ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Εισαγωγή στον Προγραμματισμό
Introduction to Programming

Διάλεξη 6: Ολοκλήρωση προγράμματος

Γ. Παπαγιαννάκης

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΣΠΑ 2007-2013
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Άδειες Χρήσης

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

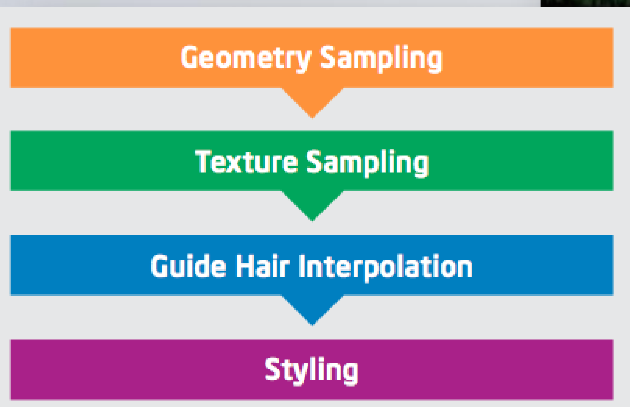
HY-150 Προγραμματισμός
CS-150 Programming

Lecture 6:
Completing a program

G. Papagiannakis

for loops can be critical…

The Parallel Universe, Intel, Issue 9, February 2012




```
void Fur::generateFur()
{
  for(size_t i=0; i<numHairs; i++)
  {
      // compute each hair
  }
}
```

```
void Fur::generateFur()
{
  tbb::parallel_for(
      tbb::blocked_range<size_t>(0,numHairs),
      [=](const tbb::blocked_range<size_t> &r)
      {
          for(size_t i = r.begin() i<r.end(); i++),
          {
              // compute each hair
          }
      } );
}
```

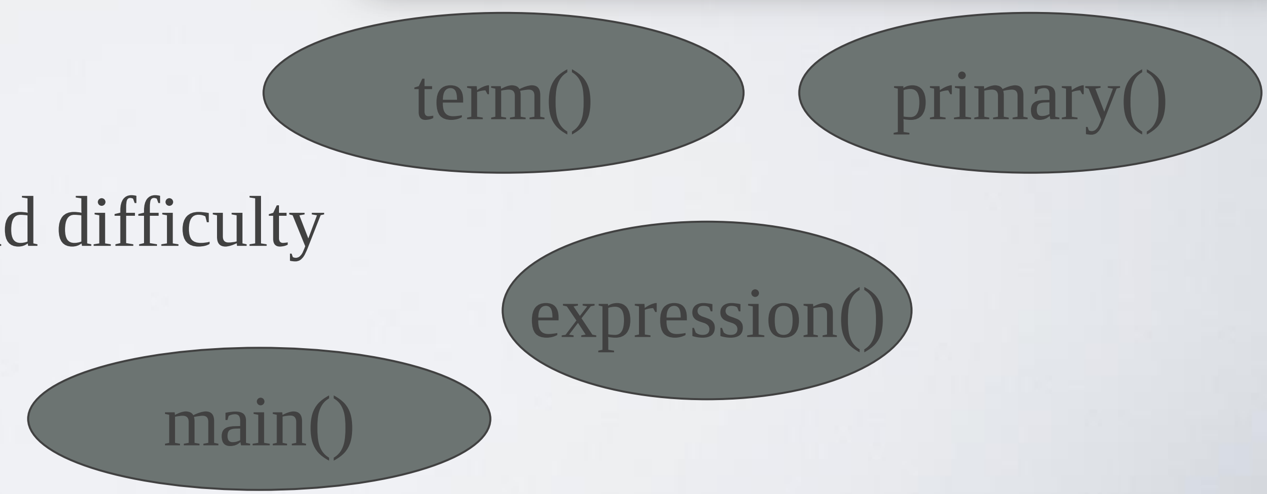
Figure 5: Left-hand side—original fur shader code. Right-hand side—modified code using Intel® TBB `parallel_for` with C++ lambda expression

KungFu Panda 2 HD Trailer (implementation of Fur Shader using parallelized for loops), http://www.youtube.com/watch?v=YdaMGcOyfjM

