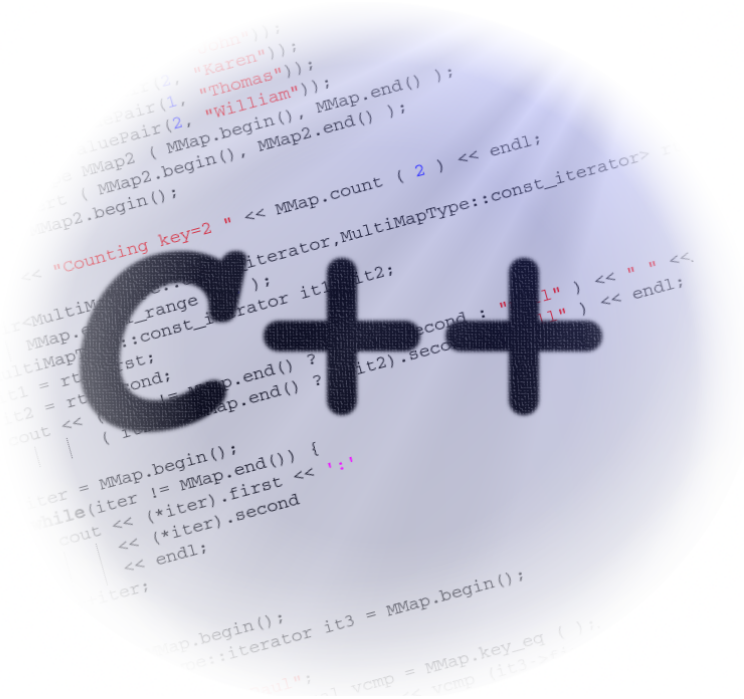
Języki Programowania

Prowadząca:
dr inż. Hanna Zbroszczyk

e-mail: *hanna.zbroszczyk@pw.edu.pl*
tel: +48 22 234 58 51

Konsultacje:
piątek: 14.00 – 15.30

www: http://www.if.pw.edu.pl/~gos/students/jp



Politechnika Warszawska
Wydział Fizyki
Pok. 117b (wejście przez 115)

OBSŁUGA WYJĄTKÓW

Obsługa wyjątków

Sytuacje wyjątkowe – sytuacje, jakie mogą mieć miejsce, choć nie muszą (np. kiedy funkcja nie może się z jakiś powodów wykonać się poprawnie). Poprawnie napisany program "przewidzi", co ma wydarzyć się w przypadku ich wystąpienia. Sytuacją wyjątkową może być wszystko, co za takową zostanie uznane.

Jak napisać poprawnie program z obsługą sytuacji wyjątkowych?

1) Określić, kiedy zaczyna się obszar spodziewanego wystąpienia sytuacji wyjątkowej (blok *try*)
2) W przypadku wystąpienia sytuacji wyjątkowej należy zasygnalizować jej wystąpienie (instrukcja *throw*)
3) Reakcja na wystąpienie sytuacji wyjątkowej (blok *catch*)

Po co są wyjątki?

… wykorzystywane w wielu sytuacjach...

… kiedy dwa odmienne fragmenty kodu mają ze sobą współpracować (jeden wykrywa sytuację wyjątkową – funkcja biblioteczna, drugi ją obsługuje..)

Obsługa sytuacji wyjątkowych

```
//program...
try {
        //instrukcje w trakcie wykonywania ktorych
        //moze zajsc sytuacja wyjatkowa...
        throw ...
}
catch(type1) {
        //obsluga sytuacji wyjatkowej sygnalizowanej
        //za pomoca rzucenia wyjatku type1
}
catch(type2) {
        //obsluga sytuacji wyjatkowej sygnalizowanej
        //za pomoca rzucenia wyjatku type2
}
...
catch(typen) {
        //obsluga sytuacji wyjatkowej sygnalizowanej
        //za pomoca rzucenia wyjatku typen
}
//dalsze instrukcje programu
```

Obsługa sytuacji wyjątkowych

Kolejność bloków catch ma znaczenie..

```
if (typ1) {
        ...
}
else if (typ2) {
        ...
}
...
else if (typn) {

}
else {
        ...
}
```

```
try {
        throw ...
}
catch(type1) {
        ...
}
catch(type2) {
        ...
}
...
catch(typen) {
        ...
}
catch (...) {
        throw;
}
```

Zostanie wykonany pierwszy napotkany warunek, który zostanie spełniony

(bez znaczenia jest fakt, że kolejny byłby lepszy)

Bloki *try*, *catch* można zagnieżdżać.

Wywołanie zwrotne, a rzucanie wyjątku

1) Wywołanie zwrotne (ang. call back) – pomocnicza funkcja “ratunkowa”, funkcja biblioteczna

Wykrywa sytuację wyjątkową, wywołuje zewnętrzną funkcję “ratunkową” (w wywołaniu funkcji bibliotecznej znajduje się adres funkcji pomocniczej). **Wada: uzależnienie od środowiska wykonywania.**

2) Rzucenie wyjątku – **wariant nie uzależniający od środowiska wykonywania**. Funkcja biblioteczna

Sygnalizuje sytuację wyjątkową, jeśli program ją uruchamiający nie będzie potrafił tej sytuacji obsłużyć,

To I tak się zzlinkuje, a problem pojawi się dopiero w przypadku wystąpienia takiej sytuacji.

Dwa sposoby przeniesienia informacji o sytuacji wyjątkowej

1) Wystrzelenie rakiety – spada nie wiadomo gdzie, ale wiadomo jaki ma kolor I to wystarczy (**flaga**)

2) Rzucenie wartościowego obiektu – oprócz jego typu ważne jest, jakim jest **obiektem**

Mechanizm throw to “inny” powrót z funkcji (niż return)

Zagadnienie	***return***	***throw***
Typ zwracanej wartości	Wszystkie wystąpienie muszą zwracać obiekt tego samego typu (typ rezultatu funkcji)	Każda z instrukcji może wyrzucać obiekt innego typu (różne sytuacje wyjątkowe)
Sterowanie programu	.. powraca do funkcji, która daną funkcję wywołała	.. przenosi się bezpowrotnie w zupełnie inny obszar programu, gdzie obsługiwane są sytuacje wyjątkowe

Kilka zasad

Argument rzucany (argument wyjątku) – stoi przy *throw*

Argument oczekiwany – stoi przy *catch*

1) Dana procedura nadaje się do obsługi danego typu wyjątku:

- typ argumentu rzucanego == typ argumentu oczekiwanego
- typ argumentu oczekiwanego ma przydomek *const*
- argument rzucany jest danego typu, a argument oczekiwany jest referencją tego typu

2) Dana procedura nadaje się do obsługi danego typu wyjątku:

- typ argumentu oczekiwanego przez blok *catch* jest publiczną klasą podstawową typu argumentu Rzucanego przez *throw*

3) Dana procedura nadaje się do obsługi danego typu wyjątku:

- obiekt rzucany przez *throw* jest wskaźnikiem typu A, a blok *catch* oczekuje wskaźnika B, który Może być zamieniony drogą konwersji

PRZESTRZENIE NAZW

Przestrzenie nazw - 1

Przestrzeń nazw - zbiór obiektów, ograniczający dostęp do nich.

