Instrukcja `do-while`

- Składnia:

  `do` *instrukcja* `(`*`wyrażenie`*`)`

- Przykład

```
do {
 char ch ;
 cin >> ch ;
 cout << ch ;
} while(ch != 'q') ;
```

- Pętla wykonuje się tak długo jak warunek jest prawdziwy.
- Warunek jest sprawdzany na końcu pętli.

Przykład `break` i `continue`

```
for(;;) { // pętla nieskończona
  char ch ;
  cin >> ch ;
  if(ch == 'q') break ; // Wyjdź z pętli
  if(ch == 'a') {
    cout << 'Ala ma kota << endl ;
    continue ; // Następny obrót pętli

  if(ch == 'b') cout << "Beeee!" << endl ;
}
```

Instrukcje `break` i `continue`

- Instrukcja `break` wewnątrz pętli powoduje natychmiastowe wyjście z niej.
- W przypadku zagnieżdżonych pętli wyjście dotyczy tylko jednej (wewnętrznej) pętli.
- Instrukcja *continue* powoduje natychmiastowe przejście do sprawdzania warunku pętli.

Instrukcja `switch`

- Instrukcja `switch` pozwala na wybór wariantu w zależności od wartości podanego wyrażenia całkowitego.
- Składnia:

```
switch(wyrażenie) {
case stała_1:
ciąg instrukcji
// ...
case stała_n:
ciąg instrukcji
default:
ciąg instrukcji
}
```

Instrukcja `switch` c.d.

- Każdy z ciągów instrukcji powinien w zasadzie kończyć się instrukcją `break`, w przeciwnym wypadku zostanie wykonany również następny ciąg instrukcji...
- Ciąg instrukcji z etykietą `default` zostanie wykonany jeśli wartość wyrażenia nie pasuje do żadnego z wymienionych przypadków.

Przykład `switch`

```
int quit = 0 ;
do {
  char ch ;
  cin >> ch ;
  switch(ch) {
  case 'q':
    quit = 1 ;
    break ; // Wyjdź ze switch
  case 'a':
    cout << 'Ala ma kota << endl ;
    break ;
  // ...
  }
} while(!quit)
```

Deklaracje funkcji

- Deklaracja funkcji umożliwia użycie funkcji zdefiniowanej później lub w innym pliku.
- Deklaracja funkcji podaje typ wyniku funkcji, jej nazwę oraz typy argumentów.
- Można (ale nie trzeba) podać nazwy argumentów.
- Typ wyniku *void* oznacza, że funkcja nie daje wyniku
- Deklarację funkcji kończymy średnikiem.
- Przykłady:

```
void drawLine(int x1,int x2,int y1,int y2) ;
int foo(int, float) ;
```

Definicje funkcji

- Definicja funkcji podaje jej nagłówek (podobnie jak deklaracja, z tym, że tu nazwy argumentów są obowiązkowe) oraz treść
- Treść funkcji jest zawsze blokiem
- Definicji funkcji **nie** kończymy średnikiem.
- Przykład:

```
void paint()
{
  drawLine(100,100,300,100) ;
}
```

Powrót z funkcji i dawanie wyniku

- Instrukcja `return` powoduje natychmiastowy powrót z funkcji
- Jeśli funkcja daje wynik (typ wyniku różny od `void`), argumentem dla `return` jest wyrażenie, którego wartość stanie się wynikiem funkcji.
- Przykład:

```
char cfunc(int i) {
  switch(i) {
  case 0: return 'a';
  case 1: return 'g';
  } //switch
} //cfunc
```

- Zwróćmy uwagę, że po `return` użycie `break` jest zbędne.

Modyfikacja obiektów zewnętrznych

- Argumenty funkcji są przekazywane "przez wartość", tj. funkcja otrzymuje kopię wartości argumentu.
- Zmiana wartości argumentu wewnątrz funkcji jest widoczna tylko wewnątrz funkcji

```
void f(int a) {
  a = 5;
}
int main {
  int x = 42 ;
  f(x) ; // x ma nadal wartość 42
}
```

- Jeśli chcemy zmienić wartość obiektu "zewnętrznego" (z punktu widzenia funkcji) możemy przekazać wskaźnik do niego