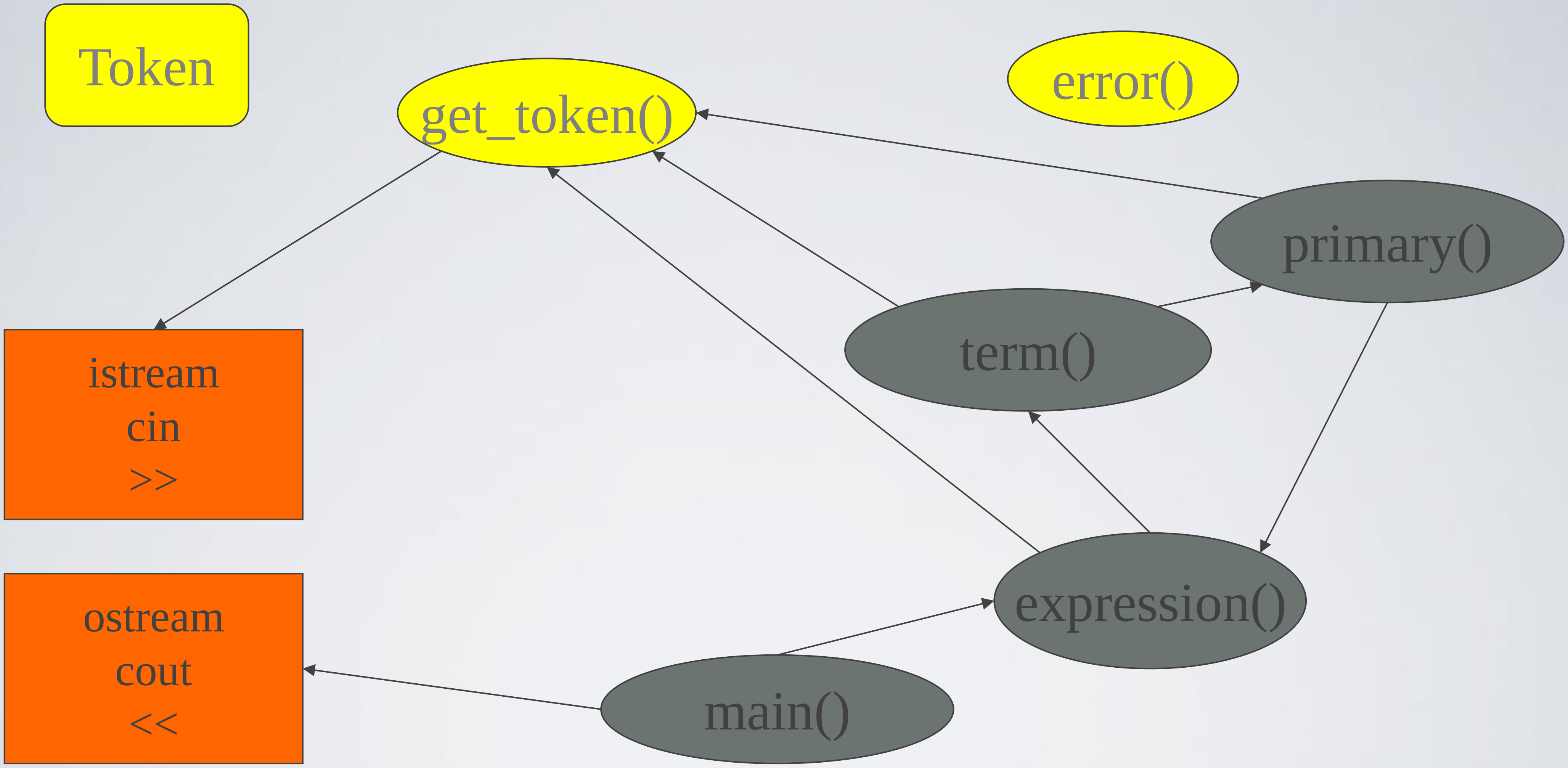
Overview

- Tokens and token streams
  - Classes and structs
- Cleaning up the code
  - Prompts
  - Program organization
    - constants
  - Recovering from errors
  - Commenting
  - Code review
  - Testing
- A word on complexity and difficulty