Oprócz nazwy obiektu niezbędne jest też wspomnienie, z której przestrzeni nazw chcemy go użyć, obchodząc tym samym problem konfliktu nazw.

```
#include <iostream>
using namespace std;
int main() {
	char lit;
	cout<<"Podaj znak.."<<endl;
	cin>>lit;
	cout<<"Wprowadziles znak: <<lit<<endl;
	return 1;
}
```

Przestrzenie nazw - 2

```
#include <iostream>
int main() {
        char lit;
        std::cout<<"Podaj znak.."<<std::endl;
        std::cin>>lit;
        std::cout<<"Wprowadziles znak: <<lit<<std::endl;
        return 1;
}
```

```
#include <iostream>
using namespace std::endl;
using namespace std::cin;
using namespace std::cout;
int main() {
        char lit;
        cout<<"Podaj znak.."<<endl;
        cin>>lit;
        cout<<"Wprowadziles znak: <<lit<<endl;
        return 1;
}
```

Tworzenie własnej przestrzeni nazw

```
namespace Mathematica {
        int add (int a, int b) {
                return a+b;
        }
        int diff (int a, int b) {
                 return a-b;
        }
        int mult (int a, int b) {
                return a+b;
        }
        int div (int a, int b) {
                 return a-b;
        }
}
...
int main() {
        Matematyka::add(1,2);
        return 1;
}
```

PREPROCESOR

Dyrektywa #define

#define wyraz ciag_znakow_zastepujacych_go

Przy kompilacji każde wystąpienie słowa wyraz (lub ciagu znaków bez znakow białych)

będzie zastępowane przez ciag_znakow_zastepujacych_go

```
#define N 10
int tab[N];
for (int i=0; i<N-1; i++) {
      ...
}

#define N 3
#define M 5
int tab[N][M];
for (int i=0; i<N-1; i++) {
      for (int j=0; j<M-1; j++)
      ...
}
```

Dyrektywa #undef

```
#define N 10
int tab[N];
for (int i=0; i<N-1; i++) {
        ...
}
#undef N

#define N 3
#define M 5
int tab[N][M];
for (int i=0; i<N-1; i++) {
        for (int j=0; j<M-1; j++)
        ...
}
#undef M
```

Makrodefinicje, sklejacz ##, zmiana parametru na string'a

```
#define KWADRAT(a) ((a)*(a))
B= KWADRAT(c) + KWADRAT(d);
B= ((c)*(c)) + ((d)*(d));


ale..
D= KWADRAT(x++);
// i co teraz?
```

```
#define ST(co, jaki) co##jaki
int ST(lewo, prawo);
int lewoprawo;
```

```
#define M(z) #z
M(ilosc_amunicji)
“ilosc_amunicji”
```

Kompilacja warunkowa - 1

```
#if warunek
      //linie kompilowane warunkowo
#endif
```

```
#if NAZWA == NAZWA2
#if NAZWA >2
#if (NAZWA_A == 2 || NAZWA_B >10)
#if NAZWA == 6 && defined(WERSJA_ROBOCZA)
if NAZWA ==6 && !defined(WERSJA_ROBOCZA)
```

```
defined nazwa
defined(nazwa)
```

Prawdziwe, jeśli wcześniej

```
#define nazwa
```

oraz nie odwołaliśmy:

```
#undef nazwa
```

Kompilacja warunkowa - 2

```
#if warunek
	instrukcje1
#elif
	instrukcje2
#else
	instrukcje3
#endif
```

```
#define PODWOZIE_707 1
#define PODWOZIE_747 2

#define TYP_PODWOZIA PODWOZIE_747

#if (TYP_PODWOZIA==PODWOZIE_707)
	cout<<”Podwozie 707”<<endl;
#elif (TYP_PODWOZIA==PODWOZIE_747)
	cout<<”Podwozie 747”<<endl;
#else
	cout<<”Nie ma takiego podwozia”<<endl;
```

Kompilacja warunkowa - 3

```
#ifdef nazwa
        instrukcje
#endif
```

musi wcześniej być dyrektywa:

#define nazwa

oraz nie została uniewąniona:

#undef nazwa

```
#ifndef nazwa
        instrukcje
#endif
```

Wcześniej nie mogło być dyrektywy

#define nazwa

Lub została ona unieważniona:

#undef nazwa

Dyrektywa #error, #line

.. po jej napotkaniu kompilacja zostaje przerwana:

#error tekst

```
#if (WERSJA ==1)
        instrukcje1;
#elif (WERSJA ==2)
        instrukcje2;
#else
        #error “Musi byc wersja 1 lub 2”
#endif
```

```
#line 128 “file.cpp”
```

To oszukanie kompilatora, myśli, że to 128 linijka programu w pliku o nazwie file.cpp

Wstawianie treści innych plików; d. zależne od implementacji

#include “nazwa_pliku_A”

#include <nazwa_pliku_B>

W tym miejscu zostaną wstawione linie z odpowiedniego pliku

“” - to szukanie w bieżącym katalogu, jeśli nie zostanie znaleziony, to jest szukany <>

