C++ tips and tricks

Bruce Merry

IOI Training Dec 2013

Outline

Outline

Assertions

You can check that something is true using `assert`:

```
#include <cassert>
int main()
{
    assert(1 == 2);
}
```

Output:

```
test_assert: test_assert.cpp:4: int main():
    Assertion '1 == 2' failed.
```

Disabling Assertions

To disable assertions, add

```
#define NDEBUG
```

as the first line of your source.

Caution


```
#define NDEBUG
#include <cassert>
#include <iostream>
using namespace std;
bool foo() {
    cout << "In foo\n";
    return true;
}

int main() {
    assert(foo());
}
```

Caution


```
#define NDEBUG
#include <cassert>
#include <iostream>
using namespace std;
bool foo() {
    cout << "In foo\n";
    return true;
}

int main() {
    assert(foo());
}
```

When assertions are disabled, the expression is not evaluated.

Outline

String-To-Integer Conversions

Use `istringstream` to treat a string as an input stream:

```
#include <sstream>
int x;
istringstream stream("123");
stream >> x;
// Now x == 123
```

String-To-Integer Conversions

You can reduce typing by using a C function instead:

```
#include <cstdlib>
string xstr = "123";
string ystr = "12345678912345678";
int x = atoi(xstr.c_str());
long long y = atoll(ystr.c_str());
```

String-To-Integer Conversions

C++11 has a more convenient wrapper:

```
#include <string>
string xstr = "123";
string ystr = "12345678912345678";
int x = stoi(xstr);
long long y = stoll(ystr);
```

Integer-To-String Conversions

The general solution is `ostringstream`:

```
#include <sstream>
ostringstream o;
o << 123;
string s = o.str();
// s == "123"
```

Integer-To-String Conversions

C++11 again has a convenience wrapper

```
#include <string>
string s = to_string(123);
```

Outline

Introduction

In Java and Python, all objects are references:

- Passing to a function is cheap: just another reference
- Callee function can modify the object
- Every object must be explicitly created (e.g., with `new`)

C++ Default

By default, C++ objects are values:

C++ Default

By default, C++ objects are values:

```
void foo(vector<string> grid)
{
    // foo operates on a *copy* of grid
}
```

C++ Default

By default, C++ objects are values:

```
void foo(vector<string> grid)
{
    // foo operates on a *copy* of grid
}

string mystrings[4];
// array contains 4 empty strings
```

Reference arguments

To make a parameter a reference, prefix it with `&`:

```
void foo(vector<string> &grid)
{
    // foo now operates on the original grid
}
```

Reference arguments

To make a parameter a reference, prefix it with `&`:

```
void foo(vector<string> &grid)
{
    // foo now operates on the original grid
}
```

Can also qualify references as `const`:

```
void foo(const vector<string> &grid)
{
    // foo is prevented from modifying grid
}
```

Reference Variables

Variables can be references, but they cannot be changed:

```
vector<string> strings(5);
string &first = strings[0];
string &second = strings[1];
string &something; // error
first += "hello"; // appends to strings[0]
// copy one *string* to another:
second = first;
```

Pointers

Pointers are similar to references

- Can be changed to point at other things
- Can be null pointers
- Syntax is more roundabout
- Avoid them for now

Outline

Typedefs

Can define shorthand for other types:

```
typedef long long ll;
typedef vector<vector<ll> > vvll;
...
// declare a vector<vector<long long> >:
vvll myarray;
```

Outline

Improving Read Performance

Add `ios::sync_with_stdio(false)` to the start of your program to improve `cin` performance.

Table : Input performance (time to read 10^7 integers)

Method	Time (s)
`cin`	2.70
`cin` with tweak	0.78
`scanf`	0.84

Improving Read Performance

Add `ios::sync_with_stdio(false)` to the start of your program to improve `cin` performance.