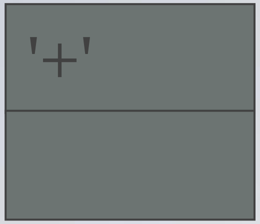
```
Expression:
        Term
        Expression "+" Term        // addition
        Expression "–" Term        // subtraction
Term:
        Primary
        Term "*" Primary           // multiplication
        Term "/" Primary           // division
        Term "%" Primary           // remainder (modulo)
Primary:
        Number
        "(" Expression ")"         // grouping
Number:
        floating-point-literal
```



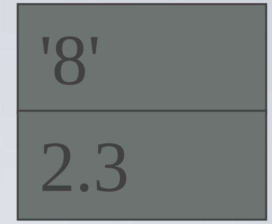
Calculator Program organization



- What we implemented in the last lecture



Token



- We want a type that can hold a “kind” and a value:

```
class Token {  // define a type called Token
    char kind;   // what kind of token
    double value;           // used for numbers (only): a value
    // …
};

Token t;
t.kind = '8';               // . (dot) is used to access members
                            // (use ‘8’ to mean “number”)
t.value = 2.3;

Token u = t;                // a Token behaves much like a built-in type, such as int
                            // so u becomes a copy of t
cout << u.value;            // will print 2.3
```

Token

```
class Token {         // user-defined type called Token
    // data members
    // function members
};
```

- A **struct** is the simplest form of a class
- "**class**" is C++'s term for "user-defined type"
- Defining types is the crucial mechanism for organizing programs in C++
  - as in most other modern languages
- a **class** (including **struct**s) can have
  - data members (to hold information), and
  - function members (providing operations on the data)

Token

```
class Token {
  char kind;                 // what kind of token
  double value;              // for numbers: a value

  Token(char ch) : kind(ch), value(0) { }                      // constructor
  Token(char ch, double val) : kind(ch), value(val) { }        // constructor
};
```

- A constructor has the same name as its class
- A constructor defines how an object of a class is initialized
  - Here **kind** is initialized with **ch,** and
  - **value** is initialized with **val** or **0**
  - Token('+'); // make a **Token** of “kind” **‘+’**
  - Token('8',4.5); // make a **Token** of “kind” **‘8’** and value **4.5**

Token get_token()

```
Token get_token()    // read a token from cin
{
    char ch;
    cin >> ch;                  // note that >> skips whitespace (space, newline, tab, etc.)

switch (ch) {
    case '(': case ')': case '+': case '-': case '*': case '/':
        return Token(ch);       // let each character represent itself
    case '.':
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        {
            cin.putback(ch);                    // put digit back into the input stream
            double val;
            cin >> val;                                 // read a floating-point number

            return Token('8',val);   // let '8' represent "a number"
        }
    default:
        error("Bad token");
    }
}
```

Token_stream

```
Expression:
    Term
    Expression "+" Term      // addition
    Expression "–" Term      // subtraction
Term:
    Primary
    Term "*" Primary         // multiplication
    Term "/" Primary         // division
    Term "%" Primary         // remainder (modulo)
Primary:
    Number
    "(" Expression ")"       // grouping
Number:
    floating-point-literal
```

- A **Token_stream** reads characters, producing **Token**s on demand
- We can put a **Token** into a **Token_stream** for later use
- A **Token_stream** uses a "buffer" to hold tokens we put back into it

Token_stream buffer: empty

Input stream: 1+2*3;

- For 1+2*3;, expression() calls term() which reads 1, then reads +,
- decides that + is a job for "someone else" and puts + back in the Token_stream
- (where expression() will find it)

Token_stream buffer: Token('+')

Input stream: 2*3;

Token_stream

- A **Token_stream** reads characters, producing **Token**s
- We can put back a **Token**

```
class Token_stream {
   // representation: not directly accessible to users:
   bool full;            // is there a Token in the buffer?
   Token buffer;    // here is where we keep a Token put back using putback()
public:
   // user interface:
   Token get();                    // get a Token
   void putback(Token); // put a Token back into the Token_stream
   Token_stream();          // constructor: make a Token_stream
};
```

- A constructor
  - defines how an object of a class is initialized
  - has the same name as its class, and no return type

