# C Minicourse

## Introduction to C

Yeah, C rulez!

Because creepy, old kernel hackers use C!!!
(Ken Thompson and Dennis Richie)

# Part 0: Overview

- What is C and why should I care?
- Differences from Java and C++
- Overview of a C program

# This is C!

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("Hello, world!\n");
  return 0;
}
```

# What is C?

- Low level programming language used primarily for implementing operating systems, compilers, etc.
- Great at:
  - interfacing with hardware
  - efficiency
  - large, entrenched user-base
- Not so great:
  - long development times
  - writing cross-platform C can be hard
  - notorious for hard-to-find bugs

# Java *vs* C *vs* C++

| Java | C | C++ |
|---|---|---|
| Object-Oriented | Procedural | Object-Oriented |
| Memory-Managed (automatic garbage collection) | Manual memory management (very prone to bugs) | Manual memory management (still very prone to bugs) |
| Good for high level or quick programs | Good for very low level programs where efficiency and/or interfacing with hardware are necessary | Good for everything that C is plus some (that's where the whole "++" thing comes in..) |
| Cross platform by design | Can be cross-platform, but it's not always easy | Can be cross-platform, but it's not always easy |
| Lame 無趣 | Not lame | Not lame |

- As you can see, C and C++ are very similar and much more not lame than Java

# Overview of a C Program

- Split into source files that are compiled separately
  - In Java, `.java` files are compiled into `.class` files
  - In C, `.c` files are compiled into `.o` (object) files
    - Generally done with the help of the `make` utility

- Define interface definitions in `.h` (header) files
- Define implementation in source files
  - Typically `.c` for C sources and `.cpp` for C++

# Part 1: The Basics

- Data types
- Structs and unions
- Arrays
- Strings
- printf()
- Enums
- Typedef
- Pointers

# Basic Data Types

- Variables declared with *data types*
  - e.g. int, char, float, long
    ```
    int a, b, c;
    a = b = c = 3;
    ```

- <u>No</u> boolean data type
    ```
    while(true)   →   while(1)
    ```

# Structs (1)

- Special data type, groups variables together

**To declare:**

```
struct sesamestreet {
        int     number;
        char    letter;
}; /* note the semicolon */
```

**To set values:**

```
struct sesamestreet st1 = { 3, 'a' };
/* the order must be correct */

struct sesamestreet st2;
st2.number = 7;
st2.letter = 'm';
```

# Unions

- Like a struct, but only *one* member is allowed to have a value.
- You almost never need to use these

```
union grade {
        int     percent;
        char    letter;
} ;

union grade stud1 = {100, 'a'}; /* not allowed */

union grade stud2 = {'a'}; /* unclear */

union grade stud3;
stud3.percent = 100;   /* okay! */
```

# Arrays

o Declaration is similar to Java

o Arrray size must be known at compile time (size should never be specified by a variable)

```
int nums[50];
nums[10] = 123;

/* multidimensional arrays */
int morenums[10][10];
morenums[0][3] = nums[1];

/* initialized array (size declared
implicitly) */
int somenums[] = { 1, 2, 3 };
```

# Strings (or lack thereof)

- A string in C is a `char` array.

```
char firstname[10];
char fullname[20];

firstname = "george"; /* won't work! */
fullname = firstname + "bush"; /* won't work! */
```

- `char` arrays and `char*` variables are similar and can be confusing

```
const char *georgeStr = "george"; /* works fine */
const char georgeStr[] = "george"; /* won't work! */
```

- Null termination is important.

# `printf()`, your new best friend

- Equivalent of `System.out.println()`

```
#include <stdio.h>

char *name = "yoda";
int age = 900;

printf("name: %s     age: %d\n", name, age);
```
**OUTPUT →**     name: yoda     age: 900

- Use *format specifiers* for printing variables

```
%s - string
%d - int
%f - float
```

# `enum`, a special type

- Used to declare and assign numerical values to a list of integer constants
- Start number defaults to '0'

```
enum { MON, TUE, WED, THU, FRI };
/* now, MON = 0, TUE = 1, etc.. */

enum color_t { RED = 3, BLUE = 4, YELLOW = 5 };
/* name your enum and use it as a type */

color_t myColor = BLUE;
/* can make statements like: if (myColor == RED) */

myColor = 5; /* is this allowed? */
```

# `typedef` (yay for less typing)

- Giving an alternative name to a type – especially useful with `struct`s and the STL (more to come, in C++ minicourse)
- `typedef <type> <name>`
- Example:

