# Computer Science C++ Placement Exam

The Computer Science Department now has available a placement exam for students who wish to demonstrate their competence in the material covered in the course CSCI C101 Computer Programming I. Students who pass this exam at a high level may enroll in CSCI C201 Computer Programming II without first completing C101. Such students may also have the course CSCI C101 placed on their transcript for 4 credit hours, provided they pay the required fees.

## Topics to Be Covered on the C++ Exam

The placement exam is designed to test carefully whether a student has mastery of the topics covered in C101 so that the student can do the work in C201 without burdening the instructor with inappropriate questions. Mastery of all the following topics is required:

**1.** Use of the IDE (Integrated Development Environment) of Microsoft Visual Studio .NET. Students must demonstrate the ability to edit source code efficiently, to save a file, to compile source code into an executable program, and to run a program, all from the IDE.

**2.** Systematic program development.
Students must demonstrate ability to organize a program into appropriate functions and to write pseudo-code for the algorithms used by the functions.

**3.** The elementary features of the C++ programming language.
A complete checklist of all the required features is given further along in this document.

**4.** Good programming style. Students must demonstrate that they can

a. use a clear and consistent indenting method to produce a logically formatted program;
b. choose appropriate identifiers for variables;
c. write complete, understandable documentation for a program and each of its functions, and also provide helpful line-by-line comments where these are appropriate.

## Format of the C++ Exam

The exam will be in two parts. The first part will last one hour and will require the student to answer a number of written questions about C++ and program development. The second part will be conducted in a computer laboratory, where the student will be given four hours in which to write one or more complete C++ programs and/or fill in the missing parts of some incomplete programs.

## A Warning about the Programming Part of the C++ Exam

A student program that correctly exhibits the behavior prescribed in a programming problem will not be given a high grade unless the program is also well written. Simple correctness of behavior is not sufficient for a good grade. Thus, to give some examples, a program that's poorly formatted, or poorly documented, or that makes unnecessary tests of data, or that uses a poor algorithm will be given a low grade.

## A Checklist of C++ Features That Must Be Mastered

**1.** Structure of C++ programs:
   a. placement of "`#include`" directives and "`using..`" statements";
   b. placement of the `main` function and other function declarations (prototypes) and definitions (code).

**2.** Basic syntax rules for
   a. identifiers;
   b. declarations and initializations of variables;
   c. control structures ("`if`", "`if...else...`", "`switch`", "`while`", "`do...while`", "`for`").

**3.** Fundamental data types:
   a. `int, char, float, long, double;` understanding automatic type conversion in arithmetic expressions and assignments;
   b. enumerated types.

**4.** Common operators and the value of an operator expression:
   a. assignment operators: `= , += , -= , *= , /= , %= ;`
   b. arithmetic operators: `+ , ++ , - , -- , * , / , % ;`
   c. relational operators: `< , <= , > , >= , == , != ;`
   d. logical operators: `&& , || , ! ;`
   e. the conditional operator (ternary): `? : ;`
   f. elementary precedence rules among these operators.

**5.** Elementary I/O (input/output) in C++ using the following operators and functions defined in the `iostream` header file:
   a. `cout, cin, <<  and  >> ;`
   b. `endl;`
   c. `cin.get(), cin.getline();`

**6.** Conditional statements using `if...` or `if...else...;` nested conditional statements.

**7.** Loops using `while...` or `do...while...` or `for....`

**8.** Functions and parameter passing. Difference between a value parameter and a reference parameter. Function declarations (prototypes) and function definitions (code).

**9.** These functions from the `cmath` header file: `sqrt(); pow(); fabs();`

**10.** Elementary scope rules.

**11.** Arrays. The use of the qualifier "`const`" while passing an array to a function;

**Advice to C Programmers**

Most of the features of C++ listed in the "Checklist" above will be familiar to you from the C language. Here are the four principal differences between C and C++ that you will be expected to have mastered. (There are many other differences, and those will be covered in C201.)

1. In addition to C style commenting using "`/*`" and "`*/`", C++ has single-line comments starting with "`//`".
2. C++ allows declarations of constants that can, for example, be used to dimension arrays. Good C++ programmers avoid using the `#define` directive, since it does not allow for type checking of the defined quantities.
3. C++ allows passing variables to functions by reference, which avoids having to deal with pointers while passing elementary data objects. You will be required to avoid pointers and to use reference parameters whenever reference parameters are appropriate.
4. C++ has a simpler system of I/O (input/output) that uses the type-safe operators << and >> as well as several functions in the header file `iostream`.

**Things to Avoid in the C++ Programming Exam**

1. Do not use "`break;`" to exit from a loop. Similarly, do not use any "`goto`" statements.
2. Do not use the C language I/O functions `printf()` and `scanf()`. Use only the functions and operators defined in the `iostream` header file and (if you like) the `iomanip` header file.
3. Do not use any "`#define`" directives in any program.
4. Do not use a global variable in any program.

## Some Formatting Requirements for the C++ Programming Exam

**1.** Use all upper case letters for identifiers that denote constants. This is standard practice in C and C++ programming.

