

CSC 221: Computer Programming I

Fall 2001

- top-down design
 - approach to problem solving, program design
 - focus on tasks, subtasks, ...
- reference parameters
 - provide mechanism for functions to "return" more than one value
 - reference vs. value parameters

programmer's toolbox

you now have an extensive collection of programming tools for solving problems

- variables – for storing values
- expressions – for performing computations
- functions – for computational abstraction
- if statements – for conditional execution
 - if (1-way), if-else (2-way), cascading if-else (multi-way)
- while loops – for conditional repetition
 - counters & sums for keeping running totals
 - for loop variant if you know the number of repetitions

problem-solving is all about organizing the right tools for the job

- (1) identify the tools that would be useful
- (2) sketch out a solution in pseudo-code (i.e., organize the tools)
- (3) fill in the details to complete the code

Sprint (cont.)

GetUserOption handles all the details of reading the user's choice

simple enough to be coded

- display menu of options
- repeatedly read user input until valid (1 or 2)
- return user input

```
int GetUserOption()
// Returns: user input, either 1 (for specific plan) or
//         2 (for a comparison of all plans)
{
    cout << endl << "Are you interested in:" << endl
         << " (1) a specific plan, or" << endl
         << " (2) a comparison of all plans?" << endl
         << "Select an option: ";

    int userOption;
    cin >> userOption;

    while (userOption != 1 && userOption != 2) {
        cout << "illegal option. Enter 1 or 2: ";
        cin >> userOption;
    }

    return userOption;
}
```

note: you were not required to do this for HW3
(hadn't seen loops yet)

Sprint (cont.)

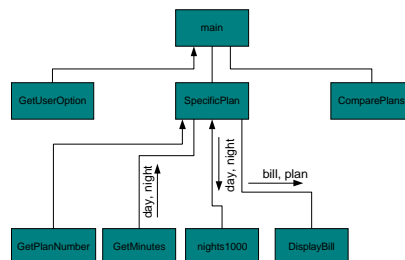
SpecificPlan is too complex a task to start coding – break down into subtasks

TASK 1: get plan number

TASK 2: get minutes (day & night)

TASK 3: compute bill using specified plan

TASK 4: display bill



Sprint (cont.)

```
void SpecificPlan()
// Results: computes and displays bill using all Sprint plans.
{
    int plan = GetPlanNumber();

    int dayMinutes, nightMinutes;
    GetMinutes(dayMinutes, nightMinutes);

    cout << endl << "Your bill would be:" << endl;
    if (plan == 1) {
        DisplayBill(AnyNickel(dayMinutes+nightMinutes), "Nickel Anytime");
    }
    else if (plan == 2) {
        DisplayBill(AnySense(dayMinutes+nightMinutes), "Sense Anytime");
    }
    else if (plan == 3) {
        DisplayBill(Any500(dayMinutes+nightMinutes), "500 Anytime");
    }
    else if (plan == 4) {
        DisplayBill(Any1000(dayMinutes+nightMinutes), "1000 Anytime");
    }
    else if (plan == 5) {
        DisplayBill(NightsNickel(dayMinutes, nightMinutes), "Nickel Nights");
    }
    else if (plan == 6) {
        DisplayBill(Nights1000(dayMinutes, nightMinutes), "1000 Nights");
    }
}
```

problem: we want to delegate the task of reading day & night minutes to a function – but functions can only return a single value (LATER)

Sprint (cont.)

```
int GetPlanNumber()
// Returns: user input, integer between 1 and 6
{
    cout << endl << "Which plan are you interested in:" << endl
    << " (1) Sprint Nickel Anytime" << endl
    << " (2) Sprint Sense Anytime" << endl
    << " (3) Sprint 500 Anytime" << endl
    << " (4) Sprint 1000 Anytime" << endl
    << " (5) Sprint Nickel Nights" << endl
    << " (6) Sprint 1000 Nights" << endl
    << "Select an option: ";

    int plan;
    cin >> plan;

    while (plan < 1 || plan > 6) {
        cout << "Illegal option. Enter 1-6: ";
        cin >> plan;
    }

    return plan;
}
```

```
void DisplayBill(double cost, string planName)
// Assumes: cost >= 0, planName is name of the billing plan
// Results: displays the cost and name of the plan
{
    cout << setiosflags(ios::fixed) << setprecision(2);
    cout << " $" << cost << " using Sprint " << planName << endl;
}
```

Sprint (cont.)

```
double AnyNickel(int totalMinutes)
// Assumes: user spent totalMinutes on long distance calls
// Returns: bill using Nickel Anytime plan
{
    const double MONTHLY_FEE = 8.95;
    const double PER_MINUTE = 0.05;

    return MONTHLY_FEE + (PER_MINUTE * totalMinutes);
}

double Any500(int totalMinutes)
// Assumes: user spent totalMinutes on long distance calls
// Returns: bill using 500 Anytime plan
{
    const double MONTHLY_FEE = 25.00;
    const double PER_MINUTE = 0.10;

    if (totalMinutes > 500) {
        return MONTHLY_FEE + (PER_MINUTE * (totalMinutes-500));
    }
    else {
        return MONTHLY_FEE;
    }
}

double Nights1000(int dayMinutes, int nightMinutes)
// Assumes: number of long distance day and night minutes, resp.
// Returns: bill using 1000 Nights plan
{
    const double MONTHLY_FEE = 20.00;
    const double PER_MINUTE = 0.05;

    if (nightMinutes > 1000) {
        return MONTHLY_FEE + (PER_MINUTE * dayMinutes)
            + (PER_MINUTE * (nightMinutes-1000));
    }
    else {
        return MONTHLY_FEE + (PER_MINUTE * dayMinutes) ;
    }
}
}
```

