C^{++} Basic Guide. Programming Tips and Tricks

Creating C^{++}-Programs

Where to Code

The CCC provides a number of facilities for programming, including the CCC-lab (2nd floor FL-building) and the zoo- and garden-labs (FL-basement). Rumour has it that the latter two will be closed during most of June, so the CCC-lab is the remaining option. You can write programs from any computer with a modem if you telnet to WPI; however you will not be able to use graphics-mode editors (emacs).

Alternative Compilers

Your programs must work on a CCC-Machine (bert, ernie, reno, grover, wpi) to receive full credit. That means that you need to obtain an account on the CCC-Unix-system in order to participate in this class. However, we will be using mostly relatively simple C^{++}-features, so you should be able to develop your software on any other system running C^{++}. UNIX/LINUX comes with built-in compilers; for Windows there is Microsoft Visual C^{++} available from the WPI library[1] or the free command-line version of Borland-C^{++}-5.5, available from Inprise (`www.inprise.com`) after registration. If you use another machine to develop, remember to double-check that your program runs on a CCC-machine, since different C^{++}-compilers have subtle differences.

Writing Source Code

The best editor for coding, and unfortunately the hardest to use, is **emacs**. Emacs is available on all UNIXes and exists also in a Windows-version which can be downloaded from most shareware-archives. In order to run the graphical version of emacs on UNIX, you have to have X running (or connect from a Windows-machine through PCXWare or a similar product). Important commands are: `CTRL-X CTRL-F` to load a file, `CTRL-X CTRL-S` to save, `CTRL-X CTRL-C` to leave emacs, `CTRL-_` to undo. An emacs-introduction is available from the CCC (`http://www.wpi.edu/Academics/CCC/Help/Unix/emacs.html`). If you put the following three lines in a file `.emacs` in your home directory, you will get nice colorful syntax highlighting:

```
(global-font-lock-mode t)
(set-background-color "white")
(set-foreground-color "black")
```

If you do not want to learn emacs, you can use **pico** on UNIX or the editor of your choice on Windows. Pico is a relatively primitive editor and does not provide syntax highlighting or automatic indentation as emacs does, but is more intuitive to use in a UNIX-text-mode terminal window.

[1] paid for through the WPI Technology Fee

Compiling and Running your Programs

Compilation is very different on different compilers. Graphical environments (like Visual C^{++} or Borland C^{++}-Builder) normally provide sufficient online help and interactive customization so they will not be covered in this document. For command-line compilers, we recommend using **makefiles**. For most homework assignments, a makefile for the CCC machines will be provided. In order to generate your program from the source code and include libraries, simply copy the makefile into your working directory and type `make`. `make clean` normally cleans up all generated files which is useful once you have finished a project. Linking is done automatically with most current compilers, so we will not elaborate on that.

Debugging

For command-line driven development, the most efficient debugging methods are black-box-testing, printing variables, and step-by-step debugging.

In **black-box testing**, the program is tested by giving certain input and comparing the output with what is expected. Many errors can be found in this way.

Printing variables helps with more subtle errors. Include simple output statements to the standard error stream (`cerr << "loop counter:  " << i;`) at strategic positions in your code and compare the variable values with what you expect. Sometimes, it helps to just print an asterisk or a number in several places in the source code to see the control flow of the program. Conditional printouts (`if (i>10) cerr << "i>10";`) can also be useful. This method often helps finding errors in border cases (at the beginning or end of loops), or at rarely used code segments.

Step-by-step-debugging is the most powerful tool, but also the hardest to use. On the CCC-machines, the software for step-by-step debugging is the GNU-Debugger `gdb`. Start gdb with your executable file as a parameter and type `help` for a command list. The usual way of debugging is to set a breakpoint shortly before the suspected error in the source file and then step through the file with `next` or `step` commands.