Token_stream implementation

```
class Token_stream {
   bool full;          // is there a Token in the buffer?
   Token buffer; // here is where we keep a Token put back using putback()
public:
   Token get();                          // get a Token
   void putback(Token);                  // put back a Token
   Token_stream() :full(false), buffer(0) { }        // the buffer starts empty
};

void Token_stream::putback(Token t)
{
   if (full) error("putback() into a full buffer");
   buffer=t;
   full=true;
}
```

Token_stream implementation

```
Token Token_stream::get()            // read a Token from the Token_stream
{
    if (full) { full=false; return buffer; }  // check if we already have a Token ready

    char ch;
    cin >> ch;                  // note that >> skips whitespace (space, newline, tab, etc.)

    switch (ch) {
    case '(': case ')': case ';': case 'q': case '+': case '-': case '*': case '/':
            return Token(ch);                   // let each character represent itself
    case '.':
    case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
    {       cin.putback(ch);                    // put digit back into the input stream
            double val;
            cin >> val;             // read a floating-point number
            return Token('8',val);  // let '8' represent "a number"
    }
    default:
            error("Bad token");
    }
}
```

Streams

- Note that the notion of a stream of data is extremely general and very widely used
  - *Most I/O systems*
    - *E.g.*, C++ standard I/O streams
  - with or without a putback/unget operation
    - We used putback for both **Token_stream** and **cin**

The calculator is primitive

- We can improve it in stages
  - Style – clarity of code
    - Comments
    - Naming
    - Use of functions
    - ...
  - Functionality – what it can do
    - Better prompts
    - Recovery after error
    - Negative numbers
    - % (remainder/modulo)
    - Pre-defined symbolic values
    - Variables
    - …

Error handling

- The first thing to do once you have a program that "basically works"
- Is to try to "break" it: feeding extreme input in order to find errors
  - Technically this is known as "testing"
  - E.g. this gives an error:
    - -1/2
    - We have to write:
      - (0-1)/2
      - To fix it we must allow the grammar to handle unary minus

```
1+2+3+4+5+6+7+8
1–2–3–4
!+2
;;;
(1+3;
(1+);
1*2/3%4+5–6;
();
1+;
+1
1++;
1/0
1/0;
1++2;
–2;
–2;;;;
1234567890123456;
'a';
q
1+q
1+2; q
```

```
Primary:
    Number
    "(" Expression ")"
```

Error handling: negative numbers

```
Primary:
        Number
        "(" Expression ")"
        "–" Primary
        "+" Primary
```

```
double primary()
{
        Token t = ts.get();
        switch (t.kind) {
        case '(':      // handle '(' expression ')'
                {
                        double d = expression();
                        t = ts.get();
                        if (t.kind != ')') error("')' expected");
                        return d;
                }
        case '8':                       // we use '8' to represent a number
                return t.value;         // return the number's value
        case '–':
                return – primary();
        case '+':
                return primary();
        default:
                error("primary expected");
        }
}
```

Error handling: remainder

- Add % as a Token inside term()
- Convert **double** to **int**
  - so that we can use the % on them

```
case '%':
{       int i1 = narrow_cast<int>(left);
        int i2 = narrow_cast<int>(term());
        if (i2 == 0) error("%: divide by zero");
        left = i1%i2;
        t = ts.get();
        break;
}
```

- int xl = narrow_cast<int>(2.9); //error
- int x2=narrow_cast<int>(2.0); //ok

- narrow_cast<> defined in std_lib_facilities.h
- Note the <> are same as are used for vector<int>

Prompting

- Initially we said we wanted

  Expression: 2+3; 5*7; 2+9;

  Result : 5

  Expression: Result: 35

  Expression: Result: 11

  Expression:

- But this is what we implemented

  2+3; 5*7; 2+9;

  5

  35

  11

- What do we really want?

  > 2+3;

  = 5

  > 5*7;

  = 35

  >

Adding prompts and output indicators

```
double val = 0;
cout << "> ";                                  // print prompt
while (cin) {
   Token t = ts.get();
   if (t.kind == 'q') break;          // check for "quit"
   if (t.kind == ';')
          cout << "= " << val << "\n > ";   // print "= result" and prompt
   else
          ts.putback(t);
   val = expression();                        // read and evaluate expression
}
```

```
> 2+3; 5*7; 2+9;
= 5
> = 35
> = 11
>
```

the program doesn't see input before you hit "enter/return"

“But my window disappeared!”