#pragma komenda

Jeśli komenda nie jest znana danemu kompilatorowi, to jest ona ignorowana

Nazwy predefiniowane

__LINE__ - numer linii

__FILE__ - nazwa kompilowanego pliku

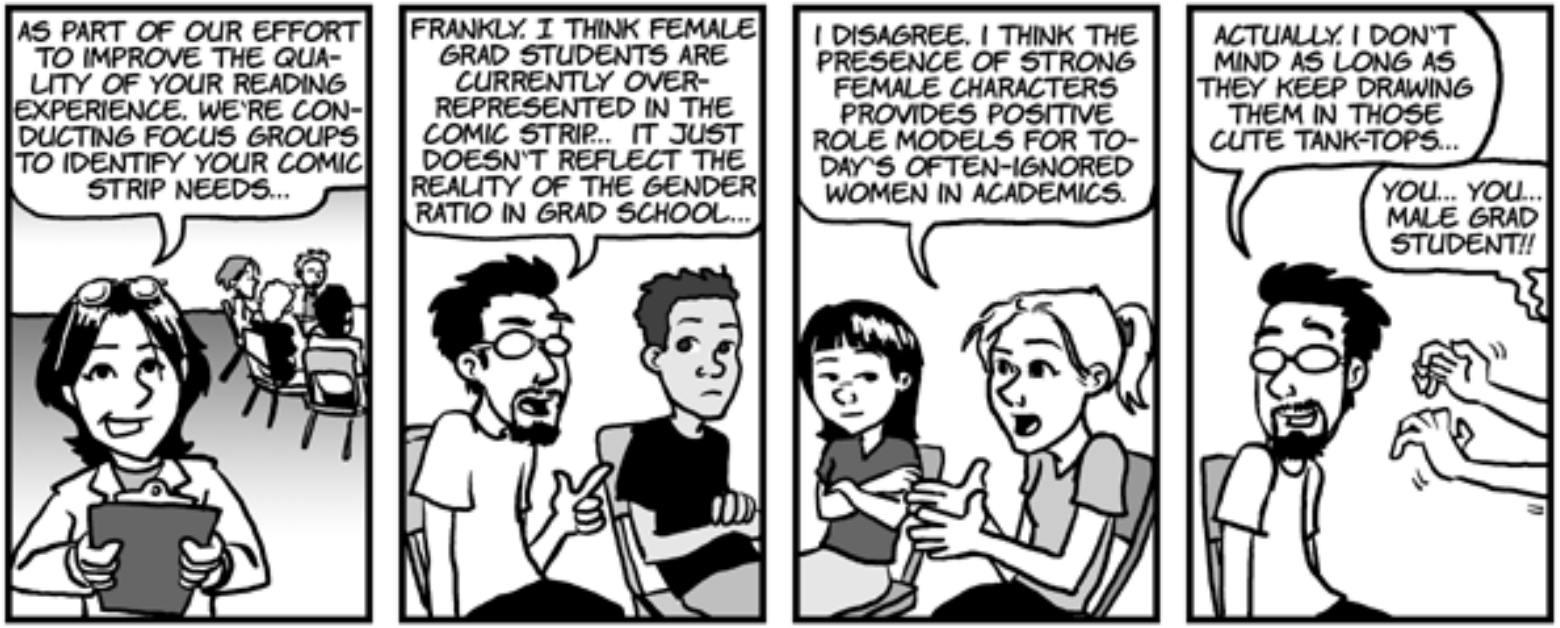
__DATE__ - data w momencie kompilacji

__TIME__ - czas w momencie kompilacji

__cplusplus

__STDC__ - (skrót od StandarD C)

```
cout<<”Kompilacja pliku<<”__FILE__;
cout<<”\n (linia: “<<__LINE__;
cout<<”\n zaczela sie ”__DATE__;
cout<<”\n o godzinie ”<<__TIME__<<endl;
```



ALGORYTMY I STRUKTURY DANYCH

Optymalizacja – wstęp - 1

Wydajność – zależy od wrażenia osoby wykorzystującej dany program. Ocena oprogramowania może być bardzo różna w zależności od doświadczenia I oczekiwań poszczególnych użytkowników.

Wydajność oprogramowania – mierzona liczbą jednostek wejściowych (danych), którymi program w danym czasie zarządza w celu przekształcenia ich na jednostki wyjściowe (dane)

Cechy dobrego oprogramowania (z punku widzenia użytkownika):
- nie wymaga stałej interwencji ze strony użytkownika
- ma intuicyjny interfejs
- można szybko nauczyć się jego obsługi
- jest elastyczny
- zawiera szczegółową I czytelną dokumentację
- operacje wykonywane są bez zauważalnego dla uzytkownika opóźnienia
- informacje są łatwodostępne dla użytkownika

Cechy dobrego oprogramowania (z punktu widzenia programisty):
- jest nastawiony na dalszy rozwój
- prosty w utrzymaniu
- dobry I intuicyjny projekt
- zawiera dobrze napisaną dokumentację techniczną
- może zostać przekazany dowolnemu programiście

Optymalizacja – wstęp - 2

Proces optymalizacji:

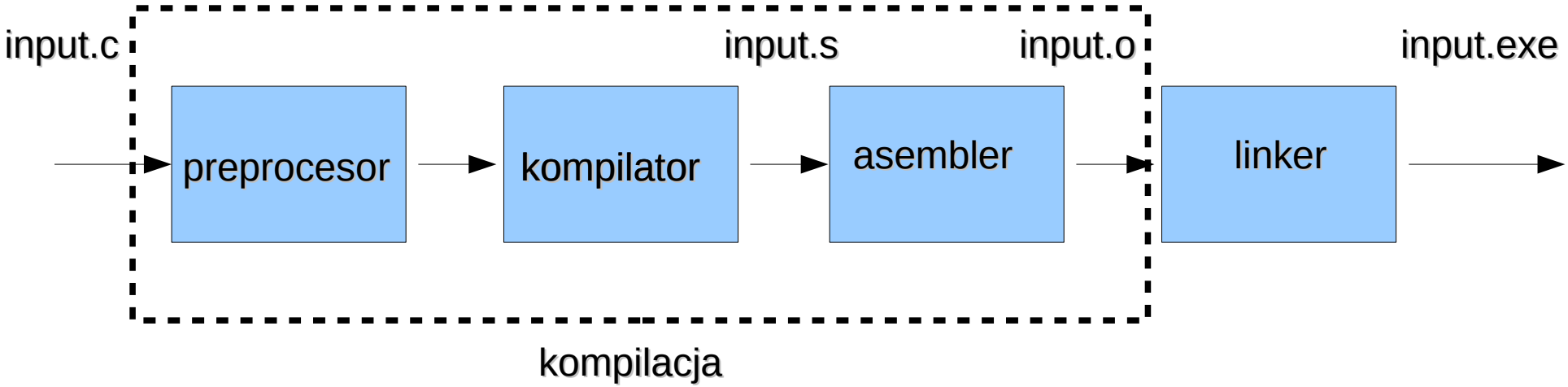
- znajdowanie obszarów nadających się do optymalizacji
- analiza tych obszarów I wybór celów optymalizacji
- wykonanie projektu I realizacja (wybranych) optymalizacji

Po określeniu obszarów optymalizacji można część z nich zastąpić innym oprogramowaniem, a pozostałe zoptymalizować przez dostrojenie istniejącego oprogramowania.

Zastąpienie oprogramowania jest możliwe, jeśli:

- istnieje zamiennik o wystarczającej I znanej jakości
- spełnione są oczekiwania co do możliwości konserwacji zastępczego oprogramowania
- nie ma wystarczającej ilości czasu lub zasobów by ponownie napisać oprogramowanie w ramach projektu

Proces otrzymywania pliku wykonywalnego - 1



Kompilator – narzędzie do przekształcenia programu napisanego w języku wysokiego poziomu (Pascal, C, C++, Java, Perl, Fortran, …) w binarny plik wykonywalny

g++ -o program program.c

Plik program.c zostanie skompilowany, w wyniku czego powstanie plik wykonywalny program.

Proces kompilacji jest kilkustopniowy, można przerwać (lub kontynuować) na dowolnym etapie.

g++ -c -o program.o program.c

plik program.c zostanie skompilowany, w wyniku czego powstanie plik wynikowy (ale nie wykonywalny!) program.o

g++ -E program.c

Proces otrzymywania pliku wykonywalnego - 2

```
#include <stdio.h>


int main() {
        printf("Hello World!\n");
        return 1;
}
```

program.c

program.s →

```
        .file  "program.c"
        .section    .rodata
.LC0:
        .string     "Hello World!"
        .text
.globl main
        .type       main, @function
main:
.LFB0:
        .cfi_startproc
        .cfi_personality 0x3,__gxx_personality_v0
        pushq       %rbp
        .cfi_def_cfa_offset 16
        movq        %rsp, %rbp
        .cfi_offset 6, -16
        .cfi_def_cfa_register 6
        movl$.LC0, %edi
        call   puts
        movl$1, %eax
        leave
        ret
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident      "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"
        .section    .note.GNU-stack,"",@progbits
```

Proces otrzymywania pliku wykonywalnego - 3

Asembler – narzędzie , które tłumaczy mnemoniki (symboliczna reprezentacja tekstowa instrukcji mikroprocesora) na kod mikroprocesora

as program.s -o program.o (utworzenie z kodu asemlera pliku wykonywalnego)

Linker – używany w ostatniej fazie generowania pliku wykonywalnago, pobiera biblioteki I pliki wynikowe łącząc je (może dodawać włąsne pliki I biblioteki systemowe w celu wytworzenia pliku wynikowego

g++ -lm o program program.o

W przypadku projektów rozbudowanych (chociaż można zastosować I w prostych programach) używa się narzędzia np. make.