Coding Style for CS2005

Source Code Formatting and Names

In professional software development, rigid style guides are normally provided that programmers must adhere to. However, there is no generally accepted good style—readability is the most important requirement, and several different styles are used. The textbook's guide on page 744 will do. For variable and function names, I personally prefer capitalization of each word except the first one (`printEmployeeSalary()`) rather than the style used in the textbook (`print_employee_salary()`). As long as your are consistent, either is fine. One rule is important: *variable names must be meaningful words*, single letter variable names are (almost) never acceptable!

Documentation

As suggested in the textbook, a homework requirement (necessary for full credit) is to provide pre- and post-condition descriptions for each function or method except `main`. The precondition description must suffice for a programmer who has no access to the function's source code to determine which input is expected and under which conditions the function will work as described. Likewise, the postcondition description must show clearly what a function does and what the returned data means. Always assume that someone else uses one of your functions and has no access to your source code.[2]

Also, the purpose of global variables must be explained in a comment. Furthermore, any function implementation that is not trivial should be commented (any non-obvious algorithm or program statement deserves an explanation). If you would look at your code again in six months, you should be able to understand the program logic without difficulties.

Ease of use

Your software must be easy to use. That means that after starting the software, it must be clear to see what input is expected from a user or what the meaning of any output is. Be generous with text output that explains what your program is doing. A program whose usage is not clear without looking at the source code will not receive full credit.

Robustness

Your program should tolerate unexpected input to a certain degree. For example, inputting a string when a number is expected should lead to a repeated user input request in interactive use and to an error message when reading from a file. At a minimum, the program should end with an appropriate error message (using the `assert`-function). Unexpected conditions should not lead to endless loops or simple continuation of the program with invalid or wrong data. Be careful with string-input as the simple `>>`-operator provides no mechanism to catch input strings that exceed the available space (i.e., you can assign a 10-character string to a `char[5]`-variable with undefined results).

[2]For private member functions of classes, this requirement is less critical as the source code will always be available. However, missing documentation for public methods will lead to reduced credit.

C++ Primer—CS 1005 in a Nutshell

Data Types

```
int    i = 4;   // (32-bit) signed integer
double d = 2.0; // (64-bit) real number
char   c = 'x'; // '\n' is a character, too
char   c[30] = "This could be a string"; // character array = string
int    a[10];   // an array of 10 integers (indices from 0 to 9)
struct Person {
   char lastName[30];
   char firstName[20];
   int  yearOfBirth;
};               // a structure, containing several primitive data types
```

- The last character in a string is `'\0'`, so a `char c[30]` fits strings of length up to 29.
- `structs` can hold sets of element data and can then be used just like the pre-defined data types. Declaring a variable of `struct`-type reserves memory for all elements. Access to an element is done with the `.`-operator. Example (with the `struct` above):

```
Person onePerson;
onePerson.lastName = "Doe";
onePerson.firstName = "John";
onePerson.yearOfBirth = 1972;
```

- Don't forget the semicolon after the `struct` definition.

Control Structures

<u>`if`</u>

```
if ( a > b ) { // other operators: >= < <= != ==
   return a;
}
else {
   return b;
}
```

- don't forget the `()` around the if-condition
- most common error: using = instead of ==. *Tip:* if possible, use `3 == a` instead of `a == 3`, that way the compiler will find the error.

for

```
for (int i = 0; i < 10; i++) {
        cout i << " " << i*i;
}
```

- The three elements of a `for`-loop: “starting at”; “as long as”; “do this”.
- The condition (`i < 10`) is tested *before* the variable is changed (`i++`).
- The variable is changed *after* all the code inside the `{ }` has been executed.
- Any of the three elements of the `for`-statement can be empty. `for(;;)` means an endless loop.

while

```
while (ch != 'Q') {
   process();
   cin >> ch;
}
```

`while` is more intuitive than `for` in some cases but you have to initialize variables separately. Compare:

```
for (double x = 100; x > 2; x = sqrt(x))
    cout << x << endl;

double x = 100;
while (x > 2) {
    cout << x << endl;
    x = sqrt(x);
}
```

- `continue` statements are not-so-great programming style but may be convenient in some cases
- `break` statements can often be avoided by using the `do {...} while(condition)` construct instead.

switch

```
char ch;
cin >> ch;
switch (ch) {
    case 'l':
       cout << "List of Records";
```

```
      printListOfRecords();
      break;
   case 'h':
   case '?':
      printHelpScreen();
      break;
   default:
      cout << "Invalid input!";
}
```

- Switch variables must be numeric or `char`, *not* strings.
- As you can have more than one statement in each `case`, you need a `break` after each case, otherwise the program execution "falls through" to the next case (in the example, if "h" is pressed, the same code gets executed as for "?").
- The `default` gets executed whenever none of the `cases` applies.