```
typedef int bool_t;

bool_t happy = 1; /* 'happy' is really an int */
do {
    happy = learnMoreC();
} while(happy);
```

# First look at pointers

- When a variable is declared, space in memory is reserved for it.
- A pointer represents the memory address of a variable, NOT its *value*.
- **&** ("address of" operator) gives the address of a variable
- **\*** (dereference operator) evaluates to the value of the memory referred to by the pointer

# Some pointer examples

```c
int a, b;

a = 3;
b = a;

printf("a = %d, b = %d\n", a, b);
printf("a located at %p, b located at %p\n", &a, &b);

int *myptr = &a;

printf("myptr points to location %p\n", myptr);
printf("the value that 'myptr' points at is %d\n", *myptr);
```

# Part 2: Memory Management and Functions

- more on pointers
- malloc/free
- dynamic arrays
- functions
- main arguments
- pass by: value / pointer
- function pointers

# Pointers and Arrays

```c
int main() {
    int array[10];
    int* pntr = array;
    for(int i=0; i < 10; i++) {
        printf("%d\n", pntr[i]);
    }
    return 0;
}
```

- We can get a pointer to the beginning of an array using the name of the array variable without any brackets.
- From then on, we can index into the array using our pointer.

# Pointer Arithmetic

```c
int main() {
    int array[10];
    int *pntr = NULL; // set pointer to NULL
    for(pntr = array; pntr < array + 10; pntr++) {
        printf("%d\n", *pntr); // dereference the pntr
    }
    return 0;
}
```

- We can "increment" a pointer, which has the effect of making it point to the next variable in a array.
- Instead of having an integer counter, we iterate through the array by moving the pointer itself.
- The pointer is initialized in the for loop to the start of the array. Terminate when we get the tenth index.

# void *

- A pointer to nothing??
- NO!! A pointer to *anything*

```c
int main() {
    char c;
    int i;
    float f;
    void *ptnr;

    pntr = &c;   //OK
    pntr = &i;   //OK
    pntr = &f;   //OK
    return 0;
}
```

- All pointers are the same size (typically 32 or 64 bits) because they all store the same kind of memory address.

# when to pass by pointer

- When declaring function parameters, you can either pass by value or by reference.
- Factors to consider:
  - How much data do you have?
  - Do you "trust" the other functions that will be calling your function.
  - Who will handle memory management?

# Stack vs. Heap

Stack vs. Heap

- Both are sources from which memory is allocated
- Stack is automatic
  - Created for local, "in scope" variables
  - Destroyed automatically when variables go "out of scope"
- Heap is manual
  - Created upon request
  - Destroyed upon request

# Two ways to get an `int`:

- **On the Stack:**

```
int main() {
    int myInt; /* declare an int on the stack */
    myInt = 5; /* set the memory to five        */
    return 0;
}
```

- **On the Heap:**

```
int main() {
    int *myInt = (int*) malloc(sizeof(int));
                        /* allocate mem. from heap */
    *myInt = 5;         /* set the memory to five  */
    return 0;
}
```

# A closer look at `malloc`

```
int main() {
    int *myInt = (int*) malloc(sizeof(int));
    *myInt = 5;
    return 0;
}
```

- `malloc` short for "memory allocation"
  - Takes as a parameter the number of bytes to allocate on the heap
  - `sizeof(int)` conveniently tells us how many bytes are in an `int` (typically 4)
  - Returns a *pointer* to the memory that was just allocated

# …but we forgot something!

○ We requested memory but never released it!

```c
int main() {
    int *myInt = (int*) malloc(sizeof(int));
    *myInt = 5;
    printf("myInt = %d\n", *myInt);
    free(myInt);  /* use free() to release memory */
    myInt = NULL;
    return 0;
}
```

○ `free()` releases memory we aren't using anymore
- Takes a *pointer* to the memory to be released
- Must have been allocated with `malloc`
- Should set pointer to `NULL` when done

# Don't do these things!

o `free()` memory on the stack

```
int main() {
    int myInt;
    free(&myInt);   /* very bad */
    return 0;
}
```

o Lose track of `malloc()`'d memory

```
int main() {
    int *myInt = (int*) malloc(sizeof(int));
    myInt = 0;   /* how can you free it now? */
    return 0;
}
```

o `free()` the same memory twice

```
int main() {
    int *A = (int*) malloc(sizeof(int));
    int *B = A;
    free(A);
    free(B);   /* very bad; shame on you */
    return 0;
}
```

# Dynamic arrays (1)

○ Review: static arrays

```
int main() {
    int array[512];                    // static array of size 512
    for(int i = 0; i < 512; i++) { // initialize the array to 0
        array[i] = 0;
    }
    /* alternative (faster) way of clearing the array to zero
    memset(array, 0, sizeof(int) * 512);
    */
    return 0;
}
```

○ **Problem:** Size determined at compile time. We must explicitly declare the exact size.

# Dynamic arrays (2)

o **Solution:** use `malloc()` to allocate enough space for the array at run-time

