with lvalue
initial_salary = salary;
```

- Pre/post increment/decrement: ++, --

```
int num_objects = 0, grades[50], student = 6;
num_objects++;    // increments num_objects by one
grades[++ student] = 0;    // student == 7 and
                           // grades[7] == 0
```

- Compound: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=

```
salary += raise;    // same as: salary = salary + raise;
```

- * Quick to get used to and easy to read
- * Often used with pointers and indices

Basic Operations

- Operators and Assignments
- ⇒ Type Conversions
- Statements
- Input/Output
- The Free Store

Type/Variable Size and Conversion

- Recall: `sizeof()` for types and variables
 - * reminder: all pointers are of the same size
- For initialized array: good maintainability practice

```
float prices[] = {54.4f, 97f, 0, /* ... */, 72.0f};
const int num_prices = sizeof(prices) /
    sizeof(prices[0]); // AOT: sizeof(float); (why?)
```
- Without casts, some conversions are implicit, plus warnings

```
int i = 10.2;    // i == 10
i *= 2.5;       // i == 25
```
- Explicit casts \Rightarrow *do it*, for assignments, binary operators, ...

```
int i = (int) 10.2, *i_ptr = &i; // _temp_ val. created
i_ptr = (int *) (((char *) i_ptr) + 1); // DANGEROUS!
```

Use “-Wall” and be explicit about all casts.

Basic Operations

- Operators and Assignments
- Type Conversions
- ⇒ Statements
- Input/Output
- The Free Store

Simple Statements

- Null statement: `;` (used, e.g., in loops)
- Declaration/definition: `char bell = 7;`
- Simple statement: `bell *= 6; // ??`
- Compound statement (with declarations: *block*):

```
{  
    bell /= 6;  
  
    // note not at beginning of block  
    int i;  
    i = (int) bell;  
}
```

Conditional if Statement

- if: branch in one of two directions

```
if (grades[student] >= passing_grade)
    congratulate_student(student);
else {
    call_rector(student);
    call_parents(student);
}
```
- Can be nested and daisy-chained
- Use {}s to avoid ambiguity

Note “;” which follows “then” *simple* statement.

Conditional switch Statement

- `switch`: a multi-branch juncture

```
switch (user_command) {
    case 'Y': case 'y':
        // ... //      don't forget breaks, else code
        break; //      falls through
    case 'N': case 'n':
        // ...
        break;
    default:
        // ... //      supply even final break, as other
        break; //      cases may be added later
}
```

- Do not define variables in cases, unless “block” ed

Some say to avoid in C++.

Outputting enums

```
enum menu_ops {INPUT, ANALYZE, OUTPUT, QUIT};
// ...
cout << "Please enter an option:" << endl;
cout << "    0 -- input" << endl;
cout << "    1 -- anaylze" << endl;
cout << "    2 -- output" << endl;
cout << "    3 -- quit" << endl;
// ...    what's wrong?
switch (chosen_op) {
    case INPUT:
        // ...
    case ANALYZE:
        // ...    ...    ...
}
}
```

Let enum Do Its Job!

- What if VALIDATE added after INPUT?
- Use enum enumeration just like in cases
- May need typecast: << (int) INPUT

```
cout << "Please enter an option:" << endl;
cout << "    " << INPUT    << " -- input" << endl;
cout << "    " << VALIDATE << " -- validate" << endl;
cout << "    " << ANALYZE  << " -- anaylze" << endl;
cout << "    " << OUTPUT   << " -- output" << endl;
cout << "    " << QUIT     << " -- quit" << endl;
```

Loops

- `while`: loop forever while condition remains true, i.e., non-0

```
while (input_is_valid) {
    congratulate_user();
    process_input();
    get_more_input();
}
```

- `for`: with more standard indexing structure (note indentation)

```
int hour;                // int hour = 6;
for (hour = 6;           // while (hour < 24) {
    hour < 24;          //     study(hour);
    hour ++);           //     hour ++;
    study(hour);        // }
```

- `do/while`: like `while`, but iterate at least once

Use the most appropriate loop mechanism.

For-initialization and Jumps

- Variables defined in `for` initialization
 - * used to: last until end of enclosing block
 - * new standard: last in `for` loop alone
 - * yet to be settled \Rightarrow define before `for`
- `break` out of the inner most
 - * loop of `while`, `for` or `do`, or
 - * enclosing `switch`
- `continue` to the next loop iteration
- `goto` a specific “label:”
often for breaking out of nested loops

Avoid “spaghetti” `goto` statements.

Basic Operations

- Operators and Assignments
- Type Conversions
- Statements
- ⇒ Input/Output
- The Free Store

I/O Overview

- All standard I/O require