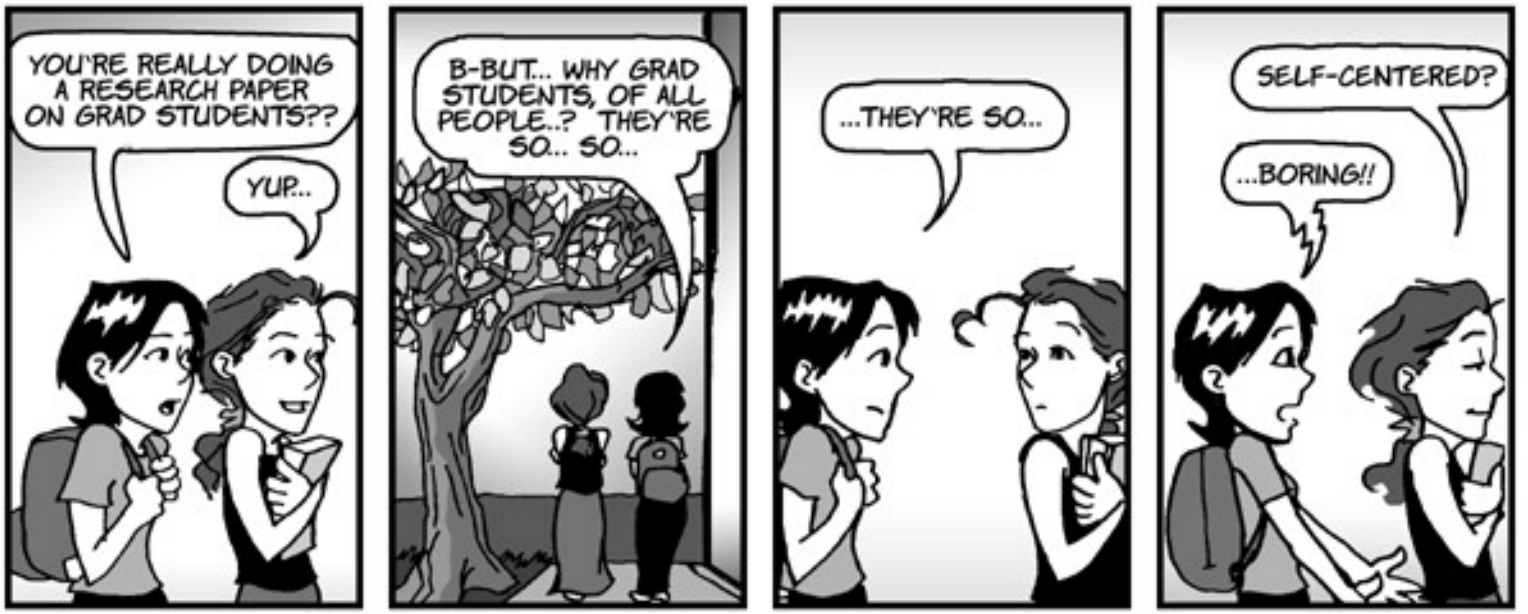
Optymalizacja za pomocą kompilatora

Większość kompilatorów wywołana bez specjalnych opcji optymalizacyjnych dokonuje tylko podstawowych optymalizacji.

Najczęstsza opcja -On (n to poziom optymalizacji, -O3 bardziej rozbudowana optymalizacja niż -O2, itd),
Najniższy poziom optymalizacji to pominięcie opcji -O

g++ -O3 -o program program.c

Dlaczego kompilatory nie optymalizują automatycznie?
- im wyższy poziom optymalizacji tym trudniejsze jest debugowanie kodu
- przy optymalizacji kompilator próbuje łączyć lub pominąć możliwie największą liczbę segmentów kodu
- optymalizacja szybkości wykonywania generowania pliku wykonywalnego może spowodować jego rozrost



STRUKTURY PRZECHOWYWANIA

Struktury przechowywania

1. Tablice:

– struktury najlepsze do stałej ilości danych;

- użycie zmiennej ilości danych jest bardziej skomplikowane, ponieważ dane muszą być wtedy kopiowane (potrzebne będą większe zasoby przy dodawaniu elementów, tablica musi zostać przepisana nawet jeśli zasoby są zmniejszane);
- wyszukiwanie jest proste za względu na bezpośredni dostęp do każdego elementu (znacznie łatwiejsze w przypadku tablicy posortowanej);
- jest zaimplementowana w STL (vector)

2. Listy:

– struktury bardzo dobre do przechowywania małej ilości danych o dynamicznej strukturze;

- powodują więcej kosztów na element niż tablica;
- niebezpieczna jest też fragmentacja;
- lecz dynamiczne dodawanie i usuwanie elementów nie wymaga dodatkowych zasobów (oprócz pamięci niezbędnej do przechowywania zasobów);
- wyszukiwanie elementów jest czasochłonne

Struktury przechowywania

Przykłady list:

- lista jednokierunkowa: zawiera wskaźnik na element następny (pierwszy element: head)
- lista dwukierunkowa: zawiera wskaźnik na element następny oraz element poprzedni (pierwszy element: head, ostatni element: tail)
- lista cykliczna jednokierunkowa: węzły są połączone w pierścień (element ostatni wskazuje na pierwszy)
- lista cykliczna dwukierunkowa
- lista z przeskokami: odmiana listy uporządkowanej, w zwykłej liście każdy element posiada połączenie (ze względu na prostotę implementacji - najczęściej realizowane poprzez wskaźnik) wyłącznie do elementu następnego (ew. I poprzedniego), w liście z przeskokami takich połączeń jest więcej - oprócz bezpośredniego następnika, wskazują także elementy znajdujące się dalej.
- lista (jednokierunkowa) samoorganizująca się: lista sortująca się przy jej wypełnianiu
- tablica podwójnie kończona (kolejka – STL: deque)

Listy (dwukierunkowe) są też dostępne w STL..

3. Stos (STL)

4. Kolejka, kolejka priorytetowa (STL)

Lista jednokierunkowa – przykład – 1 (Adam Drozdek)

```
#include <iostream>
using namespace std;
class IntNode {
        public:
        int nr;
        IntNode *next;
        IntNode(int el, IntNode *ptr=0) {
                nr = el; next = ptr;
        }
};
class IntList {
        IntNode *head, *tail;
        public:
        IntList() { head = tail = 0; }
        ~IntList();
        int isEmpty() { return head == 0; }
        void addToHead(int);
        void addToTail(int);
```

Lista jednokierunkowa – przykład - 2

```
        int deleteFromHead();
        int deleteFromTail();
        void deleteNode(int);
        bool isInList(int);
        void printAll();
};

IntList::~IntList() {
        for (IntNode *p; !isEmpty(); ) {
                p = head->next;
               delete head;
               head = p;
        }
}

void IntList::addToHead(int el) {
        head = new IntNode(el, head);
        if (tail == 0) tail = head;
}
```

Lista jednokierunkowa – przykład - 3

```
void IntList::addToTail(int el) {
        if (tail!=0) {
                tail->next = new IntNode(el);
                tail = tail -> next;
        }
        else head = tail = new IntNode(el);
}

int IntList::deleteFromHead() {
        int el = head->nr;
        IntNode *tmp = head;
        if (head == tail) head = tail =0;
        else head = head->next;
        delete tmp;
        return el;
}
```

Lista jednokierunkowa – przykład - 4

```
int IntList::deleteFromTail() {
        int el = tail->nr;
        if (head == tail) {
                delete head;
                head = tail =0;
        }
        else {
                IntNode *tmp;
                for (tmp = head; tmp->next!=tail; tmp=tmp->next);
                delete tail;
                tail = tmp;
                tail->next = 0;
        }
        return el;
}
```