Table : Input performance (time to read 10^7 integers)

Method	**Time (s)**
`cin`	2.70
`cin` with tweak	0.78
`scanf`	0.84

Side effect: do not mix `cin` and `scanf`

Improving Write Performance

What is the difference between these two lines?

```
cout << 123 << endl;
cout << 123 << '\n';
```

Improving Write Performance

What is the difference between these two lines?

```
cout << 123 << endl;
cout << 123 << '\n';
```

Using `endl` flushes the output.

Improving Write Performance

What is the difference between these two lines?

```
cout << 123 << endl;
cout << 123 << '\n';
```

Using `endl` flushes the output.

Table : Output performance (time to write 10^7 integers)

Method	Time (s)
`endl`	2.34
`'\n'`	0.75
`printf`	0.80

Outline

Warnings

`-Wall` Provide lots of helpful warnings

`-W` Provide even more warnings, some useless

Optimisation

Use `-O2` to optimize your code

- Speedup varies a lot, depending on code

Optimisation

Use `-O2` to optimize your code

- Speedup varies a lot, depending on code
- Interferes with debugging tools

Optimisation

Use `-O2` to optimize your code

- Speedup varies a lot, depending on code
- Interferes with debugging tools
- Undefined behaviour can change

Optimisation

Use `-O2` to optimize your code

- Speedup varies a lot, depending on code
- Interferes with debugging tools
- Undefined behaviour can change
- Some warnings only work with optimisation

Optimisation

Use `-O2` to optimize your code

- Speedup varies a lot, depending on code
- Interferes with debugging tools
- Undefined behaviour can change
- Some warnings only work with optimisation
- Can also do `-O3`, but has diminishing returns

Outline

Including The Standard Libraries

This will pull in all the standard library headers

```
#include <bits/stdc++.h>
```

It does make compilation quite slow.

Outline

Uninitialized Data


```
int x;
int y[3];
vector<int> z(4);
cout << x << ' ' << y[1] << ' ' << z[2];
```

Which values are well-defined?

Uninitialized Data

The following are generally safe:

- Classes with a constructor, if the constructor explicitly initialises all fields.
- STL containers like `vector` (even for primitive types)

Primitive types are undefined when:

- Declared directly
- Declared in an array
- Declared in a struct/class and not set by constructor

Out-of-range Array Access


```
int x[3] = {1, 2, 3};
x[3] = 4;
```

Anything can happen here!

References to Local Variables

Do not try to *return* containers by reference:

```
vector<int> &foo(int n)
{
    vector<int> ans;
    for (int i = 0; i < n; i++)
        ans.push_back(i);
    return ans;
}
```

References to Local Variables

Do not try to *return* containers by reference:

```
vector<int> &foo(int n)
{
    vector<int> ans;
    for (int i = 0; i < n; i++)
        ans.push_back(i);
    return ans;
}
```

Return by value

References to Local Variables

Do not try to *return* containers by reference:

```
vector<int> &foo(int n)
{
    vector<int> ans;
    for (int i = 0; i < n; i++)
        ans.push_back(i);
    return ans;
}
```

Return by value — GCC will optimise it

Outline

Mod on Negative Values

- `5 % 3 == 2`
- `-5 % 3 == -2` (unlike Python)

When a problem asks for an answer modulo *M*:

```
ans %= M;
if (ans < 0)
    ans += M;
```

Unsigned Is Evil

What is wrong with this code?

```
// One pass of bubblesort
for (int i = 0; i < arr.size() - 1; i++)
    if (arr[i] > arr[i + 1])
        swap(arr[i], arr[i + 1]);
```

Unsigned Is Evil

What is wrong with this code?

```
// One pass of bubblesort
for (int i = 0; i < arr.size() - 1; i++)
    if (arr[i] > arr[i + 1])
        swap(arr[i], arr[i + 1]);
```

If `arr` is empty, then `arr.size() - 1` wraps around.

Stack Overflow

- Function parameters and local variables kept on a stack
- Stack size limits possible recursion depth
- Linux defaults to an 8 MiB stack!

So be careful with more than 100 000 levels of recursion.