```
#include <iostream>
using namespace std;
```
- Basic operators: << and >>; think of data movement direction

```
int num_points = 3, point1, point2, point3;
cout << "Please enter " << num_points << " points: ";
cin >> point1 >> point2 >> point3;
```
- Note how multiple I/O is daisy-chained
- Predefined: `cin`, `cout`, `cerr` (unbuffered), `clog`
- Also for I/O from/to: files (see below) and strings (FYI)

General Output

- `cout <<` understands all fundamental types, pointers (in hex), `'\t'`, `endl`, ...

```
unsigned short us = 5u, *usp = &us;
```

```
cout << *usp << " is stored at location\t: " <<
```

```
usp << endl;
```

produces \Rightarrow 5 is stored at location : 0x7fff2efe

- \exists compilers: non-void pointers output as 0/1 \Rightarrow

- * typecast to `(void *)`

- Strings are handled in a special way (by `op<<()`)—watch out!

```
char message[] = "WOW!!", *m_ptr = message;
```

```
cout << *m_ptr << " is stored at location\t: " <<
```

```
m_ptr << endl;
```

produces \Rightarrow W is stored at location : WOW!!

How do we handle strings?

String Output and Other Formatting

- `char[]` and `char *`: print ASCII values from `rvalue` until `NULL`
- ... instead, use a typecast (why not `&m_ptr?`):

```
cout << m_ptr << " is stored at location\t: " <<  
    (void *) m_ptr << endl;
```

produces \Rightarrow `WOW!! is stored at location : 0x7fff2ef0`
- Prefix address with `(unsigned long)` for decimal representation
- To change conversion base, insert format manipulators: `oct`, `hex`, `dec` (default)
- `#include <iomanip.h>` settings: `setprecision()` for output precision, `setw()` for minimum numeric or string field width
`setf()` for more format control, scientific notation,

Reformatted Output Example

```
cout << "Here is the answer: " << 42 << endl <<
    setw(20) << "and in octal: " << oct << 42 << endl <<
    setw(20) << "and in hex: " << hex << 42 << endl;

cout << "Here is the default format for pi: " <<
    setw(13) << M_PI << endl <<
    "    and here's some more precision: " <<
    setprecision(12) << M_PI << endl;
```

produces:

```
Here is the answer: 42
    and in octal: 52
    and in hex: 2a
```

```
Here is the default format for pi:          3.14159
    and here's some more precision: 3.14159265359
```

- Note: `setw(20) << 'f'` does nothing to *fixed*-length

Standard Input

- Input is daisy-chained

```
cout << "Please input your exercise and exam grades: ";  
cin >> exercise_grade >> exam_grade;  
if ((exercise_grade / exam_grade) > 10)  
    cout << "You copied your homework!!" << endl;
```

- `char[]` and `char *`: store ASCII values starting from `rvalue`

```
char name[80], *phone_ptr = &(name[40]);  
cin >> name; cin >> phone_ptr;
```

- White space (`' '`, `'\t'`, `'\n'`) separates fields

- `cin >>` and white space

- * skips over WS if necessary
- * does not read subsequent WS

Standard Input — Example

```
int i; float f; char s[10], c; // input:
                                // 4.9
cin >> i >> f >> s >> c; // 3.2 asdf
cout << i << " " << f << " " << // produces:
     s << " " << c << endl; // 4 0.9 3.2 a
```

- `cin >>` returns 0 upon EOF
 - * `^D` will enter an EOF (value: -1)
 - * use in while loop