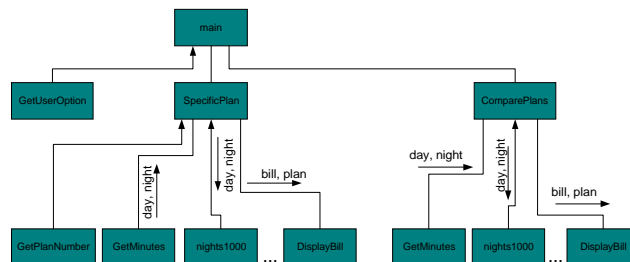
Sprint (cont.)

similarly, `ComparePlans` can be broken down into subtasks

TASK 1: get minutes (day & night)

TASK 2: compute bill using all plans

TASK 3: display bills



Sprint (cont.)

note: we have already defined functions for the necessary tasks
(due to overlap between SpecificPlans and ComparePlans

→ computational abstraction, code reuse

```
void ComparePlans()
// Returns: computes bill for user using different Sprint
// plans and display the results
{
    int dayMinutes, nightMinutes;
    GetMinutes(dayMinutes, nightMinutes);

    cout << endl << "Your bill would be:" << endl;
    DisplayBill(AnyNickel(dayMinutes+nightMinutes), "Nickel Anytime");
    DisplayBill(AnySense(dayMinutes+nightMinutes), "Sense Anytime");
    DisplayBill(Any500(dayMinutes+nightMinutes), "500 Anytime");
    DisplayBill(Any1000(dayMinutes+nightMinutes), "1000 Anytime");
    DisplayBill(NightsNickel(dayMinutes, nightMinutes), "Nickel Nights");
    DisplayBill(Nights1000(dayMinutes, nightMinutes), "1000 Nights");
}
```

reference parameters

top-down design involves breaking a problem into tasks

- define a function for each task, divide into subtasks as necessary
- must be able to pass information as needed between subtasks (functions)

- can pass information into a function via parameters
- can pass a single value back out as the function return value
- what if a function needs to return more than one value?

C++ provides an alternative method of parameter passing

- *reference* parameters allow functions to pass values back to the calling environment
- identified by placing '&' in between the parameter type and name in the function definition

Sprint (cont.)

with reference parameters, changes made to the parameters in the function affect the argument in the function call

```
void GetMinutes(int & day, int & night)
// Results: reads day and night minutes from user, returns via parameters
{
    cout << endl;
    cout << "How many daytime (7am to 7pm) minutes did you foresee using in a month? ";
    cin >> day;

    while (day < 0) {
        cout << "You can't use negative minutes. Try again: ";
        cin >> day;
    }

    cout << "How many night (7pm to 7am) minutes did you foresee using in a month? ";
    cin >> night;

    while (day < 0) {
        cout << "You can't use negative minutes. Try again: ";
        cin >> night;
    }
}
```

```
.....
int dayMinutes, nightMinutes;
GetMinutes(dayMinutes, nightMinutes);
.....
```

[view complete program](#)

value vs. reference parameters

by default, parameters are passed *by-value*

- a copy of the argument is stored in the parameter (a local variable)
- result: value passed in, no changes are passed out

```
void foo(int x)
{
    x = 5;
    cout << x << endl;
}

int a = 3;
foo(a);

cout << a << endl;

foo(3);
```

note: argument can be any value

& implies *by-reference*

- the parameter does not refer to a new piece of memory – it is an alias for the argument
- result: changes to the parameter simultaneously change the argument

```
void foo(int & x)
{
    x = 5;
    cout << x << endl;
}

int a = 3;
foo(a);

cout << a << endl;
```

note: argument must be a variable

diabolical example

```
#include <iostream>
using namespace std;

void Increment(int & x, int & y)
{
    x++;
    y++;
    cout << "Inside: " << x << ", " << y << endl;
}

int main()
{
    int a = 3, b = 7;
    Increment(a, b);

    cout << "Outside: " << a << ", " << b << endl;

    int q = 1;
    Increment(q, q);

    cout << "Outside: " << q << endl;

    return 0;
}
```

top-down design summary

design first, code later

"The sooner you start coding, the longer it will take to finish."

top-down design focuses on dividing complex problems into tasks

- can further divide tasks into subtasks, ..., until simple enough to code
- natural to implement each task with a function

- need to be able to pass information from one subtask (function) to another as needed
 - pass info into a function via (value) parameters
 - if need to pass one value out, do so via return value
 - if need to pass more than one value out, do so via reference parameters