- Test case: +1;

```
cout << "> ";                                   // prompt
while (cin) {
        Token t = ts.get();
        while (t.kind == ';') t=ts.get();// eat all semicolons
        if (t.kind == 'q') {
                keep_window_open("~~");
                return 0;
        }
        ts.putback(t);
        cout << "= " << expression() << "\n > ";
}
keep_window_open("~~");
return 0;
```

The code is getting messy

- Bugs thrive in messy corners
- Time to clean up!
  - Read through all of the code carefully
    - Try to be systematic (“have you looked at all the code?”)
  - Improve comments
  - Replace obscure names with better ones
  - Improve use of functions
    - Add functions to simplify messy code
  - Remove “magic constants”
    - E.g. **'8'** (**'8'** what could that mean? Why **'8'**?)
- Once you have cleaned up, let a friend/colleague review the code (“code review”)

Remove “magic constants”

```
// Token “kind” values:
const char number = '8';        // a floating-point number
const char quit = 'q';          // an exit command
const char print = ';';         // a print command

// User interaction strings:
const string prompt = "> ";
const string result = "= ";     // indicate that a result follows
```

Remove “magic constants”

```
// In  Token_stream::get():

    case '.':
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
      {    cin.putback(ch);                    // put digit back into the input stream
           double val;
           cin >> val;                         // read a floating-point number
           return Token(number,val); // rather than Token('8',val)
      }

// In primary():

    case number:  // rather than case '8':
           return t.value;    // return the number's value
```

Remove "magic constants"

```
// In main():

while (cin) {
        cout << prompt;                          // rather than "> "
        Token t = ts.get();
        while (t.kind ==print) t=ts.get();      // rather than ==';'
        if (t.kind == quit) {                   // rather than =='q'
                keep_window_open();
                return 0;
        }
        ts.putback(t);
        cout << result << expression() << endl;
}
```

Remove "magic constants"

- But what's wrong with "magic constants"?
  - Everybody knows **3.14159265358979323846264**, **12**, **-1**, **365**, **24**, **2.7182818284590**, **299792458**, **2.54**, **1.61**, **-273.15**, **6.6260693e-34**, **0.5291772108e-10**, **6.0221415e23** and **42**!
  - No; they don't.
- "Magic" is detrimental to your (mental) health!
  - It causes you to stay up all night searching for bugs
  - It causes space probes to self destruct (well … it can … sometimes …)
- If a "constant" could change (during program maintenance) or if someone might not recognize it, use a symbolic constant.
  - Note that a change in precision is often a significant change **3.14!=3.14159265**
  - **0** and **1** are usually fine without explanation, **-1** and **2** sometimes (but rarely) are.
  - **12** can be okay (the number of months in a year rarely changes), but probably is not (see Chapter 10).
- If a constant is used twice, it should probably be symbolic
  - That way, you can change it in one place

So why did we use “magic constants”?

- To make a point
  - Now you see how ugly that first code was
    - just look back to see
- Because we forget (get busy, etc.) and write ugly code
  - “Cleaning up code” is a real and important activity
    - Not just for students
    - Re-test the program whenever you have made a change
  - Ever so often, stop adding functionality and “go back” and review code
    - It saves time

Recover from errors

- Any user error terminates the program
  - That's not ideal
  - Structure of code

```
int main()
try {
    // … do "everything" …
}
catch (exception& e) {          // catch errors we understand something about
    // …
}
catch(…) {                  // catch all other errors
    // …
}
```

Recover from errors

- Move code that actually does something out of main()
  - leave main() for initialization and cleanup only

```
int main()          // step 1
try {
    calculate();
    keep_window_open(); // cope with Windows console mode
    return 0;
}
catch (exception& e) {                    // errors we understand something about
    cerr << e.what() << endl;
    keep_window_open("~~");
    return 1;
}
catch (...) {                             // other errors
    cerr << "exception \n";
    keep_window_open("~~");
    return 2;
}
```