```
while (cin >> f)
    array[++i] = f;
```

More Input Control

- Can avoid skipping white spaces by

- * reading char-by-char: `get()` (\exists also `put()`)

- `int in_char; // int, for -1 of EOF`

- `while ((in_char = cin.get()) != EOF)`

- `cout.put(in_char);`

- input:

I am
bored.

 loops 12 times ("`cin >>`" loops 9)

- * store return value from `get()` as (signed) int

- * reading until '`\n`' (or other char): `getline()`

- `const int buffer_size = 75; // of course warning user ...`

- `char buffer[buffer_size];`

- `cin.getline(buffer, buffer_size);`

Input Tools and Techniques

- Advanced tools: `putback()`, `peek()`, `ignore()`
 - For password input (UNIX): `system("stty -echo")`
 - Warning: do not be lazy—allow for multi-word strings!
 - * delineating from other info in a file
 - ★ have on a separate line
 - ★ separate fields with a special char, e.g., `' : '`
 - ★ leave to last entry on line
- ```
200 4.9 green apples
400 3.9 oranges
50 7.9 small pears
```

# File I/O

---

```
#include <iostream> // stdlib.h for exit(), but
#include <fstream> // better: try again,
#include <stdlib.h> // ask for another name, ...
using namespace std;
int main(void) {
 ifstream num("numbers.dat", ios::in); // open
 ofstream sqr("squares.dat", ios::out); // as well
 int number; // also: ios::app

 if (!num) { // bad name, no permission, ...
 cerr << "No success on opening input." << endl;
 exit(-1); // cerr: be even more informative!
 } // do check for sqr as well

 while (!num.eof()) {
 num >> number; // use just like: cin >> ...
 sqr << number * number << ' '; // cout << ...
 }

 return 0; // files closed here
} // note: if '\n' at end, last square printed twice (why?)
```

# Additional File Flexibility

---

- Can open and close files manually:

```
ofstream data; char file_name[80];
cout << "file name: "; cin >> file_name;
data.open(file_name, ios::app); // flexible,
// ... // not "magic"
data.close();
```

- Use `fstream` type for input *and* output together
- Recall: some I/O, e.g., `cout`, is buffered  $\Rightarrow$   
`cout << endl; // or: cout << flush;`  
helpful for exact location when debugging (`cerr` is unbuffered)
- For moving about: `seekg()` (from: beginning, current, end),  
`tellg()`

# Common Input Pitfalls

---

- Inputting to a `char`: signed/unsigned not standardized
- Inputting to an `enum` can be problematic
- Inputting to a `char *`, without allocated space
  - \* use free store, or
  - \* write to a `char[]`, *and*
  - \* inform user of maximum length
- “>>” and `getline()` interlaced
  - \* “>>” reads up to, but *not* including white space
  - \* follow it with `get()` to slurp up ‘\n’
- For user permission to continue, no need for `char` input (alternatively: `cin.get()`)

```
cout << "Please press <RETURN> to continue ..." ;
cin.ignore(line_length, '\n'); // skips line
```
- Database access: code `save()` first, then `restore()`

# Input States and Suspension

---

- I/O states: eof(), bad(), fail(), good()

```
int age = -1;
while ((age < 0) || (age > 150)) {
 cout << "Please input a reasonable age: ";
 cin >> age; // what if 'a' is entered?
 if (cin.fail()) { // alt.: !cin.good()
 cin.clear(); // without if, loops forever (why?)
 cin.ignore(250, '\n'); // b|c 'a' not read;
 } // line skipped
}
```

- Suspension: possible causes
  - \* “eating” too much (into too small string)
  - \* other snafus
- *Important*: can lead to infinite loop if waiting for eof()
  - \* check for non-zero istream (in while) or gcount()
- clear() can sometimes rectify

# Basic Operations

---

- Operators and Assignments
- Type Conversions
- Statements
- Input/Output
- ⇒ The Free Store

# The Free Store—Motivation

---

- Array restriction: size must be known at compile time

- For image processing:

```
const int height = 1024, width = 256;
```

```
float image[height][width]; // 1 Mbyte (4-byte floats)
```

what if actual image is

- \* much smaller (and perhaps many pictures) ⇒

- ★ wasted space

- ★ needlessly can exhaust memory

- \* even a little larger ⇒ error

- Why not ask (user and then) computer for exact amount needed?

- Free store is for additional memory allocation

# The Free Store

---