Lista jednokierunkowa – przykład - 5

```
void IntList::deleteNode(int el) {
    if (head!=0) {
        if (head == tail && el == head->nr) {
            delete head;
            head = tail= 0;
        }
        else if (el == head->nr) {
            IntNode *tmp = head;
            head = head->next;
            delete tmp;
        }
        else {
            IntNode *pred, *tmp;
            for (pred = head, tmp = head->next; tmp!=0 && !(tmp->nr == el);
                pred = pred->next, tmp = tmp->next);
            if (tmp!=0) {
                pred->next = tmp->next;
                if (tmp == tail) tail = pred;
                delete tmp;
            }
        }
    }
}
```

Lista jednokierunkowa – przykład - 6

```
bool IntList::isInList(int el) {
        IntNode *tmp;
        for (tmp = head; tmp!= 0 && !(tmp->nr == el); tmp = tmp->next);
        return tmp!=0;
}
void IntList::printAll() {
        for (IntNode *tmp = head; tmp!= 0; tmp = tmp->next) cout<<tmp->nr<<"  ";
        cout<<endl;
}
int main() {
        IntList lst;
        lst.addToHead(5);
        lst.addToHead(15);
        lst.addToTail(10);
        lst.printAll();
        lst.deleteNode(5);
        lst.printAll();
        cout<<lst.isInList(10)<<endl;
        lst.deleteFromTail();
        lst.printAll();
        return 1;
}
```

Struktury przechowywania

5. Tablice mieszane:

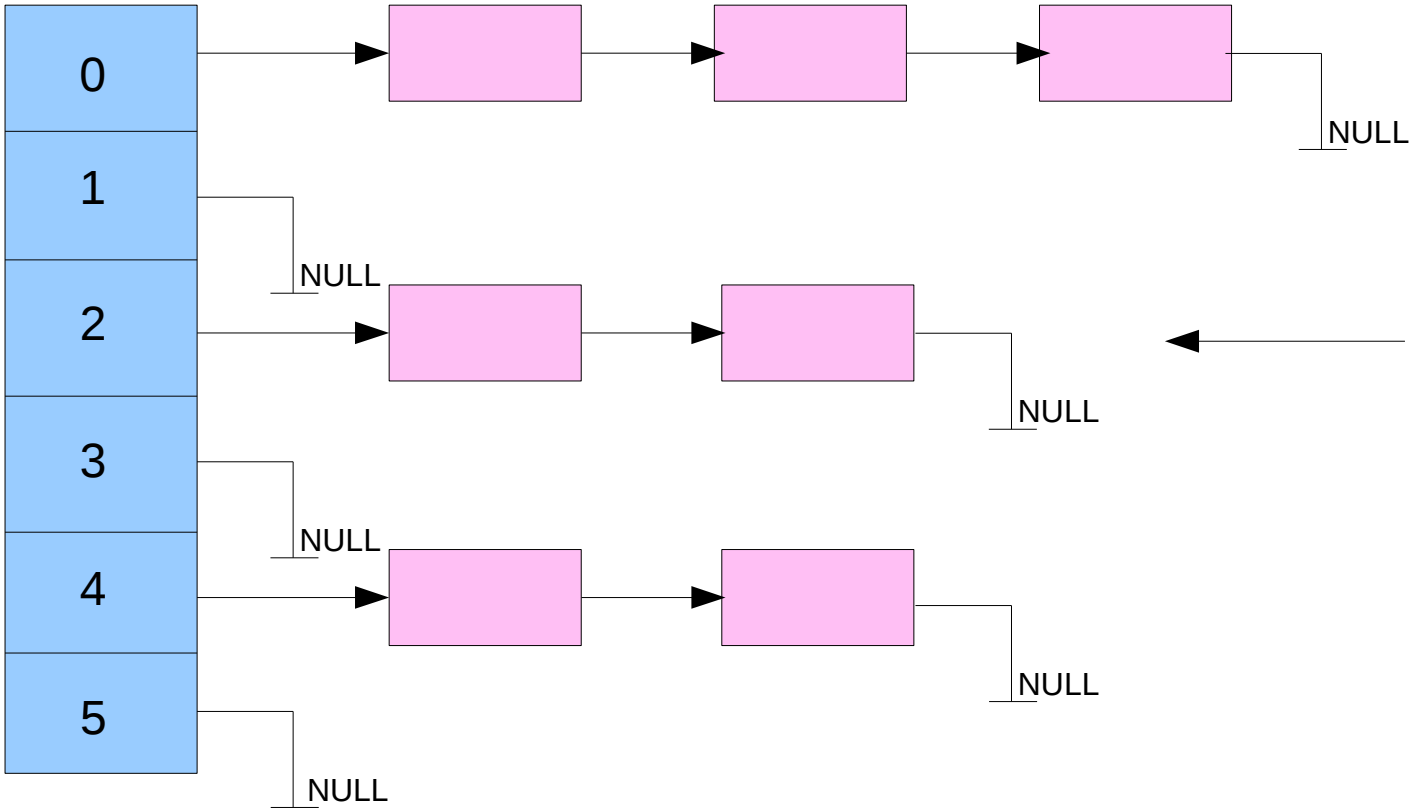
– dobre do dużych ilości danych;

- potrzebna jest funkcja klucza (mieszająca);

- realizacja może używać I innych struktur danych;

- koszt na element może być tak mały jak w przypadku tablicy lub tak duży jak w przypadku innych struktur połączonych

Tablica mieszania (ang. hash table) – łączy realizację tablicy oraz listy (tablica list połączonych)



Struktury przechowywania

Każda osobna lista w tablicy to pojemnik – identyfikowany przez pojedynczy element tablicy (mieszania). Istotna jest informacja o sposobie porządkowania list (w celu określenia, który element tablicy powinien zawierać szukany element); miejsce umieszczenia elementu w tablicy mieszania określane jest przez jedno Z pól tego elementu (klucz – np. nazwisko, numer ubezpieczenia, ..).
Funkcja mieszania – funkcja odwzorująca wartości klucza elementów na pojemniki

Klucz (klucz mieszania)– część przechowywanego elementu, według której powinny być sortowane, wyszukiwane, itd.. elementy bazy danych.

Pojemnik – pojedyncza pozycja tablicy mieszania, za którą może znajdować się inna struktura zawierajaca elementy tego pojemnika

Funkcja mieszająca – odwzoruje wartośc klucza na numer pojemnika

Doskonała funkcja mieszająca – funkcja mieszająca, odwzorujaca każdą unikatową wartość klucza na unikatową wartość całkowitą.

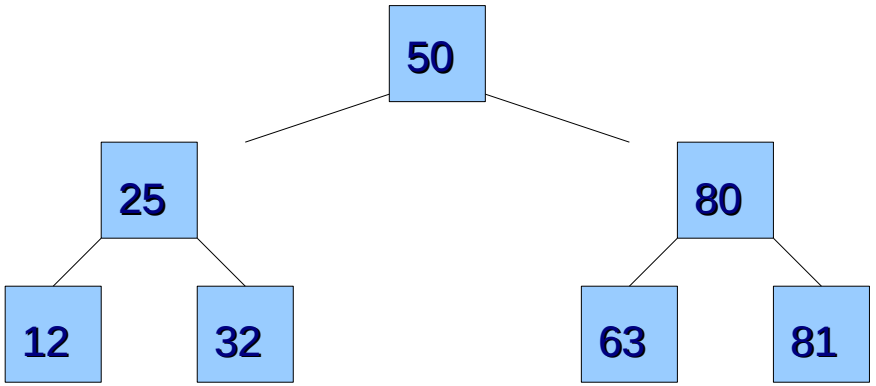
Struktury przechowywania

4. Drzewa binarne (ang. Binary search tree, bst):

– bardzo dobre struktury do bardzo dużej ilości danych dynamicznych (dopóki struktura drzwa jest względnie zrównoważona – dane powinny być dodawane I usuwane w nieuporządkowanej kolejności);

- ma większe koszty na element niż lista I jest równie nieefektywne w przypadku drzewa niezrównoważonego.

