Chapter 3. The C in C++

C++ Object Oriented Programming
Pei-yih Ting
93/03 NTOU CS

Contents
- Creating functions
- Controlling execution
- Data types
- Scoping Rules
- Storage allocation
- Operators
- Explicit Type Casting
- Alias a type name
- Debugging
- Complex Data Type Definitions

Introduction
- C++ is based on C
  - The part of grammars for C is almost the same as ANSI C except only a few enhancements
  - This part of grammars can be used to do procedural programming (including very low level system programming; C was dubbed a high-level assembly language)
  - About 75% of C++ grammars
  - The other 25% of grammars in C++ is for supporting object oriented programming
  - An OOP program written in C++ uses all grammars both C part and non-C part but is very easy to tell apart from a procedural program written in C.
    It has intrinsically different program structure.

Creating Functions
- Function is the atomic unit of instructions in C/C++.
- Prototypes (forward declarations, declarations)
  ```cpp
  int translate(float x, float y, int z);
  int translate(float, float, int);
  ```
- Definitions
  ```cpp
  int translate(float x, float y, int z)
  {
    float r = y * z + 2.0 * x;
    if (r > 0)
      return 1;
    return 0;
  }
  ```
  Return value could be void.
  Parameter could be void.
Function Argument Passing
- Pass-by-value (call-by-value): the value of an actual argument is copied into the function’s formal parameter variable as the control is passed to the calling function.
  ```c
  double square(double x);
  ...
  int x, y;
  y = square(x);
  ```

**Basic types**
**Array**
**Pointers**

Controlling Execution
- C: 0 is false, non-zero is true
- Conditional expressions: a == b, c > 1.5…
- if-else
  ```c
  if (expression) statement
  ...
  if (expression) statement else statement
  ```
- while
  ```c
  while (expression) statement
  while (expression) statement do statement while (expression);
  ```
- for
  ```c
  for (initialization; conditional; step) statement
  for (initialization; conditional; step) statement
  ```
- break and continue
- switch
  ```c
  switch (selector) {case integral-value: statement; break; ... default: statement;}
  ```
- goto
- function call and recursion

Data Types
- Built-in data types
  - char, int, float, double
  - Ranges: limits.h and float.h
  - int may have different size of storage on different platforms
- Specifiers
  - long, short, signed, unsigned
- Pointers: storing memory locations or address or a variable or a function
  - & operator to extract the address of a variable
  - * dereference operator
  - Indirect access of variables or functions through pointers
    - Indirection means flexibility; let the same code have effects on different objects
    - Change “outside objects” from within a function
- User-defined data types (abstract data types, with behaviors)
  - array and struct

Arrays
- int a[100];
  ```c
  ...
  a[99]
  ```
  - or even have changeable names a[expression]
  - they occupies contiguous memory spaces
  - the value of a is the address of the first elements, the type of a is ‘int[100]’
  - &a is not suitably defined since a is not a variable, does not have an address; but many compiler take this value as the same as a
  - a[i], denotes one of these variable, is equivalent to *(a+i), *(i+a)
    - (or even i[a])
  - &a[i] denotes the address of the i-th variable a[i]
Arrays

- `int b[100][200];`
  - defines 100*200 variables for storing integers, their names are `b[0][0]`, ..., `b[99][199]`
  - they occupies contiguous memory spaces in the form of 100 repeated blocks of 200-element of `int`. (so called row major storage allocation)
  - each value of `b[0], b[1], ..., b[99]` is the starting address of an `int [200]` array, the type of `b[i]` is `int[200]`
  - `&b[i][j]` denotes the address of the `i,j`-th variable (`(i*200+j)`'s variable)

Storage Allocation

- Global variables:
  - defined outside all function bodies
  - allocated on .data segment
  - use `extern` to declare a global variable in another file
  - the scope of a `static` global variable is constrained in a file (file scope)

- Local variables: (automatic variables: automatically come into being when the scope is entered and going away when the scope closes)
  - defined inside a block or a function
  - allocated on the stack
  - register keyword makes the variable on the a register (a register variable can not be global or static)
  - `static` keyword makes the variable be allocated on .data segment and the lifetime of the variable extends over multiple entrances of this block
  - `const` qualifier tells the compiler “This never changes”
  - `volatile` qualifier tells the compiler “You never know when this will change”. Used with a multithreaded program.

- Dynamically allocated variables:

void *

- In C, dynamic polymorphism can be implemented using untyped pointers `void *`
- Ex. The qsort() stdlib function
  ```c
  void qsort(void *base, size_t num, size_t width, 
              int (*compare)(const void *elem1, const void *elem2));
  ```

Scoping Rules

- The scoping rules tell you where a variable is valid (visible). Sometimes the rules also tell you where a variable is create and where it gets destroyed.
- The lifetime of a variable starts from the creation of a variable till the destruction of a variable. (A variable can be static or dynamic.)
Operators

- In C, operators +, -, *, /, ++, --, ==, >, & , ^, ~, >>, ?:, (type)... have default semantic effects. All operators produce a value from their operands without modifying the operands. Some of these operators represent different actions depending on the types of their operands.
- In C++, operator is a special type of function. You can use operator overloading to define an operator.
- Precedence: determining the order in which an expression evaluates
  * Programmer can always change the default rules with parentheses.
- Auto increment operator ++, auto decrement operator –
  * Pre-increment and post-increment
  * Side effects
- Assignment: =
  * lvalue = rvalue (lvalue must be a distinct, named variable; must have physical space to store data)