**2.** Use a consistent style for identifiers for variables. You may choose any one of the following styles, but then use only that one style you have chosen:

```
location_of_2nd_target     // Digits and lower case letters are used,
                           // with words separated by the underscore
                           // character.
LocationOf2ndTarget        // Each word of the identifier starts with
                           // an upper case letter or a digit.  Words
                           // are not separated.
Location_Of_2nd_Target     // Each word of the identifier starts with
                           // an upper case letter or a digit, and
                           // words are separated.
Location_of_2nd_target     // First word starts with upper case
                           // letter.  Lower case is used elsewhere;
                           // words are separated.
locationOf2ndTarget        // Each word of the identifier except the
                           // first starts with a digit or upper case
                           // letter.
```

**3.** Use exactly two spaces for each level of indenting.

```
 Correct:    while (i < LIMIT)
               if (a[i] > a[best])
                 best = i;
 Incorrect:       while (i < LIMIT)
              if (a[i] > a[best])
               best = i;
 Incorrect:       while (i < LIMIT)
                if (a[i] > a[best])
                   best = i;
```

**4.** Use any one of the following styles for placement of the braces in a compound statement in a C++ program. Whichever style you pick, use that style consistently throughout the program.

```
Style 1:   if (x < 0)
           {
             cout << "The value is negative." << endl;
             cout << "Enter another value: ";
             cin  >> x;
           }
Style 2:   if (x < 0) {
             cout << "The value is negative." << endl;
             cout << "Enter another value: ";
             cin  >> x;
           }
```

```
Style 3:   if (x < 0)
             {
               cout << "The value is negative." << endl;
               cout << "Enter another value: ";
               cin  >> x;
             }
```

**5.** Place a blank on each side of every binary operator symbol.

```
Correct:    while (i < LIMIT)
              if (a[i] > a[best])
                best = i;
Incorrect:      while (i<LIMIT)
              if (a[i]>a[best])
                best=i;
```

The following three pages reproduce a "handout" distributed to students in Dr. Knight's programming classes at IUSB. They explain in detail the C++ programming style that he prefers. The pages contain a number of valuable suggestions that will help you write C++ code that is easily understood. The handout is adapted from an earlier handout prepared by Dr. Russo. This is provided for your information and for the benefit of upcoming classes you might take, but will not be strictly enforced for the placement test.

**Knight's C++ Programming Style Guidelines**

Here are some requirements I have concerning your C++ coding style.

**(0)** Identifiers for constants should be all upper case, with underscore characters. Examples:

```
const double PI           = 3.14159;
const int    STRING_LIMIT = 80;
```

**(1)** Identifiers for variables and functions should be formed (consistently throughout any program) according to any one of the following three rules:

(a) They can be all lower case, with underscore characters serving as "glue". Examples:

```
rate_of_change    target_found    car_count .
```

This is the classical style used by almost all C and early C++ programmers and many professionals today. I find it to be the most easily readable.

(b) They can be start with an lower case letter followed by more lower case letters, except that where a new word is "glued on", use an upper case letter. Examples:

```
rateOfChange    targetFound    carCount .
```

This style is also used by many professionals today. It is favored by your text.

(c) Just like rule (b), but start with an upper case letter. Examples:

```
RateOfChange    TargetFound    CarCount .
```

This is old Pascal style. Many C++ instructors continue to use it at IUSB and elsewhere.

**(2)** Do not EVER use lower case "el" ( `l` ) or upper case "oh" ( `O` ) by themselves as identifiers for variables. They look too much like "one" ( `1` ) and "zero" ( `0` ) respectively.

**(3)** Do not EVER use any global variables in any program you submit. Global constants are another matter, however. They are preferred if the same constant is used in more than one function.

**(4)** When you declare a variable in a function, do not initialize it unless you plan to use that value immediately. For example, there is no reason to initialize a variable to zero if you never use that zero value and later overwrite the zero with another value. Do not initialize a variable and then first use that initial value many lines farther along. Instead, place the assignment of a first value to the variable as close as possible to the point at which that value is used. A person reading your code should not have to hunt back through the program to find where you initialized a variable.

**(5)** Indent to indicate subsidiary material, but not for any other reason. Each new level of indenting should be 2 spaces farther to the right than the prior level. Do not put the body of an if statement on the same line as if .