```
int *ip = new int; /* ... */ delete ip;
ip = new int(7); // initialized (3 things happening)
delete ip;
int sz; cin >> sz;
ip = new int[sz]; // uninitialized "array" of sz ints
delete [] ip; // free up "array"; "[]" iff in "new"
```

- Uninitialized by default
- Unnamed, accessed by pointers, e.g.: `*(ip + 2)`, `ip[5]`
- If free store is exhausted, `new` returns 0—check for it!
  - \* perhaps implementation dependent
  - \* better: have pointer set to 0 before `new()`
- Recall: pointers themselves are also allocated memory

# Allocation Size and Deallocation Placement

---

- Invocation of `new()` performs the allocation  $\Rightarrow$   
allocation size must be known at that point
- Note: if allocation size is known at compile time  $\Rightarrow$   
no sense (and more complications) in using free store
- With free store allocation, control life-span directly
- For conservation reasons, perform `delete()`
  - \* as soon as possible
  - \* but no sooner
- Note: pointer retains address info even after `delete()`
  - \* therefore: turn off pointer by hand (how?)

Always pair `new/delete`—beware of *memory leaks!*

# Segmentation Faults

---

Segmentation fault (core dumped)

- Access is only allowed to requested memory
- Stepping out of bounds (segment) can be catastrophic
- Typical scenarios
  - \* dereferencing unset (or uninitialized) pointer
  - \* out of array bounds
  - \* illegal `delete()` (e.g., of: `char *name = "Smith"`)
    - ★ hint: use `const` (where?) or initialize to 0
  - \* writing to pointer value, without `new()` allocation
- File core contains snapshot of memory at program crash
  - \* it will be big
  - \* there are programs which can decipher (e.g., `gdb`)
  - \* do not forget to remove

# 2-D Free Store Allocation

---

- Motivation: user-defined size (`new()`) of 2-D array
  - \* want 2-D array syntax, e.g.: `mat[i][j] = ...`
- Restriction: `new()` will only allocate a 1-D array,  $\therefore$ 
  - \* constantly calculate the 1-D  $\leftrightarrow$  2-D transformations
  - \* allocate additional pointer array, to row beginnings  $\Rightarrow$  allow for 2-D array syntax
- Options for allocation of 2-D space
  - \* one contiguous allocation
  - \* separate allocations for each row
  - \* pros and cons
    - ★ different length rows/changing a row's length
    - ★ assumption that element `[row][0]` follows `[row - 1][num_cols - 1]` in memory
    - ★ availability of contiguous memory
    - ★ multiple `new()/delete()` pairs

## 2-D Free Store Allocation—Implementation

```
short **spp = new (short *) [num_rows]; // why ptrs from FS?
 // check for success! (here and below)
spp[0] = new short[num_rows * num_cols];
for (row = 1; row < num_rows; row ++)
 spp[row] = spp[row - 1] + num_cols;

// ... use the 2--D array, e.g.: spp[3][7] = ...

delete [] spp[0];

// or, two for-loops, for individual row new()/delete():
// for (row = 0; row < num_rows; row ++)
// spp[row] = new short[num_cols];
// for (row = num_rows - 1; row >= 0; row --)
// delete [] spp[row]; // note order (-- why?)

delete [] spp;
```

# Functions

---

- ⇒ Basics
- Prototypes
  - Arguments
  - Special Functions

# Avoiding Repeated Code

---

```
// ...
sum = 0;
for (index = 0; index < num_weights; index ++)
 sum += weights[index];
sum /= num_weights;
cout << "Average weight: " << sum << endl;
```

```
// same 5 lines for heights array, num_heights
// same 5 lines for grades array, num_grades
// same 5 lines for iqs array, num_iqs
```

- Cutting-and-pasting and duplicate code  $\Rightarrow$  error-prone

Repeated similar code  $\Rightarrow$  function.

# Functions—More reasons

---

- Readability ( $\leq \approx 30$  lines)
- Localized code—eases maintenance
- Can be used by other functions
- Rule of thumb: When in doubt, break it out.

Multiple files: once broken into functions:

- Combine similar functions in one file
- Can be used by other programs and programmers

How do we work with multiple files? See later.

# Definitions vs. Declarations

---

- Function *definition*
  - \* return type
  - \* name (perhaps with operator)
  - \* argument list
  - \* body = block of statements (always in {}s)
  - \* not allowed within another function definition
- Function prototype = function *declaration*
  - \* a “;” instead of block
  - \* allows for function usage until end of block or file
  - \* allowed within another function definition
- Program necessity:  $\exists!$  definition for each function

# Functions

---

- Basics
- ⇒ Prototypes
- Arguments
- Special Functions

# Strong Type Checking

---

- Compiler type checks arguments and return type, i.e., it's prototype or signature

```
float c2f(float degrees_in_celsius) {
 return (degrees_in_celsius * 1.8f + 32.f);
}
```

```
float pretty_cold = c2f(-10.f); // exact match
```

- What should happen here?