```
int main() {
    int n = rand() % 100;                      /* random number between 0 and 99 */
    int *array = (int*) malloc(n * sizeof(int));  /* allocate array of size */
    for(int i = 0; i < n; i++)      /* we can count up to (n-1) in our array */
        array[i] = 0;

    free(array);                               /* make sure to free the array! */
    array = NULL;
    return 0;
}
```

# Using `malloc` for "strings"

○ Since "strings" in C are just arrays of `char`s, `malloc` can be used to create variable length "strings"

```c
int main() {
    int n = rand() % 100;        // random number between 0 and 99
    char *myString = (char*) malloc(n);

    /* a char is one byte, so we don't have to use sizeof */
    for(int i = 0; i < n - 1; i++) {
        myString[i] = 'A';

    printf("myString: %s\n", string);
    myString[n - 1] = '\0';   /* must be null terminated! */
    free(myString);           /* make sure to free the string! */
    myString = NULL;
    return 0;
}
```

# mallocing structs

- We can also use `malloc` to allocate space on the heap for a `struct`

```c
struct foo_t {
    int     value;
    int[10] array;
};

int main() {
    /* sizeof(foo_t) gives us the size of the struct */
    foo_t *fooStruct = (foo_t*) malloc(sizeof(foo_t));
    fooStruct->value = 5;

    for(int i = 0; i < 10; i++) {
        fooStruct->array[i] = 0;
    }

    /* free the struct. DON'T need to also free the array */
    free(fooStruct);
    fooStruct = NULL;
    return 0;
}
```

# Functions

- So far, we've done everything in main without the use of functions
- But we have *used* some functions
  - `printf` in "hello world"
  - `malloc` for allocating memory
- C functions are similar to methods in Java
  - In Java, all methods are associated with a particular class. In C, all functions are global
  - In Java, you can overload methods. In C, you can't

# Our first function

```
int foo(int a, int b) {
    return a + b;
}

int main() {
    int sum = foo(1, 2);
    return 0;
}
```

o First we define the function "foo"
  • foo will return an int
  • foo takes two ints as arguments
o In main, we call foo and give it the values: 1 and 2
  • foo is invoked and returns the value 3

# Order matters

- In Java, the order that methods are defined doesn't matter
- Not true in C. Look what happens when we flip the order around:

```c
int main() {
    int sum = foo(1, 2);
    return 0;
}

int foo(int a, int b) {
    return a + b;
}
```

- Compiling the above code yields the following errors:

```
test.c: In function 'int main()':
test.c:2: error: 'foo' undeclared (first use of this function)
...
test.c: In function 'int foo(int, int)':
test.c:6: error: 'int foo(int, int)' used prior to declaration
```

# Definition vs. declaration

- In C, we can *declare* functions using a "function prototype" without *defining* what happens when the function is called
- This tells the compiler about the existence of a function so it will expect a definition at some point later on.

```c
int foo(int, int);      /* function prototype */

int main() {
   foo(1, 2);
   return 0;
}

int foo(int a, int b) {
   return a + b;
}
```

- The program will now compile

# Declarations

```
int foo(int, int);
```

- In a function declaration, all that is important is the *signature*
  - function name
  - number and type of arguments
  - return type

- We do not need to give names to the arguments, although we can, and should (for clarity)

- We can also declare `struct`s and `enum`s without defining them, but this is less common

# void* (*function) (point, ers);

- In C, we can treat functions just like types
- We can pass around "function pointers" just like regular pointers.
- Since all functions have memory addresses, we're really not doing anything different here.

```c
int foo(int a, int b) {
    return a + b;
}

int main() {
    int (*funcPntr)(int, int) = &foo;
    int sum = funcPntr(1, 2);   /* sum is now 3 */
    return 0;
}
```

# Let's break it down

```
int (*funcPntr)(int, int) = &foo;
```

- Function pointer syntax is weird, and not exactly consistent with the rest of C. You just have to get used to it.
- Starting from the left…
  - `int`: the return type
  - `(*funcPntr)`: the * denotes that this is a function *pointer*, not a regular function declaration. `funcPntr` is the name of the pointer we are declaring
  - `(int, int)`: the argument list
  - `= &foo`: we are assigning the address of `foo` to the pointer we just declared

# ...if you thought that was ugly

- What is being declared here?