Recover from errors

- Separating the read and evaluate loop out into **calculate()** allows us to simplify it
  - no more ugly **keep_window_open()** !

```
void calculate()
{
    while (cin) {
            cout << prompt;
            Token t = ts.get();
            while (t.kind == print) t=ts.get();     // first discard all "prints"
            if (t.kind == quit) return;              // quit
            ts.putback(t);
            cout << result << expression() << endl;
      }
}
```

Recover from errors

- Move code that handles exceptions from which we can recover from **error()** to **calculate()**

```
int main()      // step 2
try {
   calculate();
   keep_window_open();          // cope with Windows console mode
   return 0;
}
catch (...) {                   // other errors (don't try to recover)
   cerr << "exception \n";
   keep_window_open("~~");
   return 2;
}
```

Recover from errors

```
void calculate()
{
   while (cin) try {
            cout << prompt;
            Token t = ts.get();
            while (t.kind == print) t=ts.get();  // first discard all "prints"
            if (t.kind == quit) return;          // quit
            ts.putback(t);
            cout << result << expression() << endl;
   }
   catch (exception& e) {
            cerr << e.what() << endl;            // write error message
            clean_up_mess();                     // <<< The tricky part!
   }
}
```

Recover from errors

- First try

```
void clean_up_mess()
{
  while (true) {                // skip until we find a print
   Token t = ts.get();
   if (t.kind == print) return;
  }
}
```

- Unfortunately, that doesn’t work all that well. Why not? Consider the input **1@$z; 1+3;**
  - When you try to **clean_up_mess()** from the bad token **@**, you get a **“Bad token”** error trying to get rid of **$**
  - We always try not to get errors while handling errors

Recover from errors

- Classic problem: the higher levels of a program can't recover well from low-level errors (i.e., errors with bad tokens).
  - Only **Token_stream** knows about characters
- We must drop down to the level of characters
  - The solution must be a modification of **Token_stream**:

```
class Token_stream {
    bool full;                    // is there a Token in the buffer?
    Token buffer;                 // here is where we keep a Token put back using putback()
public:
    Token get();                  // get a Token
    void putback(Token t);        // put back a Token
    Token_stream();               // make a Token_stream that reads from cin
    void ignore(char c);          // discard tokens up to and including a c
};
```

Recover from errors

```
void Token_stream::ignore(char c)
   // skip characters until we find a c; also discard that c
{
   // first look in buffer:
   if (full && c==buffer.kind) {     // && means and
            full = false;
            return;
   }
   full = false;      // discard the contents of buffer
   // now search input:
   char ch = 0;
   while (cin>>ch)
            if (ch==c) return;
}
```

Recover from errors

- clean_up_mess() now is trivial
  - and it works

```
void clean_up_mess()
{
   ts.ignore(print);
}
```

- Note the distinction between what we do and how we do it:
  - **clean_up_mess()** is what users see; it cleans up messes
    - The users are not interested in exactly how it cleans up messes
  - **ts.ignore(print)** is the way we implement **clean_up_mess()**
    - We can change/improve the way we clean up messes without affecting users

Features

- We did not (yet) add
  - Pre-defined symbolic values
  - Variables
  - Check Chapter 7 in the book to see them implemented

- Major Point
  - Providing "extra features" early causes major problems, delays, bugs, and confusion
  - "Grow" your programs
    - First get a simple working version
    - Then, add features that seem worth the effort

commenting

- When you go back to your code to clean it up, make sure that the comments you wrote while writing the code are:
  - Still valid (you might have changed the code since the comments)
  - Adequate for another reader (usually they are not except for you)
  - Not so verbose that they distract from the code
- Avoid comments that explain something perfectly clear e.g.:
  - x = b + c; //add b and c and assign the result to x
- Instead write comments for things that code cannot express:
  - E.g. intent (πρόθεση)
    - The code says what it does, not what was intended to do

Next lecture

- In the next two lectures, we'll take a more systematic look at the C++ language features we have used so far. In particular, we need to know more about classes, functions, statements, expressions, and types.

Acknowledgements

Bjarne Stroustrup

Programming -- Principles and Practice Using C++

http://www.stroustrup.com/Programming/

Thank you!

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