```
float as_cold_as = c2f(-40,3);
```

- Two options:
  - \* compile time error
  - \* run time error (by avoiding compile-time check)

Let the compiler do the work.

# Argument Conversions

---

- Explicit conversions are okay—and *highly* recommended

```
int pretty_hot = (int) c2f((float) 40);
```

- Implicit conversions are okay—sometimes

```
int pretty_nice = c2f(25.2); // okay
float kinda_cold = c2f("zero"); // error
```

- Advanced: one of these will not work (which one? why?)

```
void foo(double *p) { /* ... */ }
```

```
// ... later:
```

```
double *const p1 = 0; const double *p2;
foo(p1); foo(p2);
```

Be explicit about conversions.

# Return Values

---

- Allowed:
  - \* base types (int [default], double, ...)
  - \* pointers (here, a pointer to a short is returned)  
`short *foo(int num) { /* ... */ }`
  - \* references (recall: must refer to another existing object)  
`long& date(char *date_str) { /* ... */ }`
  - \* user-defined (enumerated, later: classes)
  - \* void
- Not allowed: arrays and functions

Be explicit: always supply a return statement.

# Functions for Printing

---

```
print_avg(float avg) {cout << "avg is: " << avg;}
// ... later in main()
cout << "the info: " << print_avg(4.5f) << endl;
// produces:
```

the info: avg is: 4.5260795864

- Q: "260795864"? A: garbage returned from `print_avg()`
  - \* invocation without "()" prints address of `print_avg()`
- should return `int` (e.g., 1, note warning)  $\Rightarrow$  but will be printed
- returning `void`  $\Rightarrow$  compiler error
- problem: we do not want to print the *function* value
- solution: remove function call from `cout` statement
- later: OOP for daisy-chaining `cout`

# Multiple Return Values

---

- Wanted: to return an  $(x, y)$  coordinate  $\Rightarrow$   
but we can only return one *thing*
- How do we do this?
- Three options:
  - \* Use global variables  $\Rightarrow$ 
    - ★ leads to unintuitive *side effects*
    - ★ error is not localized
  - \* Later: combine information together with classes
  - \* Use pointers and references in argument list

And how is this done? ...

# Functions

---

- Basics
- Prototypes
- ⇒ Arguments
- Special Functions

# Argument List

---

- With return value, is function's public interface

```
float average_rainfall(int *countries, int num_countries);
double sin(const double);
```
- Argument *name*
  - \* not needed for prototype declaration (why?)
  - \* unless obvious, include for documentation
- “...” at end = zero or more arguments, types unknown, e.g.:

```
int printf(const char *format, ...);
```
- Use sensible names for variables and function names
- Memory allocated for arguments and return value
  - \* Q: how much for functions above?

# Avoiding Arguments and Return Values

---

```
void input_info(void) { /* ... */ }
void process_database(void) { /* ... */ }
void output_statistics(void) { /* ... */ }
...
```

- Avoiding communication between functions
- Information passing must occur in another fashion
- Possibilities
  - \* global data and scope—bad news!! (see later)
  - \* information stored in external data files
    - ★ at least have file names as arguments
    - ⇒ do not hardwire them!

# Default Values

---

- What if `num_countries == size_UN` almost always?

- Functions can be called with fewer arguments

- Missing arguments receive default values

```
float average_rainfall(int *countries,
 int num_countries = size_UN);
```

- Only works for last argument(s) (i.e., no skipping)

```
void calibrate(short *values, int num_vals,
 short center = 0);
```

can be called:

```
short data[] = {6, 9, -2};
const int dsize = sizeof(data) / sizeof(data[0]);
calibrate(data, dsize, (short) 5);
calibrate(data, dsize); // same as: ... (data, dsize, 0);
```

# Argument List Order

---

- How to order the arguments?
- Need to think through for most common default-valued variables
- Different declarations can have different default value schemes