```
void*(*func)(void*(*)(void*), void*);
```

- First person with the correct answer wins a delicious pack of **Skittles**®

# The Answer

```
void* (*func) (void* (*) (void*), void*);
```

- We are declaring a function pointer called `func`. `func` is a pointer to a function which returns a `void*` (un-typed pointer), and takes two arguments. The first argument is a function pointer to a function that returns a `void*` and also takes a `void*` as its only argument. The second argument of the function pointed to by `func` is a `void*`.
- Inconceivably contrived way to set a pointer to null:

```
void* foo(void* pnt) {
  return pnt;
}
void* bar(void*(*func)(void*), void* pntr) {
  return func(pntr);
}
int main() {
  void* (*func)(void*(*)(void*), void*) = &bar;   /* here's the declaration */
  void* pntr = func(&foo, NULL);                  /* pntr gets set to NULL */
  return 0;
}
```

# main arguments

- Remember those funky arguments that were passed to main in the hello world program?

```c
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello, world!\n");
    return 0;
}
```

- Main arguments are a simple way to pass information into a program (filenames, options, flags, etc.)

# `argc` and `argv` ?

- `argc` stands for "argument count"
  - An `int`, representing the number of arguments passed to the program
- `argv` stands for "argument vector"
  - A `char**`, meaning an array (size `argc`) of null terminated strings.
  - The first (0th) null terminated string is always the name of the executable

# Here's a program which simply prints out the arguments given to it

```c
#include <stdio.h>

int main(int argc, char **argv) {
    for(int i = 1; i < argc; i++) { /* start at 1, since arg 0 is name of program */
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

o   Enters the loop once for every argument
o   Prints out each argument, adds a newline

```
> gcc printArgs.c -o printArgs
> printArgs how are you today?
how
are
you
today?
```

# Part 3: Making Projects

- Header files
- <u>Makefiles</u>
- Input / Output functions
- `const` / `static` keywords
- `gcc` basics
- Common errors (segfaults, etc.)
- Tips for compiling code

# Header Files

- Use as an interface to a particular .c file. It's always a good idea to separate interface from implementation
- Put the following in header files
  - `#define` statements
  - `extern` function prototypes
  - `typedef` statements

# Example Header File

```c
#ifndef __HELLO_WORLD__
#define __HELLO_WORLD__

#define TIMES_TO_SAY_HELLO 10

extern void printHello(void);

#endif // __HELLO_WORLD__
```

# Header File Do's

- Always enclose header files with `#ifndef`, `#define`, and `#endif` preprocessor commands (called header guards)
  - This makes it ok to include the header file in multiple places. Otherwise, the compiler will complain about things being redefined.
- Make sure function prototypes match functions
- Double-check semi-colons; they are necessary after function prototypes

# Header File Don'ts

- Don't `#include` files if you can avoid it; you should use `#include` in .c files instead
- Never `#include` a .c file
- Don't **define** functions inside your header file unless you have a good reason.
- Don't use a single header file for multiple .c files

# Makefiles

- Makefiles save you typing
- Can reduce compile time by only compiling changed code
- Can be extremely complicated
- But existing Makefiles are easy to edit
- And it's easy to write quick-n-dirty ones

# Sample Makefile

```
EXEC_NAME = helloWorld

all: main.o
    gcc -g -o $(EXEC_NAME) main.o

main.o: main.c
    gcc -g -c main.c -o main.o

clean:
    rm -rf main.o $(EXEC_NAME)
```

# More Makefiles

- Rule names like "all" and "clean" and "main.o"

- "all" and "clean" are special – most rule names should be filenames

- Dependencies are listed after the colon following the rule name

- Operations are listed underneath; must be preceded by a tab

# Input / Output functions

- `printf` – takes a format string and an "unlimited" number of variables as arguments; prints to `stdout`

- `scanf` – similar to `printf`, but it takes pointers to variables and fills them with values received from `stdin`

- `read` / `write` – lower level system calls for getting and printing strings

# printf/scanf example

```c
#include <cstdio>

int main ()
{
    char  name[100];      /* to hold the name entered */
    int   age;            /* to hold the age entered */

    printf("Enter your name: ");
    scanf("%s", name);
    printf("Enter your age: ");
    scanf("%d", &age);
    printf("%s is %d years old\n", name, age);

    return 0;
}
```

# `printf/scanf` example cont.

(bold represents data typed in by the user)

```
> ./printfScanfExample
> Enter your name: Andy
> Enter your age: 1000000
> Andy is 1000000 years old
```

# `const`/`static` keywords

- Declaring a variable `const` tells the compiler its value will not change
- `static` limits the usage of a variable or function to a particular .c file
  - `static` functions cannot be declared extern in header files
  - `static` variables retain their values between function calls

# gcc basics

- gcc is C compiler
  - -Wall turns on all warnings
  - -g adds debugging info
  - -o specifies output filename
  - -c just builds a .o (object) file
    - does not link symbols into an executable

```
gcc -Wall hello.c -o helloWorld
```