Functions

In most non-trivial programs, functions need to be declared before their implementation is given ("prototypes"). Normally, this is done in the "header"-file (ending on `.h`) belonging to a C++-file. Variable names can be omitted in the prototype (and types in the implementation) if that does not lead to confusion.

maximum.h

```
// Prototype:
int max(int , int );
```

maximum.cxx

```
#include "maximum.h"

// Implementation:
int max (int a, int b) {
   int max;
   if (a>b) max = a;
   else     max = b;
   return max;
}
```

- Parameter types can be references (marked by `&`) [3].
- Passing (and returning results) by value (`int max(int a, int b)`) makes a copy of the data (i.e., in the example, `a` and `b` would be copied at the beginning of the function, then the local variable `max` is copied again to provide a return value from the function).

[3] Return types can be references, too, but this makes sense only in a few cases.

- Passing by reference (`int max(int &a, int &b)`) does not copy data but may change the original (passed) data in the calling function.
- Declaring reference parameters constant (`Item max(const Item &a, const Item &b)`) makes data in the calling function non-changeable.

Input/Output

Simple I/O:

```
char no[7] = "number";
cout << 3 << " is a " << no;

int  i;
char ch;
char str[20];

cin >> i >> ch >> str;
```

File I/O:

```
#include <iostream.h>
#include <fstream.h>

char (database[80])[30];

ifstream infile;
infile.open("input.txt");
int i = 0 ;
while (infile.peek() != EOF) {
    infile >> database[i];
    i++;
}
infile.close();

ofstream outfile;
outfile.open("outfile.txt");
int size = i; // copy variable i from above into something meaningful
for (i = 0; i < size; i++)
  outfile << database[i] << endl;
outfile.close();
```

- never forget to close a file after you are done reading/writing
- other useful functions of I/O-stream are in the book in Appendix F
- reading strings with >> does not work if the string has whitespaces. In that case, use

`infile.getline(charVariable, maxLength).`

This will also prevent buffer overflow.

Pointers

This section is meant for self-study. It gives correct code examples. Find out what they're doing and why.

Simple Pointers

```
int *pointer;
int i = 10;

pointer = &i; // address of i
cout << i << endl;
cout << *pointer << endl; // "dereferencing"

cout << pointer << endl; //?? what happens?
```

Assigning Pointers

```
int *pointer1, *pointer2, *pointer3;
int i = 10, j = 20;

pointer1 = &i; // address of i
pointer2 = pointer1;
cout << *pointer1 << " " << *pointer2 << endl;

pointer3 = &j;
*pointer2 = *pointer3;
cout << *pointer1 << endl; // and now?
```

Pointers as Parameters

```
int number = 4;

void sqr(int value) {
    value *= value;
}

void sqr_ptr(int* pointer) {
    *pointer = (*pointer) * (*pointer);
}

int main() {
```

```
    sqr(number);
    cout << number << endl;
    sqr_ptr(&number);
    cout << number << endl;
}
```

Pointers to a `struct`

```
struct testStruct {
    char lastName[30];
    char firstName[20];
};

testStruct person = {"Doe","John"};
testStruct *personPtr = &person;

cout << person.lastName << endl;
cout << (*personPtr).firstName << endl;
cout << personPtr->firstName << endl; // the last two lines are equivalent
```

Pointer Arithmetic

```
int array[4] = { 1,4,9,16};
int* arrayp;

arrayp = array;
cout << *arrayp    << endl;
cout << arrayp[1] << endl;
arrayp += 2;
cout << *arrayp  << endl;
```