ACCEPTABLE:
```
  if ( total > limit )
    cout << "total exceeds the limit" << endl;
```
UNACCEPTABLE:
```
  if ( total > limit )
  cout << "total exceeds the limit" << endl;
```

if ( total > limit ) cout << "total exceeds the limit" << endl;

**(6)** An if-else statement should be formatted as follows:
```
         if ( some condition )
      do_something;
   else
      do_something_else;
```

UNACCEPTABLE:

```
   if ( p != NULL )
      p = p->next;
     else return;        // The  "return;"  must be placed on a separate
line.
```

Nested if-else statements should be formatted as follows:
```
  if ( income < 0.0 )
    cout << "Lost money." << endl;
  else if ( income < 14000.00 )
    cout << "Below the poverty line." << endl;
  else if ( income < 70000.00 )
    cout << "Comfortable." << endl;
      else
    cout << "Well-to-do." << endl;
```

The following version of the code above is considered to be amateurish.
```
  if ( income < 0.0 )
    cout << "Lost money." << endl;
  else
    if ( income < 14000.00 )
      cout << "Below the poverty line." << endl;
    else
      if ( income < 70000.00 )
        cout << "Comfortable." << endl;
          else
        cout << "Well-to-do." << endl;
```

**(7)** Don't let comments spoil the logical structure of the code on the page. The best place for comments is out to the right of the actual code. When you feel you must write inter-linear comments, indent them the same amount as the code they describe.

ACCEPTABLE:

```
if (balance < 0.0)
  // Print a message denying service.
  cout << "You do not have enough money in your account.\n";
```

BETTER:

```
if (balance < 0.0)   // then print a message denying service.
  cout << "You do not have enough money in your account.\n";
```

UNACCEPTABLE:

```
      if (balance < 0.0)
 // Print a message denying service.
        cout << "You do not have enough money in your account.\n";
```

**(8)** The opening and closing braces for a compound statement should line up vertically.

MY PREFERENCE (this is the style used by the automatic indenting feature of the Visual Studio .net editor):

```
if (total < 0)
{
  cout << "Total is negative." << endl;
  return;
}
```

ACCEPTABLE ALTERNATIVES:

```
if (total < 0)    // Lot's of programmers use this style.
  {
    cout << "Total is negative." << endl;
    return;
  }

if (total < 0)    // I tolerate this one but don't care for it.
  {
  cout << "Total is negative." << endl;
  return;
  }
```

UNACCEPTABLE TO ME (even though it is used by many texts and professionals)

```
if (total < 0) {
  cout<< "Total is negative." << endl;
  return;
}
```

**(9)** The body of a function should be indented away from the left margin as in this example:

```
int main()
{
  char ch;

  cout << "Type a character and press Enter: ";
  cin  >> ch;
  .....
  return 0;
}
```

**(10)** Use blank spaces to separate all adjacent symbols on a line.

ACCEPTABLE:

```
  for ( index = 1; index <= limit; ++index )
  {
    ....
    ....
  }
```

UNACCEPTABLE:

```
  for(index=1;index<=limit;++index)
  {
    ....
    ....
  }
```

You can ignore this rule with the arithmetic operators if the operands are very short. For example, `n = 3/k;` is acceptable, but this is not: `n = strike_count/pitches` .

**(11)** Make liberal use of blank lines to group related statements within a function. Always use a blank line to separate the local variable declarations in a function from the statements that make up the rest of the function.

**(12)** Every function except main() must have its own documentation section immediately preceding the code for the function. The parameters should be described, and all assumptions that are being made about the parameters should be explicitly stated. The action(s) of the function should be described, but no mention should be made of other function(s) that will be calling this function or how it "fits into" the program as a whole. The documentation for a function should be preceded by a "banner" line that names the function. See the documentation for functions in the programs I distribute in class.

**Selecting Helpful Identifiers**

Well chosen identifiers can make reading a program almost like reading ordinary prose. Wherever reasonable, use identifiers that have mnemonic value, i.e., that help the reader understand the intended meaning of the variable, constant, or function. Without going overboard, be specific rather than vague.

| Poor Identifier Names | Good Identifier Names |
|---|---|
| `h` | `hourly_pay_rate` |
| `table` | `conversion_table` |
| `rate1, rate2` | `low_rate, high_rate` |

Avoid nonstandard abbreviations that will not be clear to others.

| Poor Identifier Names | Good Identifier Names |
|---|---|
| `sec_per_trk` | `sectors_per_track` |
| `dlm` | `delimiter` |
| `fr_tbl` | `frequency_table` |

Often when you are choosing a name for a subscript variable or a loop control variable, no identifier with mnemonic value will suggest itself. In this case a single letter is an acceptable identifier.
Be especially careful when choosing names for functions. The readability of your main program will be greatly enhanced if these names are descriptive. Since most functions are called only once or twice, you impose no typing burden on yourself by choosing fairly long names.

Generally speaking, a function that returns a single value should be named with a noun or a noun clause that describes the object being returned. An exception to this rule is the case where an integer is being returned as a boolean flag (i.e., a value representing "true" or "false"). In this case, a verb phrase may make a more appropriate name.

| Function Prototype | The function name is a |
|---|---|
| `float max_pay_rate (float pay_rate[]);` | noun clause |
| `char first_letter (char name[]);` | noun clause |
| `bool is_empty (char command_queue[]);` | verb phrase |
| `bool are_relatively_prime (int p, int q);` | verb phrase |
| `bool contains_punctuation (char line[]);` | verb phrase |

By contrast, a function that either returns no value or else returns several values by means of reference variables should generally be given a name that suggests a command.

```
void print_table (float conversion_table[], int limit);
void find_extreme_values (float list[], float & max, float & min);
```