```
int new_window(int rows = 24, int cols = 80,
 char background = ' ');
```

```
char err = (char) new_window(24, 80, '*');
int j = new_window(12, 50); // new_window(12, 50, ' ');
new_window(8); // new_window(8, 80, ' ');
if (new_window() != OKAY) // new_window(24, 80, ' ');
 report_error();
```

# Passing Arguments

---

- C++ default method (*always* in C): pass-by-value
- Method: *copy* of value placed on the stack
- Argument of calling function remains unchanged

```
void increment(int thing) {thing ++; return;}
int i = 6;
increment(i);
// i == 6
```

What if we want to change the variable?

# Passing by Pointers

---

- One method: use pointer to send lvalue

```
void increment_p(int *thing) {(*thing) ++; return;}
int i = 6;
increment_p(&i);
// i == 7
```

- Also helps for speed, for large arguments
- Note: even pointer is by value, but lvalue of argument
- Problem: awkwardness of dereferencing

Can we have our cake and eat it too?

# Passing by Reference

---

- Another method: send an *alias* to same variable

```
void increment_r(int& thing) {thing ++; return;}
int i = 6;
increment_r(i);
// i == 7
```

- Aside from “&” in parameter list, same syntax as pass-by-value
- Low-level mechanism: passing an address ⇒
  - \* can change argument itself
  - \* still save on speed
- What if do not want to allow change? ⇒  
Declare parameter as `const`, e.g.  

```
double sin(const double& angle);
```

# Array Arguments

---

- Always passed as the address of the first element, i.e.:

```
void stats(int data[100]);
```

is the same as:

```
void stats(int *data);
```

- Two implications:

- \* size of array is unknown

- \* actual array argument is changed, *not* a copy

```
data[6] = -4;
```

- Can protect with “const”

```
void stats(const int *data, int length);
```

Care must be taken to *not* change array.

# Functions

---

- Basics
  - Prototypes
  - Arguments
- ⇒ Special Functions

# Recursion

---

- Recursive function: it calls itself
- Appropriate for iterative processes where each step mimics another, e.g., tree searches
- E.g., factorial calculation:  $n! = n(n - 1)!$ 

```
// why a long?
unsigned long factorial(unsigned int number) {
 if (number > 1)
 return (number * factorial(number - 1));
 return 1ul; // for 0 or 1
}
```
- Matches the problem (i.e., the mathematics) nicely

# Recursive vs. Iterative Functions

---

```
// iterative version
unsigned long factorial_iter(unsigned int number) {
 unsigned long product;
 for (product = 1ul; number > 1; number --)
 product *= number;
 return product;
}
```

- Recursive vs. iterative
  - \* smaller
  - \* less complex
  - \* but, slower—due to additional function invocations

Unless otherwise required, do what is most natural.

# inline Functions

---

- Normally, stack used heavily for function invocation
- `inline` = *suggestion* to compiler to expand in place  $\Rightarrow$  precede invocation with *definition* (why?)
  - \* saves on speed, no jump or stack usage
  - \* for small, simple, often-called functions

```
inline int min(int a, int b) {
 return ((a < b) ? (a) : (b));
} // note: min() can call a complex foo()
```

```
int min_grade = min(grade1, grade2); // expands to:
// int min_grade = ((grade1 < grade2) ?
// (grade1) : (grade2));
```

Use but don't abused.

# Overloaded Function Names

---

```
int min(int, int);
```

```
double min(double, double);
```

```
int min(const int *iarray, int num_elements);
```

- For functions of similar operations, e.g.: (1 + 3) vs. (1. + 3.)
- Avoid confusing name mangling (imin, dmin, iamin, ...) ⇒  
let compiler do it (\_min\_\_Fii, L\_min\_\_Fdd, \_min\_\_FPii)
- Same name, but different number/types of arguments
- Return values and typedefing do not disambiguate
- *Many* rules regarding resolving, matching, promotions, conversions, multiple arguments, scope, ...

Moral: cast explicitly.

# Function Templates—Why?

---

```
int min(int a, int b) {return a < b ? a : b;}
```

```
double min(double a, double b) {return a < b ? a : b;}
```

- Point: avoid repeated code

- Beware of using CPP *text* substitution

```
#define min(a, b) ((a) < (b) ? (a) : (b))
```

which causes

```
int i = min(*(ip++), 46);
```

to increment `ip` twice (ouch!)

- Want to *instantiate* a real function for each type necessary

Enter function templates . . . .

# Function Templates

---