# Common errors

- Segfault
  - Indicates a problem with memory access; dereferencing a `NULL` pointer can cause this
- Linker error
  - Usually a problem with a Makefile and / or `gcc` commands. Can often occur if you declare, but fail to define a function
- Multiple definitions of functions
  - Often caused by failure to include header guards (`#ifndef, #define, #endif`) in header files

This page intentionally left blank.
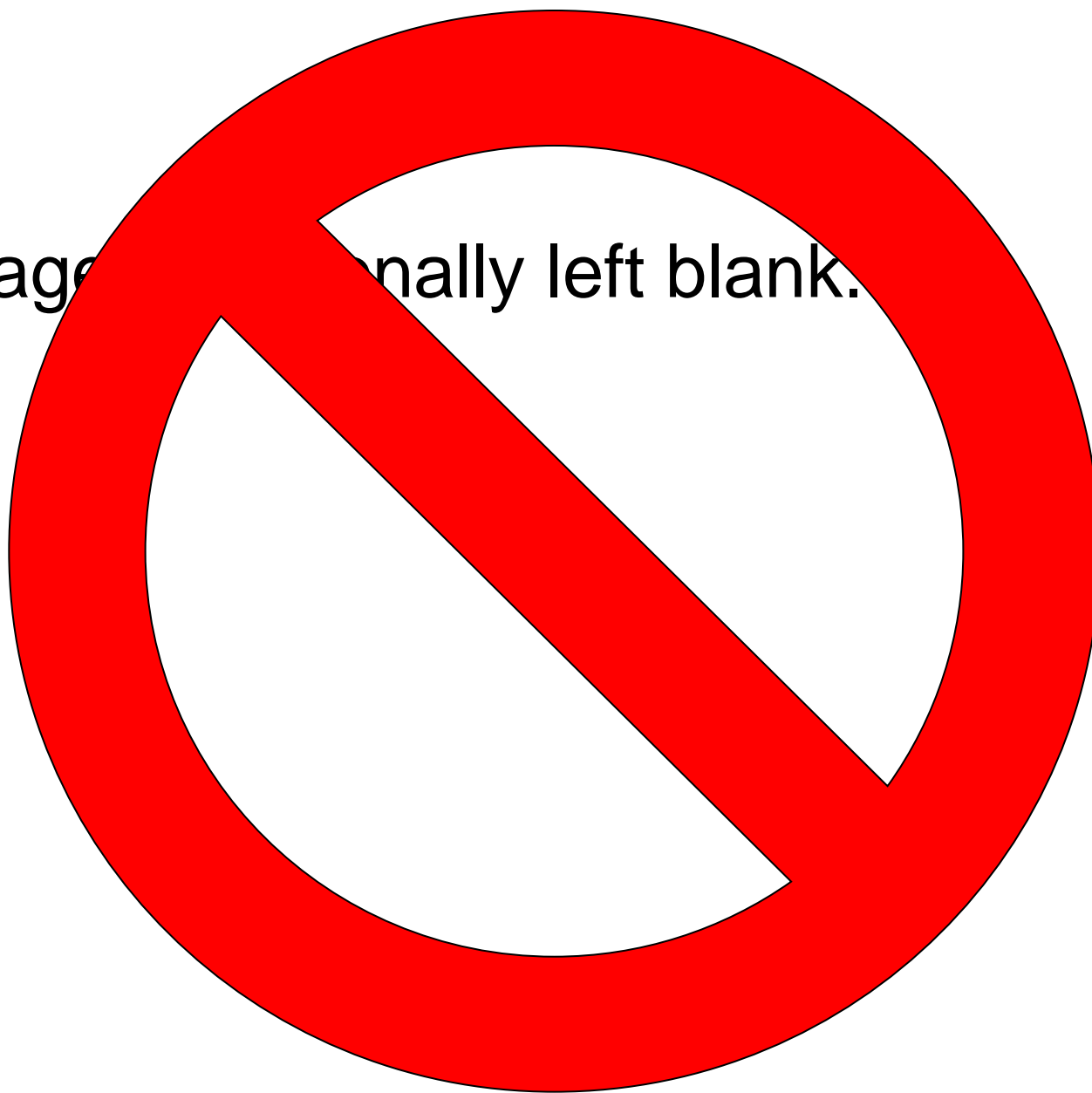
# A Crash Course in C++:
## Procedural (not Object-Oriented) Part

- In general, C++ was designed to be highly compatible with existing C programs.

- Keywords: addition of **class**, **const**, **delete**, **friend**, **inline**, **new**, **operator**, **private**, **protected**, **public**, **template**, **this**, **virtual**, etc.

# Comments

- Any text from // to the end of the current line is ignored.

- Use /* … */ for multi-line comments.

