COMP2004 Programming Practice 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

Using &

- & has 2 different meanings depending on where it's used
- In front of an object
  - Takes the address
  - `int* p = &i;`
- In a function parameter definition
  - Pass by reference
  - `int sum(std::vector<int> &v)`

Code style

- Indenting
- Whitespace
- Identifiers
- Structure
- Comments
- Bracketing

- In all cases aim to make your code
  - Easily readable
  - Consistent

Indenting

- By far the most important
- Choose a certain number of spaces and stick with it
- Common ones are 8, 4 and 2
- Indent code inside control structures
  - functions, if(), while(), for(), etc

Bad Indenting 1

```
double a_function(int a) {
   int b = 0;
      double c;
while(b < 100) {
      if (a == b) {
c = 34.2; }
    else
{  c = 12.8;
}
}
}
```

Bad Indenting 2

```
double a_function(int a) {
int b = 0;
double c;
while(b < 100) {
if (a == b) {
c = 34.2; }
else
{c = 12.8;
}
}
}
```

Good Indenting

```
double a_function(int a) {
        int b = 0;
        double c;
        while(b < 100) {
                if (a == b) {
                        c = 34.2;
                } else {
                        c = 12.8;
                }
        }
}
```

Whitespace

- Groups related code together
- Vertical
  - Blank lines between functions, code paragraphs
- Horizontal
  - Spaces around operators, brackets, etc

Bad Vertical Whitespace

```
unsigned int numDigits(unsigned long x) {
        return static_cast<unsigned int>(log10(x)) + 1;
}
void getDigits(std::vector<unsigned int> &digits, unsigned long number) {
        unsigned int i, n;
        n = numDigits(number);
        digits.resize(n);
        for (i = n - 1; i > 0; i--) {
                digits[i] = number % 10;
                number = number / 10;



        }
        digits[i] = number;
}
int main() {
        unsigned int number;
        while (std::cin >> number) {
                std::vector<unsigned int> digits;
                getDigits(digits, number);
                std::cout << digits[0];
                for (int i = 1; i < digits.size(); i++)
                        std::cout << ", " << digits[i];
                std::cout << std::endl;
        }
}
```

Good Vertical Whitespace

```
unsigned int numDigits(unsigned long x) {
        return static_cast<unsigned int>(log10(x)) + 1;
}


void getDigits(std::vector<unsigned int> &digits, unsigned long number) {
        unsigned int i, n;

        n = numDigits(number);
        digits.resize(n);
        for (i = n - 1; i > 0; i--) {
                digits[i] = number % 10;
                number = number / 10;
        }
        digits[i] = number;
}


int main() {
        unsigned int number;

        while (std::cin >> number) {
                std::vector<unsigned int> digits;
                getDigits(digits, number);

                std::cout << digits[0];
                for (int i = 1; i < digits.size(); i++)
                        std::cout << ", " << digits[i];
                std::cout << std::endl;
        }
}
```

Horizontal Whitespace

- if((d>=1)&&(d<=maxDays))
- if ((d >= 1) && (d <= maxDays))

- for(i=s.size()-1;i>0;--i)
- for (i=s.size()-1; i>0; --i)
- for (i = s.size() - 1 ; i > 0 ; --i)

Identifiers

- Names of variables, functions, enums, etc
- Descriptive and concise
- Nouns for variables, verbs for functions
- Singular for variables, plurals for collections
- Consistent scheme for multiple-word identifiers
  - `multipleWordIdentifier`
  - `multiple_word_identifier`

Bad Identifiers

- Anything one letter long (except i as a loop iterator)
  - eg: a, b, c, d, ...
- Identifiers which vary by only 1 character
  - eg: item0, item1, item2, ...
- If these items are related, consider an array/vector instead
  - eg: item[0], item[1], item[2], ...

Bad Identifiers

- Unnecessary abbreviations and spelling mistakes
  - eg: nmItms, piont
  - better: numItems, point
- Unnecessarily long/verbose
  - eg: number_of_elements_in_array
  - better: num_elems

Structure

- High cohesion
  - Each function performs exactly one task
- Low coupling
  - Each function has minimal reliance on other functions
- Low cohesion and high coupling make code hard to read and maintain
- eg. solution for Week 1 Wed Q4

Comments

- Describe the code's purpose / meaning
- Don't just restate the code in English

- i++; // Increment i
  - Bad
- i++; // Make i be 1 bigger
  - Still bad
- i++; // Move to next vector element
  - Better

Comments

- Each of these should have a comment
  - Entire program
  - Every function (except main())
    - Mention parameters and return value
  - Every large code paragraph
  - Every important variable
  - Complex or hard-to-understand statements and expressions

Comment syntax

```
int  i;   // Comment to end-of-line

int  i;   /* Comment with begin
             and end */

int  i;
#if  0
Large block comment
#endif
```

Comment syntax

- //
  - Best for small comments
- /* */
  - Can appear anywhere
  - Not nestable

```
if ((d >= 1) && (d /*<=*/ maxDays))
if ((d >= 1) /* && (d /*<=*/ maxDays)*/)
if ((d >= 1) maxDays)*/)
```

Comment syntax

- #if 0
- #endif
  - Must appear on own line
  - Nestable
  - Most useful for removing large sections of code while debugging/ testing

Bracketing

- Several styles - just be consistent

```
if (a == b) do_something();

if (b == c)
{
        part_1();
        part_2();
}
else
        something_else();
```

Another bracketing style

```
if (a == b) {
        do_something();
}

if (b == c) {
        part_1();
        part_2();
} else {
        something_else();
}
```

Simple g++ errors

- Line number usually most useful

```
1: #include <iostream>
2: int main() {
3:         int i = 42;
4:         int* p = *i;
5:         cout << &p << endl;
6: }
bash$ g++ -Wall -g -o prog prog.cc
prog.cc: In function `int main()':
prog.cc:4: invalid type argument of `unary *'
```