```
template <class T>
```

```
T min(T a, T b) {return a < b ? a : b;}
```

- Calling the desired function (*instantiation*)

```
short s1, s2, s3 = min(s1, s2);
```

- Each type parameter (<class T, class U>)
  - \* must appear as an argument type of the function
  - \* can be used to define variables in function
- Again lots of rules regarding matching, type binding, ...
- Can also overload function templates ⇒  
powerful, but watch it!

# Advanced: const Template Types

---

For the following function template definitions:

```
void foo1(T& p1, T& p2) { ... }
void foo2(T& p1, const T& p2) { ... }
```

why do the following fail?

```
double *dp;
const double *cdp;
double *const dcp = 0;
```

```
foo1(dp, cdp); // fails
foo1(dp, dcp); // fails
foo2(dp, cdp); // fails
foo2(dp, dcp); // okay
```

- Note: without the first argument, they all succeed

# Multiple Files

---

- ⇒ Scope
- Compiling and Linking
  - Header Files
  - Preprocessor

# Type and Function Scope

---

- Once upon a time, just `main()` ... then functions ...
- Visibility (types [typedefs and enums] outside of functions):
  - \* from definition until end of file
- When possible, define types inside functions or even blocks
  - \* limit visibility
- Functions
  - \* To access early (or from another file) ⇒  
*declare* the function
  - \* To *hide*—i.e., inhibit declaration elsewhere ⇒  
define as `static`

What about scope of data?

# Data: File vs. Local Scope

---

```
float ff(int i) { /* ... */ } // function-local i
int i = 7; // file scope i
double df() {
 int i; // function-local i
 for (i = 0; /* ... */) { /* ... */ }
 while (/* ... */) { int i = 0; /* ... */ } // block-local i
}
```

- Four *distinct* variables named “i”
- Definition placement, default initialization
  - \* file scope: outside of all functions, initialized to zero
  - \* local scope: inside a function, (usually) uninitialized
- Life-span = allocation/deallocation
  - \* file scope: beginning/end of program
  - \* local scope: (usually) definition → surrounding “}”

What about visibility?

# File Scope Visibility

---

- From definition until the end of file
- To access “global” variable early ⇒  
*declare with `extern int i;`*
  - \* no storage allocated
  - \* definition of local `i` with same scope—prohibited
  - \* do not initialize (only for definition)
- To access “hidden” file scope `i` with local `i` defined ⇒  
Use `::i`
- To *hide*—i.e., not allow `extern` of—a file scope variable from
  - \* above in this file
  - \* other files

```
static int i; // definition for internal linkage
```

# Vulnerability of Global Variables

---

- Vast visibility vulnerability  $\Rightarrow$  any function can change a (non-static) global variable
- Not a problem for `const` variables
- Leads to non-localization of errors/changes in global data
- If a global is corrupt, *any* function could have caused it
- Also, if a function is misbehaving, its effect is
  - \* not limited to its local data and parameters
  - \* but also global data

Avoid globals as much as possible.

# Local Scope

---

- Within functions or blocks, uninitialized
- Life-span is block-local
- Therefore, do not rely on address of, or reference to, local variable after block termination, even as return value
- Suggest `register` for heavily used variables, e.g., loop indices
- Use `static` for permanent storage
  - \* e.g., to count (specific) function invocations (alter.?)
  - \* still allocated upon first encounter of definition
  - \* initialized only once (by default, to 0)
  - \* deallocated at end of program
  - \* scope (i.e., visibility) is *not* changed

# Multiple Files

---

- Scope
- ⇒ Compiling and Linking
- Header Files
- Preprocessor

# Library and Application Programmers

---

- Multiple source code files—get used to it, allows for
  - \* code re-usage
  - \* minimal recompilation upon code changes
- A functional approach to the players
- Library programmer: code for many programmers
  - \* `matrix.C`: function (and global variable) definitions
- Application programmer: specific application
  - \* `fft.C`: including `main()` function definition
  - \* uses (calls to) functions defined in `matrix.C`
- User: the one who runs the program

# Compilation vs. Linkage

---

- Compilation
  - \* C++ .C file is reviewed for syntax
  - \* *What* (type) is each item?
  - \* Is everything understood? Okay C++ syntax?
  - \* `g++ -c -Wall a.C b.C c.C` (generates 3 .o files)
  - \* Equivalent to compiling separately
- Linkage (= “loading”, ld command)
  - \* putting the whole story together
  - \* Does each item have a *unique* definition?
  - \* *How* is each function performed?
  - \* `g++ -o foo a.o b.o c.o`

# Multiple Files

---

- Scope
- Compiling and Linking
- ⇒ Header Files
- Preprocessor

# Role of Header files

---

- Contain information which *might* be needed in several files
  - \* e.g.: `iostream[.h]`
  - \* keeps things consistent; compiled *only* via `.Cs`
  - \* not for info needed in one `.C` only
- Wrap all `[.h]`s (e.g., `calculator[.h]`) to avoid multiple inclusion