- The /* … */ comments do not nest.

# Input / Output

- The iostream header file must be included in all source code using stream input and output operations.

- The special object endl prints a newline character and flushes the stream buffer.

- Operations to the same stream can be chained.

# Input / Output (2)

- *#include <iostream.h>*
  *int i = 5;*
  *cout << i;*
  *cout << "i has the value " << i << "\n";*
    is the equivalent of
  *printf("i has the value %d\n", i);*
- *int i;*
  *char array[80];*
  *cin >> i >> array;*
    works like
  scanf("%d%s", &i, array);

# Input / Output (3)

- Data can be sent to a file instead of standard output.

  *#include <fstream.h>*

  *ofstream os("output.dat");*

  *os << "The value of pi is approxi. "*

  *<< 3.14159 << endl;*

# Type and Var Declarations

- A type or variable declaration in C++ can go anywhere that a statement can go. It does not have to be at the ***top*** of a block.

- Constants are prefixed by const.

  ***const double pi = 3.141592653589793846;***

# Scope Resolution Operator ::

- Bypass the search in the local and class scope and refer to a global name.

```
int x = 0;
main() {
        int x = 1; printf("::x = %d\n", ::x);          // 0
        {
                int x = 2; printf("::x = %d\n", ::x);   // 0
        }
}
```

# "inline" Functions

- **For the sake of efficiency, generally with short functions.**

- **Unlike a macro, an inline does not have side effects.**

    *#define toupper(c) { islower(c) ? c + 'A' - 'a' : c }*

    *toupper(\*p++)* expands to

    *{ islower(\*p++) ? \*p++ + 'A' - 'a' : \*p++ }*


*inline toupper(int c) { return islower(c) ? c + 'A' - 'a' : c; }*

# "inline" Functions (cont)

- "inline" functions are of internal linkage (i.e., definition scope is local to that file).

# "void" Pointers

*char *cp;* // integer pointer

*void *vp;* // generic pointer

*cp = vp;* // legal in C, illegal in C++

*cp = (char *) vp;*

*int foo();* is equivalent in C++ to *int foo(void);*

# "struct"

```
struct tree {
    struct tree *next;      // in C
};


struct tree {
    tree *next;             // in C++
};


struct A { struct B *p; };   // legal in C
struct B { struct A *q; };
```

# "struct" (cont)

*struct A { B *p; };*      // illegal in C++

*struct B { A *q; };*

solving the problem by

*struct B;*   // tell the compiler that B is a struct

*struct A { B *p; };*

*struct B { A *q; };*

# "union"

- *struct {*
  *union {*
  *int i;*
  *double f;*
  *}*
  *} num;*
- Refer *"num.i"* and *"num.f"* without calling a name for the union field, as would be required in C.

# Dynamic Memory Allocation

- Using the "**new**" and "**delete**" keywords. You can call "malloc" and "free", but you should <span style="color:red">not</span>.

  *int *ip = new int; delete ip;*

  *int *ia = new int [10]; delete [10] ia;*

  *or delete [] ia;*    /* <span style="color:red">not</span> recommended */

# Default Arguments

- ***int foo(int a = 1, int b = 2);***

  /* a call to *foo()* is the same as *foo(1, 2);*

  a call to *foo(3)* is the same as *foo(3, 2);*

  a call to *foo(3, 4)* is just what it says */

- Only <span style="color:blue">trailing</span> arguments can be supplied in this way. Therefore,

  ***int foo(int a = 1, int b);*** is <span style="color:red">illegal</span>.

# References

- ***int x;***        // an int
  ***int &rx = x;***  // a reference to an int
  /* translated as "reference to" rather than
     "pointer to". */

- All reference variables must be made to reference a real variable at declaration time when using the ***assignment-like*** syntax.

