Object-Oriented Design and Programming

Overview of Basic C++ Constructs

Outline

Lexical Elements

Identifiers: A sequence of letters (including '_') and digits. The first character *must* be a letter. Identifiers are case sensitive, *i.e.*, Foo_Bar1 is different from foo_bar1.

Reserved Words: Keywords that are *not* redefinable by the programmer, *e.g.*, **int**, **while**, **double**, **return**, **catch**, **delete**. There are currently 48 C++ reserved words.

Operators: Tokens that perform operations upon operands of various types. There are around 50 operators and 16 precedence levels.

Lexical Elements (cont'd)

Preprocessor Directives: Used for conditional compilation. Always begin with #, *e.g.*, **#include**, **#ifdef**, **#define**, **#if**, **#endif**.

Comments: Delimited by either /* */ or //, comments are ignored by the \

compiler. Note that comment of the same style do

```
#if 0
    ....
#endif
```

Constants and Literals: For strings, integers, floating point types, and enumerations, *e.g.*, "hello world", 2001, 3.1416, and FOOBAR.

The Preprocessor

- Less important for C++ than for C due to **inline** functions and **const** objects.
- The C++ preprocessor has 4 major functions:
  - *File Inclusion:*

```
#include <stream.h>
#include "foo.h"
```

  - *Symbolic Constants*:

```
#define SCREEN_SIZE 80
#define FALSE 0
```

  - *Parameterized Macros*:

```
#define SQUARE(A) ((A) * (A))
#define NIL(TYPE) ((TYPE *)0)
#define IS_UPPER(C) ((C) >= 'A' && (C) <= 'Z')
```

  - *Conditional Compilation*:

```
#ifdef __"cplusplus"
#include "c++-prototypes.h"
#elif _STDC__
#include "c-prototypes.h"
#else
#include "nonprototypes.h"
#endif
```


Variables, Functions, and Classes

- *Variables*
  - In C++ all variables *must* be declared before they are used. Furthermore, variables must be used in a manner consistent with their associated type.
- *Functions*
  - Unlike C, all C++ functions *must* be declared before being used, their return type defaults to int. However, it is considered good style to fully declare all functions.
  - Use **void** keyword to specify that a function does *not* return a value.
- *Classes*
  - Combines data objects and functions to provide an Abstract Data Type (ADT).

Definition and Declaration

- It is important in C to distinguish between variable and function declaration and definition:

  Definition: Refers to the place where a variable or function is created or assigned storage. Each external variable and function must be defined exactly *once* in a program.

  Declaration: Refers to places where the nature of the variable is stated, but no storage is allocated.

  Note that a **class**, **struct**, **union**, or **enum** declaration is also a definition in the sense that it cannot appear multiple times in a single compilation unit.

- Variables and function *must* be declared for each function that wishes to access them. Declarations provide sizes and types to the compiler so that it can generate correct code.

Compound Statement

- General form:

```
'{'
     [ decl-list ]
     [ stmt-list ]
'}'
```

- *e.g.*,

```
int c = 'A'; // Global variable
int main (void) {
     if (argc > 1) {
          putchar ('[');

          for (int c = ::c; c <= 'Z'; putchar (c++))
               ;

          putchar (']');
     }
}
```

- Note the use of the scope resolution operator :: to reference otherwise hidden global **int** c.

Iteration Statements

- C++ has 5 methods for repeating an action in a program:
  1. **for**: test at loop top
  2. **while**: test at loop top
  3. **do**/**while**: test at loop bottom
  4. *Recursion*
  5. *Unconditional Branch*: local (**goto**) and non-local (setjmp and longjmp)

for Loop

- General form

  for (<initialize>; <exit test>; <increment>)
  <stmt>

- The **for** loop localizes initialization, test for exit, and incrementing in one general syntactic construct.

- All three loop header sections are optional, and they may contain arbitrary expressions.

- Note that it is possible to declare variables in the <initialize> section (unlike C).

for loop (cont'd)

- *e.g.*,

```
for ( ; ; ); /* Loop forever. */

/* Copy stdin to stdout. */
for (int c; (c = getchar ()) != EOF; putchar (c));

/* Compute n! factorial. */
for (int i = n; n > 2; n--) i *= (n - 1);

/* Walk through a linked list. */
for (List *p = head; p != 0; p = p->next) action (p);
```


while Loop

- General form

  while (<condition>)
  <stmt>

- repeats execution of stmt as long as condition evaluates to non-zero

- In fact, a **for** loop is expressible as a **while** loop:

```
<initialize>
while (<exit test>)
{
    <loop body>
    <increment>
}
```

while Loop (cont'd)

- *e.g.*,

```
while (1); /* Loop forever. */

int c;
while ((c = getchar ()) != EOF)
      putchar (c);

i = n; /* Compute n! factorial. */
while (n >= 0)
      i *= --n;

/* Walk through a linked list. */
p = head;
while (p != 0) {
      action (p);
      p = p->next;
}
```


do while loop

- General form:

```
do <stmt> while (<condition>);
```

- Less commonly used than **for** or **while** loops.