Wstawiamy do drzewa: 50, 25, 80, 32, 81, 12, 63

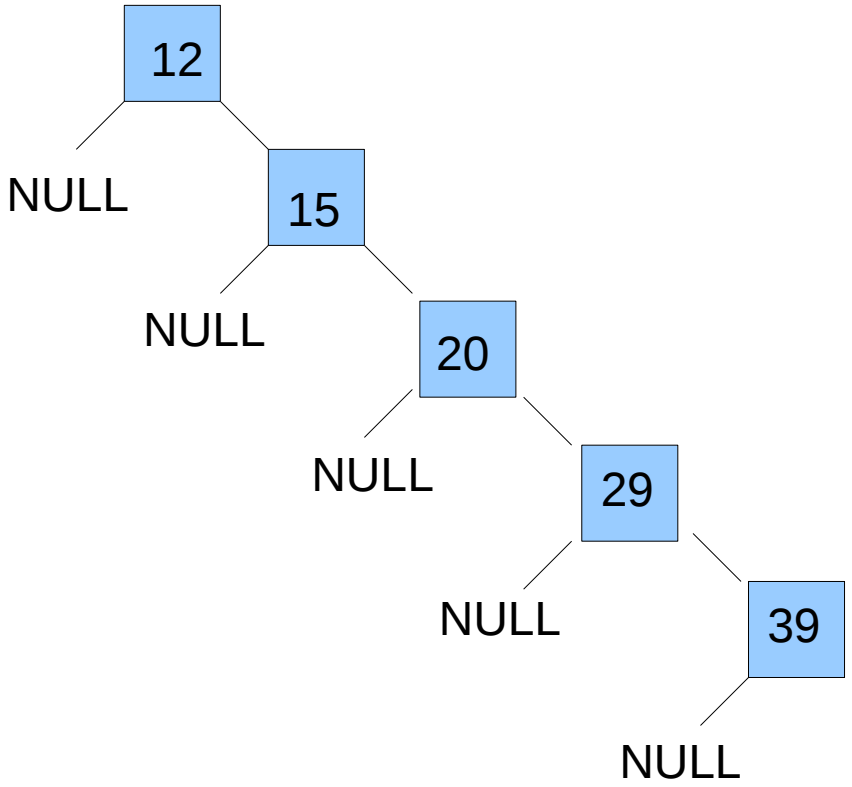


Struktury przechowywania

Przykład wstawiania w drzewo danych posortowanych → drzewo niezrównoważone

Efektywność drzewa niezrównoważonego (poniżej) = efektywność listy

Im bardziej zrównoważone drzewo, tym bardziej efektywne są operacje na nim przeprowadzane (wyszukiwanie).



Struktury przechowywania

Operacje przeprowadzane na drzewie binarnym:

- wstawianie nowego elementu – uwzględnia sortowanie drzewa
- usuwanie elementu drzewa binarnego – usunięcie liścia jest proste, usunięcie elementu, który ma jedno

    Dziecko też jest proste, usunięcie elementu ze swoim poddrzewem już nie, bo należy włączyć

    Do drzewa elementy znajdujące się pod usuwanym elementem
- wyszukiwanie elementu drzewa – zależy od zrównoważenia drzewa
- sortowanie drzewa – nie jest konieczne → drzewo binarne niezrównoważone

W STL są zaimplementowane zbiory, multizbiory, mapy I multimapy ..

TECHNIKI SORTOWANIA

Techniki sortowania

- przez wstawianie
- bąbelkowe
- Shella
- przez kopcowanie
- szybkie
- pozycyjne
- przez scalanie

Sortowanie przez wstawianie

1. Sortowanie przez wstawianie – podstawowy algorytm sotrowania. Sprawdzany jest każdy element tablicy, czy jest on mniejszy czy nie niż wartość elementu poprzedniego. Jeśli tak, jest on przesuwany w miejsce wcześniejsze.

30 20 40 10 (20<30)

20 30 40 10 (30<40 - OK)

20 30 40 10 (10<40)

10 20 30 40

```
void InsertionSort(int *data, int n) {
	int i, j, item;
	for (i=1; i<n, i++) {
		item = data[i];
		for (j=i; (j>0) && (data[j-1] > item); j--) data[j] = data[j-1];
		data[j] = item;
	}
}
```

Sortowanie bąbelkowe - 1

1. Sortowanie babelkowe– podstawowy algorytm sortowania. Sprawdzany jest każdy element tablicy,

Czy jest on większy czy nie niż wartość elementu następnego. Jeśli tak, elementy zamieniane są miejscami.

Cały proces powtarzany jest tyle razy, aż nie zostaną przestawioneżadne elementy miejscami.

30 10 40 20 (30>10)

10 30 40 20 (30<40 - OK)

10 30 40 20 (40>20)

10 30 20 40 (10<30 – OK)

10 30 20 40 (30>20)

10 20 30 40 (30<40 – OK)

10 20 30 40 (OK)

Sortowanie bąbelkowe - 2

```
void BubbleSort(int *data, int n) {
        int i, j, tmp;
        bool changes  = true;
        for (i=1; (i<n) && (changes == true), i++) {
                changes = false;
                for (j=i; (j<(n-1); j++) {
                        if (data[j-1] > data[j]) {
                                tmp = data[j-1];
                                data[j-1] = data[j];
                                data[j] = tmp;
                                changes = true;
                        }
                }
        }
}
```

Sortowanie Shella - 1

3. Sortowanie Shella – prosty algorytm sortowania, w przypadku dużych baz danych jest szybszy od obu poprzednich algortmów sortowania. Podobne do sortowania przez wstawianie, ale elementy można przesuwać na dalsze odległości (odstęp =1). W s.S odstęp może być inny, zwykle przy pierwszych iteracjach jest duży, po czym jest stopniowo zmniejszany w miarę częściowego posortowania bazy danych. W końcowej iteracji odstęp wynosi 1.

Początkowy odstęp = 2, potem 1.