# References (cont)

*int \*p = &rx;*     // puts the address of *x* into *p*

*int \*p = rx;*     // illegal

*int \*q = p;*

*int \*&rp = p;*
  // *rp* is a reference to an int pointer

*int &\*rp;*     // illegal

This page intentionally left blank.
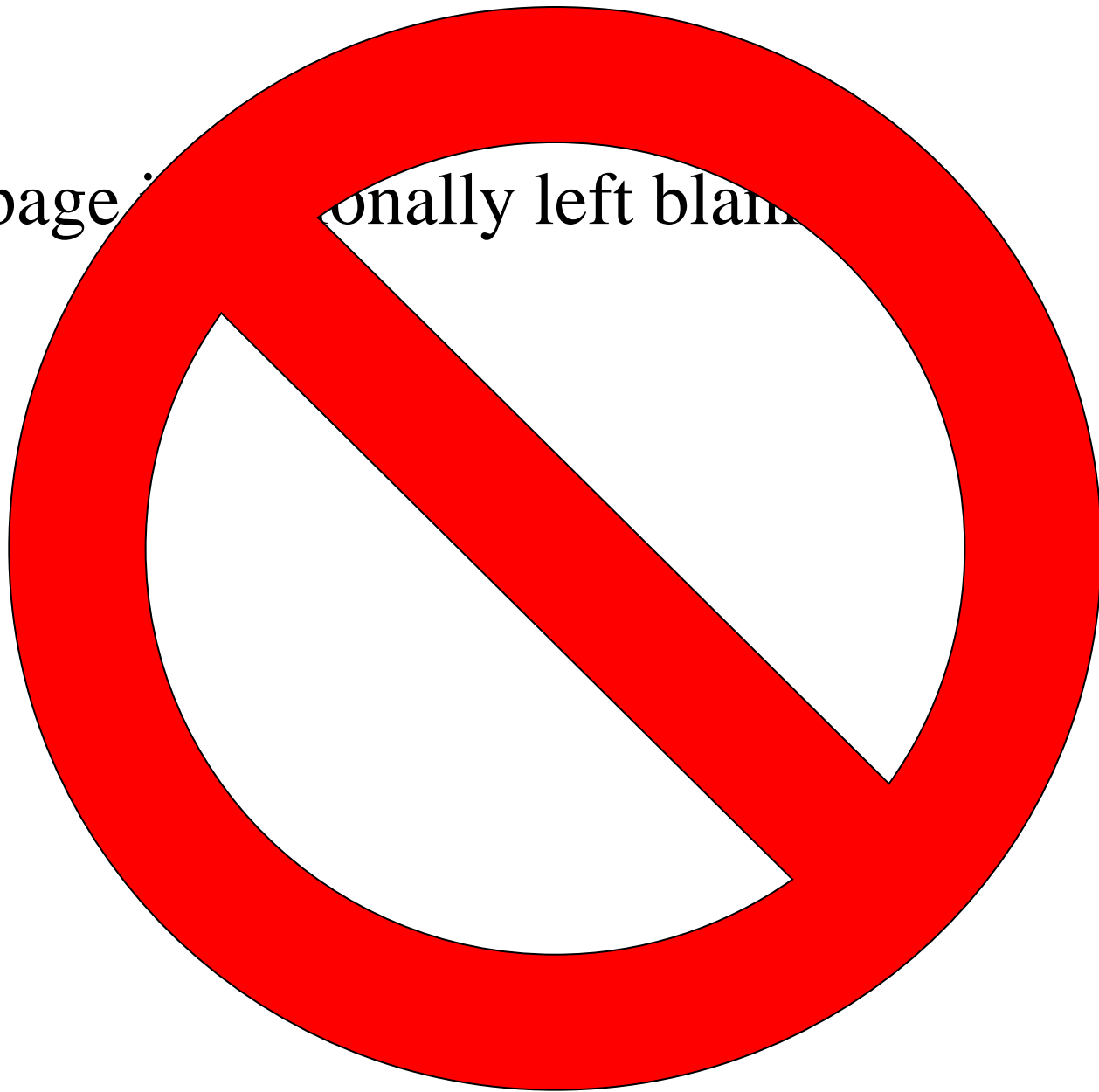
# Chapter 2 - C++ As A "Better C"

**Outline**

# 2.1    Introduction

- – Object based programming (classes, objects, encapsulation)
- – Object oriented programming (inheritance, polymorphism)
- – Generic programming (class and function templates)

# 2.2    C++

- C++
    - Improves on many of C's features
    - Has object-oriented capabilities
    - Superset of C
        - Can use a C++ compiler to compile C programs

# 2.3   A Simple Program: Adding Two Integers

- Differences
  - **`<iostream>`** - input/output stream header file
  - Return types - all functions must declare their return type
    - C does not require it, but C++ does

# 2.3   A Simple Program: Adding Two Integers (II)

- Output
  - **std::cout << "hi";**
    - Puts **"hi"** to **std::cout**, which prints it on the screen

- Input
  - **std::cin >> myVariable;**
    - Gets stream from keyboard and puts it into **myVariable**

# 2.3   A Simple Program: Adding Two Integers (III)

- **`std::endl`**
  - Stream manipulator - prints a newline and flushes output buffer

- Cascading
  - Can have multiple **`<<`** or **`>>`** operators in a single statement
    