- Note that the exit test is at the bottom of the loop, this means that the loop always executes at least once!

```
int main (void) {
      const int MAX_LEN = 80;
      char name_str[MAX_LEN];
      do {
            cout << "enter name (“exit” to quit)";
            cin.getline (name_str, MAX_LEN);
            process (name_str);
      } while (strcmp (name_str, "exit") != 0);
      return 0;
}
```


break and continue Statements

- Provides a controlled form of **goto** inside loops.

```
#include <stream.h>
int main (void) {
      /* Finds first negative number. */
      int number;
      while (cin >> number)
            if (number < 0)
                  break;
      cout << "number = " << number << "\n";
      // ...
      /* Sum up all even numbers, counts total numbers read
      int sum, total;
      for (sum = total = 0; cin >> number; total++) {
            if (number & 1)
                  continue;
            sum += number;
      }
      cout << "sum = " << sum << ", total = "
            << total << "\n";
}
```


Conditional Branching

- There are two general forms of conditional branching statements in C++:

- **if/else**: general method for selecting an action for conditional execution, linearly checks conditions and chooses first one that evaluates to TRUE.

- **switch**: a potentially more efficient method of selecting actions, since it can use a "jump table."

if Statement

- General form

```
if (<cond>)
     <stmt1>
[else
     <stmt2>]
```

- Common mechanism for conditionally executing a statement sequence.

```
#include <ctype.h>
char *character_class (char c) {
     if (isalpha (c)) {
          if (isupper (c))
               return "is upper case";
          else
               return "is lower case";
     }
     else if (isdigit (c))
          return "is a digit";
     else if (isprint (c))
          return "is a printable char";
     else
          return "is an unprintable char";
}
```


switch Statement

- General form

```
switch (<expr>) { <cases> }
```

- **switch** only works for scalar variables *e.g.*, integers, characters, enumerations.

- Permits efficient selection from among a set of values for a scalar variable.

```
enum symbol_type {
     CONST, SCALAR, STRING, RECORD, ARRAY
} symbol;
/* ... */
switch (symbol) {
     case CONST: puts ( "constant"); /* FALLTHRU */
     case SCALAR: puts ( "scalar"); break;
     case RECORD: puts ( "record"); break;
     default: puts ( "either array or string"); break;
}
```

- A **break** occurring inside a **switch** is similar to one occurring inside a looping construct.

C++ Arrays

- Arrays are a data type that consist of homogenous elements.

- A k-element one-dimensional array of ELEMENT type in C++ is a contiguous block of memory with size (k * **sizeof** (ELEMENT)).

- C array's have several distinct limitations:
  - All array bounds run from 0 to $k - 1$.
  - The size must be a compile-time constant.
  - Size cannot vary at run-time.
  - No range checking performed at run-time, *e.g.*,

```
{
     int a[10];

     for (int i = 0; i <= 10; i++)
          a[i] = 0;
}
```


Arrays (cont'd)

- Arrays are defined by providing their type, their name, and their size, for example, two integer arrays with size 10 and 1000 are declared as:

```
int array[10], vector[1000];
```

- Arrays and pointers are similar in C++. An array name is automatically converted to a constant pointer to the array's first element (only exception is **sizeof** array-name).

- Arrays can be initialized at compile-time and at run-time, *e.g.*,

```
int eight_primes[] = {2, 3, 5, 7, 11, 13, 17, 19};
int eight_count[8], i;
for (i = 0; i < 8; i++)
     eight_count[i] = eight_primes[i];
```

Multi-Dimensional Arrays

- C++ provides rectangular multi-dimensional arrays.
- Elements are stored in *row-order*.
- Multi-dimensional arrays can also be initialized, *e.g.*,

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
};
```

- It is possible to leave out certain initializer values...

Pointers

- A pointer is a variable that can hold the address of another variable, *e.g.*,

```
int i = 10;
int *ip = &i;
```

- It is possible to change `i` *indirectly* through `ip`, *e.g.*,

```
*ip = i + 1;
/* ALWAYS true! */
if (*ip == i) /* ...*/
```

- Note: the size of a pointer is usually the same as **int**, but be careful on some machines, *e.g.*, Intel 80286!
- Note: it is often possible to use *reference* variables instead of pointers in C++, *e.g.*, when passing variables by reference.

Passing Arrays as Parameters

- C++'s syntax for passing arrays as parameters is very confusing.
- For example, the following declarations are equivalent:

```
int sort (int base[], int size);
int sort (int *base, int size);
```

- Furthermore, the compiler will not complain if you pass an incorrect variable here:

```
int i, *ip;
sort (&i, sizeof i);
sort (ip, sizeof *ip);
```

- Note that what you really want to do here is:

```
int a[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
sort (a, sizeof a / sizeof *a);
```

- But it is difficult to tell this from the function prototype...

Character Strings

- A C++ string literal is implemented as a pointer to a NUL-terminated (*i.e.*, '\0') character array. There is an implicit extra byte in each string literal to hold the terminating NUL character.
- *e.g.*,

```
char *p; /* a string not bound to storage */
char buf[40]; /* a string of 40 chars */
char *s = malloc (40); /* a string of 40 chars */
```

```
char *string = "hello";
sizeof (string) == 4; /* On a VAX. */
sizeof ("hello") == 6;
sizeof buf == 40;
strlen ("hello") == 5;
```

- A number of standard string manipulation routines are available in the <string.h> header file.

Character Strings (cont'd)

- BE CAREFUL WHEN USING C++ STRINGS. They do not always work the way you might expect. In particular the following causes both str1 and str2 to point at "bar":

```
char *str1 = "foo", *str2 = "bar";

str1 = str2;
```

- In order to perform string copies you must use the `strcpy` function, *e.g.*,

```
strcpy (str1, str2);
```

- Beware of the difference between arrays and pointers...

```
char *foo = "I am a string constant";
char bar[] = "I am a character array";
sizeof foo == 4;
sizeof bar == 23;
```

- It is often better to use a C++ String class instead of built-in strings...