40 70 20 30 60 (40>20 → zamiana)

20 70 40 30 60 (70>30 → zamiana)

20 30 40 70 60 (40<60)

20 30 40 70 60

Odstęp = 1

20 30 40 70 60

.. (20<30)

.. (30<40)

.. (40<70)

20 30 40 70 60 (70>60 → zamiana)

20 30 40 60 70

Wybór początkowego odstępu ma decydujące znaczenie dla wydajności algorytmu

Ciąg odstępów (zaczynamy od 1, potem 3*poprzedni +1): 1, (3*1+1) = 4, (3*4+1) = 13 (zasada Knutha)

Sortowanie Shella - 2

```
void ShellSort(int *data, int lb, int ub) {
	int n, i, j, t, h;
	n = ub – lb +1; //najwieksza wielkosc kroku
	h = 1;
	if (n < 14) h =1;
	else {
		while (h < n) h = 3*h + 1;
		h/= 3;
		h/= 3;
	}
	while (h>0) {
		for (i= lb +h; i<=ub; i++) { //sortuj przez wstawianie z krokiem h
			t = data[i];
			for (j= i-h; j>=lb && (data[j] > t); j-=h) data[j+h] = data[j];
			data[j+h] = t;
		}
			h/=3; //nastepna wielkosc kroku
	}
}
```

Sortowanie szybkie - 1

4. Sortowanie szybkie – algorytm został dodany do biblioteki standardowej qsort(). Należy opracować własną funkcję compare(), której używa qsort() do określenia większego z dwóch elementów.
Algorytm działania ma 2 fazy:dzielenie tablicy na dwie części (z elementem centralnym pivot), inne elementy tablicy są przestawiane wokół tego elementu (elementy mniejsze od pivot'a są w jednej części tablicy, a wieksze w drugiej)

1 7 3 9 6 4 8 2 5

pivot = array[4] = 6

1 7 3 9 6 4 8 2 5 (7>6, 5<6 → zamiana)

1 5 3 9 6 4 8 2 7 (9>6, 2<6 → zamiana)

1 5 3 2 6 4 8 9 7 (4<6 → zamiana)

1 5 3 2 4 6 8 9 7

Istotny jest wybór elementu centralnego (nie musi być on pośrdoku tablicy, może być dowolny, jego wybór wpływa na efektywność. Po podzieleniu tablicy na dwie części, lewa I prawa strona tablicy sortowane są odrębnie (na nowo wybierane są elementy centralne – formuła rekurencyjna).

Sortowanie szybkie - 2

Opis funkcjonalny:

```
int partition(int *data, int n) {
        // wybierz element centralny
        // przestawiaj od data[0] do data[n-1] az do spelnienia warunku
        // data[0] do data[pivot -1] <= data[pivot] oraz
        // data[pivot +1] do data[n-1] >= data[pivot]
        return pivot; //pozycja elementu centralnego po przestawieniu
}

void QuickSort(int *data, int n) {
        if (n >1) {
                int pivot = Partition(data, n);
                if (n < 3) return;
                QuickSort(data, pivot);
                QuickSort(data + pivot +1, n - pivot - 1);
        }
}
```

Sortowanie szybkie – przykład - 1

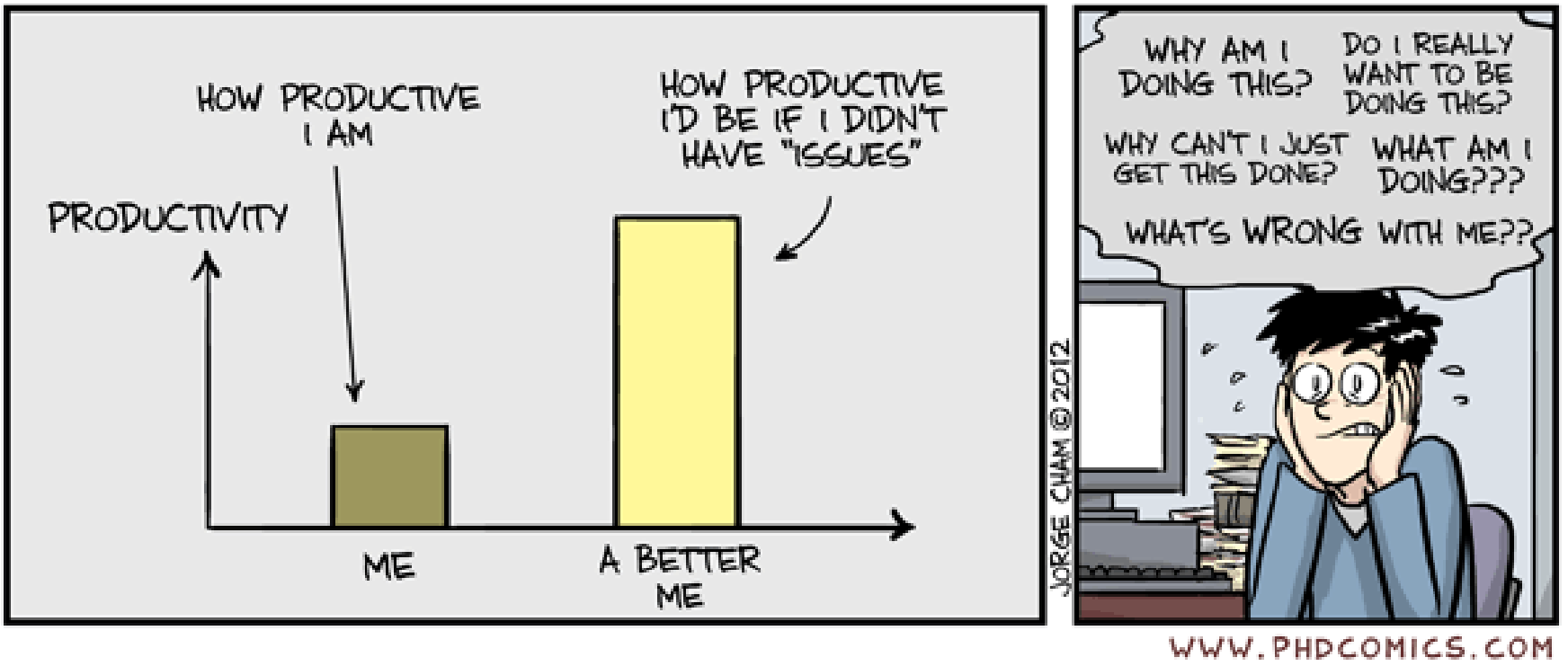
```
#include<iostream>
using namespace std;
template <class T>
int Partition(T tablica[], int p, int r) {
	T x = tablica[p], w;
	int i = p, j = r;
	while (true) {
		while (tablica[j] > x) j--;
		while (tablica[i] < x) i++;
		if (i < j) {
			w = tablica[i];
			tablica[i] = tablica[j];
			tablica[j] = w;
			i++;
			j--;
		}
		else return j;
	}
}
```

Sortowanie szybkie – przykład - 2

```
template <class T>
void QuickSort(T tablica[], int p, int r) {
        int q;
        if (p < r) {
                q = Partition(tablica,p,r);
                QuickSort(tablica, p, q);
                QuickSort(tablica, q+1, r);
        }
}
template <class T>
void printTab(T *data, long n) {
        for (int i=0; i<n; i++) cout<<"\t"<<data[i];
        cout<<endl;
}
int main() {
        short shortArray[] = {10, 20, 11, 317, 315, 510, 12, 709};
        printTab(shortArray, 8);
        QuickSort(shortArray,0, 8);
        printTab(shortArray, 8);
   return 0;
}
```

Sortowanie - podsumowanie

Metoda sortowania	Średnio	Najgorszy przypadek	Uwagi
Wstawianie	O(n^2)	O(n^2)	Do kilku elemntów
Bąbelkowe	O(n^2)	O(n^2)	Do kilku elementów
Shella	O(n^1.25)	O(n^1.5)	
Kopcowanie	O(n log2 n)	O(n log2 n)	Do dużych list
Szybkie	O(n log2n)	O(n^2)	
Pozycyjne	O(m*n)	O(m*n)	Duże listy z liczbami