    **`std::cout << "Hello " << "there" << std::endl;`**

# 2.4    C++ Standard Library

- C++ programs built from
  - Functions
  - Classes

```cpp
1  // Fig. 15.1: fig15_01.cpp
2  // Addition program
3  #include <iostream>
4
5  int main()
6  {
7      int integer1, integer2, sum;          // declaration
8
9      std::cout << "Enter first integer\n";  // prompt
10     std::cin >> integer1;                   // read an integer
11     std::cout << "Enter second integer\n"; // prompt
12     std::cin >> integer2;                   // read an integer
13     sum = integer1 + integer2;              // assignment of sum
14     std::cout << "Sum is " << sum << std::endl; // print sum
15
16     return 0;    // indicate that program ended successfully
17 }
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Outline

**1. Load <iostream>**

**2. main**

**2.1 Initialize variables integer1, integer2, and sum**

**2.2 Print "Enter first integer"**
  **2.2.1 Get input**

**2.3 Print "Enter second integer"**
   **2.3.1 Get input**

**2.4 Add variables and put result into sum**

**2.5 Print "Sum is"**
   **2.5.1 Output sum**

**2.6 exit (return 0)**
**Program Output**

# 2.6    Inline Functions

- ## Function calls
  - Cause execution-time overhead
  - Qualifier **inline** before function return type "advises" a function to be inlined
    - Puts copy of function's code in place of function call
  - Speeds up performance but increases file size
  - Compiler can ignore the **inline** qualifier
    - Ignores all but the smallest functions

```
inline double cube( const double s )
   { return s * s * s; }
```

- ## Using statements
  - By writing **using std::cout;** we can write **cout** instead of **std::cout** in the program
  - Same applies for **std::cin** and **std::endl**

# 2.7 References and Reference Parameters

- Reference parameter alias for argument
    - Use **&**

    ```
    void change(int &variable)
      {
            variable += 3;
      }
    ```

    - Adds 3 to the original variable input

# 2.7 References and Reference Parameters (II)

- Dangling references
  - If a function returns a reference to a variable, make sure the variable is **static**
    - Otherwise, it is automatic and destroyed after function ends

# 2.8 Default Arguments and Empty Parameter Lists

- If function parameter omitted, gets default value
  - If not enough parameters specified, rightmost go to their defaults


- Set defaults in function prototype

```
int myFunction( int x = 1, int y = 2, int z = 3 );
```

# 2.8   Default Arguments and Empty Parameter Lists (II)

- Empty parameter lists
  - In C, empty parameter list means function takes any argument
    - In C++ it means function takes no arguments
  - To declare that a function takes no parameters:
    - Write **void** in parentheses
    - Prototypes:

    ```
    void print1( void );
    void print2();
    ```

# 2.9    Unary Scope Resolution Operator

- Unary scope resolution operator (**::**)
  - Access global variables if a local variable has same name
  - Instead of **variable** use **::variable**

- Stream manipulators
  - Can change how output is formatted
  - **setprecision** - set precision for **float**s (default 6 digits)
  - **setwidth** - set field width

```
1  // Fig. 15.9: fig15_09.cpp
2  // Using the unary scope resolution operator
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8
9  #include <iomanip>
10
11 using std::setprecision;
12
13 using std::setw;
14
15 const double PI = 3.14159265358979;
16
17 int main()
18 {
19
20
21    cout << setprecision( 20 )
22         << "  Local float value of PI = " << PI
23         << "\nGlobal double value of PI = " << ::PI << endl;
24
25    cout << setw( 28 ) << "Local float value of PI = "
26
27         << setprecision( 10 ) << PI << endl;
28    return 0;
29 }
```

```
   Local float value of PI = 3.141592741012573242
Global double value of PI = 3.141592653589790007
   Local float value of PI = 3.1415927410
```

**Program Output**

# 2.10 Function Overloading

- ## Function overloading:
  - Functions with same name and different parameters
  - Overloaded functions should perform similar tasks
    - Function to square **int**s and function to square **float**s

    `int square( int x) {return x * x;}`

    `float square(float x) { return x * x; }`
  - Program chooses function by signature
    - Signature determined by function name and parameter types

```cpp
1  // Fig. 15.10: fig15_10.cpp
2  // Using overloaded functions
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int square( int x ) { return x * x; }
9
10 double square( double y ) { return y * y; }
11
12 int main()
13 {
14    cout << "The square of integer 7 is " << square( 7 )
15        << "\nThe square of double 7.5 is " << square( 7.5 )
16        << endl;
17
18    return 0;
19 }
```

**1. Define overloaded function**

**2. Function calls**

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

**Program Output**

# 2.11 Function Templates

- ## Function templates
  - Compact way to make overloaded functions
  - ```
    template < class T >  // or template< typename T >
    T square( T value1)
    {
        return value1 * value1;
    }
    ```
  - **T** replaced by type parameter in function call
    ```
    int x;

    int y = square(x);
    ```
    - If **int** parameter, all **T**'s become **int**s
    - Can use **float**, **double**, **long**...