```
#ifndef CALCULATOR_H
#define CALCULATOR_H
...
#endif
```
- Type definitions
  - \* enumeration definitions
  - \* `typedefs`
  - \* later: class definitions

# Global Info in Header Files

---

## Global variables and functions

- Declarations only (note: multiple declarations are no problem)
  - \* without declarations → cannot use items
  - \* with definitions → linkage receives multiple definitions
- Exceptions: `static` (why?) definitions needed for compilation
  - \* global, non-aggregate constants—automatically `static`
  - \* global, aggregate constants, add: `static`
  - \* `inline` functions—automatically `static`
  - \* function templates
    - ★ note: even not `inline`
    - ★ add “`static`”: `g++`—✓, `SGI`—× (automatic)

Definition: if not here—where?

# Various Programmers—Revisited

---

- Library programmer: code for many programmers
  - \* `matrix.C`: function and global variable definitions
  - \* `matrix.h`: function and global variable declarations (`#included` by `matrix.C`), interface for other programmers
  - \* deliverables: `matrix.h` and `matrix.o` (why not `matrix.C`?)
- Application programmer: specific application
  - \* invokes functions declared in `matrix.h`
  - \* for compilation *only*: `#includes` `matrix.h`
  - \* for linking *only*: `matrix.o`
- E.g.: `math[.h]`, `libm.a` (`math.o`), “`-lm`” at *end* of `g++` linkage

# More Possible Errors

---

- Compilation

- \* bad return value from function

```
foo.C:13: warning: 'return' with no value,
in function returning non-void
```

- \* no function declaration—missing header file #include?

```
foo.C:8: warning: implicit declaration of
function 'int some_func(...)'
```

- Linking (loading, “ld”)

- \* applies to functions and data

- \* without all object (“.o”) files or libraries

```
ld: Undefined symbol
```

```
_some_func
```

```
collect2: ld returned 2 exit status
```

- \* with too many definitions

```
ld: _some_func: multiply defined
```

# Multiple Files

---

- Scope
  - Compiling and Linking
  - Header Files
- ⇒ Preprocessor

# Motivation

---

- Avoid duplicate code
  - \* easier to maintain
  - \* less error-prone
- Primary function of preprocessor (`cpp`): (conditional) text replacement and insertion
- Directives start with “#” (in first column), happens before any processing starts

Can be very powerful.

# Text Replacement

---

- Standard C method for global (numeric) constants: macros

```
#define M_PI 3.14159265358979323846
```
- Standard C++ method for global (numeric) constants: consts

```
const double M_PI = 3.14159265358979323846;
```
- Latter method allows for addressing and debugging of M\_PI
- Macros can take arguments, looking function-like

```
#define BITS(type) (BITSPERBYTE * (int) sizeof(type))
#define MAX(a, b) ((a > b) ? (a) : (b))
... // surround def. in parens (why?)
unsigned int twice_bits_in_short = 2 * BITS(short);
float max_plus_10 = MAX(grade1, grade2) + 10;
```
- No space before first "(" in macro *definition*

# Text Insertion and Conditional Directives

---

- One can include entire files as if they are here

```
#include <math> // system header file
#include "calculator.h" // programmer-supplied
using namespace std;
```

- System header files: looked for in system directories
- *Never* #include a .C file
- Conditional directives: #if, #ifdef, #ifndef, #else, #endif
  - \* e.g.: good way to comment out many lines:

```
#if 0
... // no worry about /* ... */ in the middle
#endif
```