KONIEC WYKŁADU 14

Obsługa sytuacji wyjątkowych – przykład - 1

Pomysł programu zaczerpnięty od J. Grębosza - “Pasja C++”, tom II –

Wyprawa nieustraszonych :-)

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

typedef unsigned long skarb;

class kapsula{
public:
        double rys;
        void robimy_rysunek(skarb sss) {
                rys = (double) sss;
        }
};
skarb wyprawa_nieustraszonych();
skarb penetracja_stolpu();
int odkryli_nas();
void niestety() ;
void strzelamy();
```

Obsługa sytuacji wyjątkowych – przykład - 4

```
skarb penetracja_stolpu(){
        static skarb urzadzenie = 777 ;
        cout <<"Weszli do stolpu"<<endl;
        if(odkryli_nas()) strzelamy();
        cout <<"Zobaczyli urzadzenie bedace skarbem"<<endl;
        if(odkryli_nas()){
                kapsula kkk ;
                kkk.robimy_rysunek(urzadzenie) ;
                throw kkk ;
        }
        cout <<"Bezpiecznie opuszczamy stolp"<<endl;
        return urzadzenie ;
}

int odkryli_nas(){
        int ile = rand() % 6;
        if(ile == 4) return 1;
        else return 0 ;
}
```

Obsługa sytuacji wyjątkowych – przykład - 2

```
int main(){
        srand((unsigned)time(NULL));
        cout << "Lodz podwodna wyplywa w morze..."<<endl;
        cout << "... doplywa noca do wyspy..."<<endl;
        cout << "Smialkowie beda probowac wykonac zadanie"<<endl;
        skarb ukradziony;                                                    \
        try{
                ukradziony = wyprawa_nieustraszonych();
                cout<<"*** Dowodca gratuluje sukcesu"<<endl<<"*** Oto ukradzione urzadzenie: "
                        << ukradziony << endl<< "*** Oklaski, Medale itd." <<endl;
        }
        catch(int){
                niestety();
                cout <<"Czerwona rakieta: za dobrze strzezona twierdza"<<endl;
        }
        catch(float){
                niestety();
                cout <<"Zielona rakieta: szturmujcie z drugiej strony"<<endl;
        }
        catch(char){
                niestety();
                cout <<"Biala rakieta:  pulapka, uciekajcie "<<endl;
        }
```

Obsługa sytuacji wyjątkowych – przykład - 3

```
        catch(kapsula x){
        niestety();
        cout <<"Rysunek w kapsule"<<endl<<" wylowionej przez pletwonurkow = " <<
              x.rys<<endl;
    }
    cout <<endl<<"Po misji, na lodzi podwodnej"<<endl;
    return 0 ;
}
skarb wyprawa_nieustraszonych(){
    cout <<"Szturmowanie murow"<<endl;
    if(odkryli_nas()) {
        throw (float)1.0 ;
    }
    cout << "  Przebycie dziedzinca"<<endl;
    if(odkryli_nas()) strzelamy();
    skarb sss = penetracja_stolpu();
    cout << " Opuszczenie murow"<<endl;
    return sss ;
}
```

Obsługa sytuacji wyjątkowych – przykład - 5

```
void niestety() {
        cout <<"!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"<<endl;
        cout<<"!!!  Dzialal mechanizm sytuacji wyjatkowych  !!!"<<endl;
        cout<<"!!Nieustraszeni zgineli, ich ostatni sygnal to!!"<<endl;
}


void strzelamy() {
        int flaga_powod = rand() %3 ;
        if(flaga_powod == 0) throw 1 ;
        if(flaga_powod == 1) throw (float)3.14 ;
        if(flaga_powod == 2) throw  'd' ;
}
```

Łapanie lecącego wyjątku – przykład - 1

Przykład autorstwa Jerzego Grębosza “Pasja C++” - tom II

```
#include <iostream>
using namespace std;
class pojazd {
        int x, y ;
};
class samochod : public pojazd {
public:
        int ile_kol ;
        samochod(int k) : ile_kol(k) {}
        // ...
};
int main() {
        // ------ pierwsza zasada ------------------------------
        try {
                int liczba = 7 ;
                cout << "   Katapultujemy obiekt typu int\n";
                throw liczba ;

        }
```

Łapanie lecącego wyjątku – przykład - 2

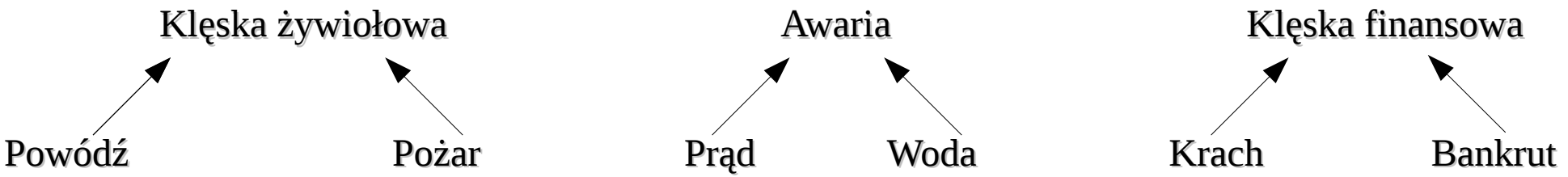
```
catch(const int a) {
        cout << "   Zlapal catch(const int a)\n";
}
/*--------
catch(int a) {              // <-- pasuje idealnie !
        cout << "!! Zlapal catch(int a)\n";
}
------------*/
// ------- druga zasada ------------------------------
try {
        samochod bialy(4) ;
        cout << "   Katapultujemy obiekt typu samochod\n";
        throw bialy ;
}
catch(pojazd p) {
        cout << "   Zlapal catch(pojazd p)\n";
}
catch(...) {
        cout << "   Zlapal catch(...)\n";
}
```

Łapanie lecącego wyjątku – przykład - 3

```
    // ------- trzecia zasada -----------------------------
    try {
        samochod tir(8) ;
        samochod * wsks = &tir ;
        cout << "   Katapultujemy wskaznik do typu samochod\n";
        throw wsks ;
    }
    catch(pojazd * p) {
        cout << "   Zlapal catch(pojazd * p)\n";
    }
    catch(...) {
        cout << "   Zlapal catch(...)\n";
    }
    return 0 ;
}
```

Obsługa sytuacji wyjątkowych – w hierarchii

Przykład programu:



(należy pamiętać o dziedziczeniu wirtualnym w przypadku klas Klęska żywiołowa, Awaria i Klęska finansowa.

Obsługa sytuacji wyjątkowych – w hierarchii – przykład - 1

```
        try {
        bool pyt1= pytanie("Czy byl pozar");
        if (pyt1==true) {
                pozar p("pozar",5);
                throw p;
        }
        else {
                powodz z("powodz", 10);
                throw z;
        }
}
catch (kleska k) {
        k.jaka();
}

try {
        bool pyt2= pytanie("Czy to bankrut");
        if (pyt2==true) {
                bankrut b("Piotr Kowalski",100);
                throw b;
        }
        else {
                krach k("krach", 50);
                throw k;
        }
}
catch (finanse &f) {
        f.jaka();
}
```

Obsługa sytuacji wyjątkowych – w hierarchii – przykład - 2

```
        try {
        bool pyt3= pytanie("Czy byla awaria pradu");
        if (pyt3==true){
                static prad pp("prad", 5);
                throw &pp;
        }
        else {
                static woda ww("woda", 12);
                throw &ww;
        }
}
catch (awaria *wsk) {
        wsk->jaka();
}
```