Explicit Type Casts

- static_cast: for “well-behaved” and “reasonably well-behaved” casts, including things you might now do without a cast
  ```c++
  int i;
  long l;
  float f;
  l = static_cast<long>(i);
  f = static_cast<float>(i);
  ```
- const_cast: to cast const to non-const or volatile to non-volatile
  ```c++
  const int i = 0;
  int j = const_cast<int*>(&i);
  volatile int k = 0;
  int *u = const_cast<int*>(&k);
  ```
  Note:
  *j = 10; // this is OK
  cout « "i = " « i « endl; // output: i=0  j=10
  cout « "i = " « i « endl; // output: i=0  j=10
  ```

Explicit Type Casts (cont’d)

- reinterpret_cast: to cast to a completely different meaning. This is the most dangerous of all the casts.
  ```c++
  istream& read(char* pch, int nCount);
  int data[100]; ifstream infile("input.dat");
  infile.read(char*)data, sizeof(data));
  ```
- dynamic_cast: For type-safe downcasting, will be explained after inheritance
  ```c++
  cp = reinterpret_cast<unsigned char*>(&x);
  ```
  Output: 40  e2  1  0
Debugging

- **C Assert macro**
  ```
  #include <cassert>               // or #include <assert.h>
  using namespace std;
  int main() {
    int i = 100;
    someFunction();
    assert(i == 100); // fails
  }
  ```

- You can verify whatever design assumptions and code them into the program.
- You can remove the code from the executable by
  ```
  #define NDEBUG
  ```

  If you use Visual Studio and set active configuration as Release, `NDEBUG` is defined automatically.

Aliasing a type name with `typedef`

- **Simple rule:** `typedef original_type_name new_type_name;`
  ```
  typedef unsigned long ulong;
  ulong x; // equivalent to long x;
  ```

- **More general rule:** `typedef type definition of new_type_name;`
  ```
  typedef int IntAry[20];
  IntAry y[30]; // equivalent to int y[30][20];
  ```

  `typedef double (*(*FP)( ))[10];`
  `FP fp; // equivalent to double (*(*fp)())[10];`
  meaning: “a function pointer fp to a function that takes no argument and returns a pointer to a 10-element array of double”

  Equivalent and more self-explaining definitions:
  ```
  typedef double DoubleArray[10];
  typedef PtrDoubleArray *DoubleArray;
  typedef PtrDoubleArray (*FP)();
  ```

You can also define multiple new type names in one `typedef` statement

- `typedef struct`
  ```
  {
    int x;
    int y;
  } Point, *PtrPoint;
  ```

  `Point point; // equivalent to struct { int x; int y; } point;`

  `PtrPoint ptrPoint; // equivalent to struct { int x; int y; } *ptrPoint;`

Complex Data Type Definitions

- **Examples:**
  ```
  int *x;
  int *x[10];
  int (*x)[10];
  int (**x)[10];
  int *(*x)[10];
  void (*funcPtr)();
  void *funcPtr(); // definition of a function
  void (*)(int, void(*)(int))(int); // definition of a function
  void (*)((*fp1)(int))[*10];
  float (*)(int, int, float)(int);
  double (*)((*fp3)())[*10]);
  int (*)(*f4)()[10]);
  ```

  Using `typedef` can simplify these definitions
Complex Data Type Definitions

- Two simplest examples first:
  - `int *x[10];` // 10-element ARRAY of (PTR to integer)
  - `int (*x)[10];` // PTR to (10-element ARRAY of integers)

- `TYPE [n]` means “`n`-element ARRAY of `TYPE`”
- `TYPE *` means “PTR to `TYPE`”
- `[]` has higher precedence than `*`, `( )` can change the priority

- Some more
  - `int **x1;` // PTR to (PTR to `int`)
  - `int (**x2)[10];` // PTR to (PTR to (10-element ARRAY of `int`))
  - `int (**x3)[10];` // PTR to (PTR to (10-element ARRAY of (PTR to `int`)))

- Use typedef to simplify the definition
  - `typedef int *IPTR;`
  - `IPTR *x1;`
  - `typedef int IARY[10];`
  - `typedef IARY *PTRIARY;`
  - `PTRIARY *x2;`

- Function pointers
  - `void (*funcPtr)();` // PTR to a function that takes no argument and return void
  - `void (*(*fp)(int, void(*)(int)))(int);` // `fp` is a PTR to a function that takes two arguments, an int, (a function pointer that takes one int argument and returns void), and returns (a function pointer that takes one int argument and returns void)

  Equivalently,
  - `typedef void (*sig_t)(int);`
  - `sig_t (*fp)(int, sig_t);`

  - Real example:
  - `typedef void (*(*fp1)(int,int))(int);`

Complex Data Type Definitions

- Ex: PTR to a function that takes an int and returns a PTR to
  - (10-element array of (PTR to void))
  - `void *(*(*fp1)(int))[10];`

- Ex: PTR to a function that takes three arguments: int, int, float and returns
  - (PTR to a function that takes an int and returns float)
  - `float *(*(*fp2)(int, int, float))(int);`

- Ex: PTR to a function that takes no argument, returns
  - (PTR to (10-element ARRAY of
    - (PTR to a function that takes no argument and returns double)))
  - `double *(*(*fp3))()[10]);`

- Ex: function that takes no argument, returns
  - (PTR to (10-element ARRAY of
    - (PTR to function that takes no argument and returns int)))
  - `int *(*(*f4))()[10]);`