Rekurencja

- Podobnie jak w innych językach możemy definiować funkcje rekurencyjne.
- Każde wcielenie funkcji ma własne wartości parametrów i zmiennych lokalnych.

```
int silnia(int n) {
  if(n<=1)
    return 1;

  int s ;
  s = silnia(n-1);
  return n * s ;
}
```

Modyfikacja obiektów zewnętrznych

- Jeśli chcemy zmienić wartość obiektu "zewnętrznego" (z punktu widzenia funkcji) możemy przekazać wskaźnik do niego

```
void g(int* a) {
  *a = 5;
}
int main {
  int x = 42 ;
  g(&x) ; // x ma wartość 5
}
```

- Funkcja `g` otrzymuje adres zmiennej `x` i może operować na niej samej, a nie tylko na jej kopii.

Referencje

- Język C++ (w odróżnieniu od C) umożliwia również przekazywanie argumentów przez referencję

```
void h(int& a) {
  a = 5;
}
int main {
  int x = 42 ;
  h(x) ; // x ma wartość 5
}
```

- Wywołanie funkcji `h` wygląda tak samo jak wywołanie przez wartość,ale w istocie przekazywany jest adres zmiennej `x`, a nie kopia jej wartości.

Wskaźniki do funkcji

```
void run(void (*paint)()) {
//...
 (*paint)()
}
void paint() {
  drawLine(100,100,300,100) ;
}
int main() {
  run(paint) ;
}
```

- Argumentem funkcji run jest wskaźnik do funkcji bezwynikowej.

Wskaźniki do funkcji

```
void run(void (*paint)()) {
//...
 (*paint)()
}
void paint() {
  drawLine(100,100,300,100) ;
}
int main() {
  run(paint) ;
}
```

- Dla ustalenia adresu funkcji nie używamy operatora `&` a tylko jej nazwy.

Wskaźniki do funkcji

```
void run(void (*paint)()) {
//...
 (*paint)()
}
void paint() {
  drawLine(100,100,300,100) ;
}
int main() {
  run(paint) ;
}
```

- Jeśli `pf` jest wskaźnikiem do funkcji, to do jej wywołania używamy konstrukcji `(*pf)(argumenty)`

Program w wielu plikach

- Program może być zapisany w wielu plikach źródłowych.
- W C++, podobnie jak w C nie ma jawnego systemu modułów.
- Namiastkę modułu tworzymy zapisując jego interfejs w jednym pliku (często z rozszerzeniem `.h`) a implementację w innym (innych).
- Korzystając z modułu włączamy jego interfejs dyrektywą preprocesora `#include`
- `#include "foo"` działa tak jakby w jej miejscu znalazła się treść pliku `foo`.
- `#include <foo>` oznacza, że pliku foo należy szukać w "standardowych miejscach dla nagłówków" (np. `#include <iostream>`).

Makefile

Zwykle wygodnie jest użyć programu `make`, opisując proces kompilacji w `Makefile`:

```
LIBS=-lX11
LDFLAGS=-L/usr/X11R6/lib -g
CFLAGS=-g

check: dtest
./dtest

drawing.o: drawing.cpp drawing.h
g++ $(CFLAGS) -c $<

btest.o: btest.cpp drawing.h
g++ $(CFLAGS) -c $<

dtest: dtest.o drawing.o
g++ -o $@ $(LDFLAGS) $(LIBS) $^

clean:
-rm -f dtest
-rm -f *.o
```

Kompilacja i łączenie

- Każdy plik z implementacją kompilujemy do pliku obiektowego z rozszerzeniem `.o` (pod Windows `.obj`), np.

  ```
  g++ -c dtest.cpp
  g++ -c drawing.cpp
  ```

- Na zakończenie łączymy uzyskane pliki obiektowe w plik wykonywalny, np:

  ```
  g++ -o dtest dtest.o drawing.o
  ```

- Gotowe biblioteki możemy dołączyć używając opcji `-l`,np

  ```
  g++ -o dtest -lX11 dtest.o drawing.o
  ```

  dołącza bibliotekę zawierającą podstawowe funkcje systemu Xwindow.

Klasy pamięci

- Domyślną klasą pamięci jest `auto`; kompilator zadecyduje gdzie umieścić zmienną (w pamięci, na stosie w rejestrze...).
- Możemy zasugerować umieszczenie zmiennej w rejestrze specyfikatorem `register`:
  `register int i;`
- `extern` oznacza, że zmienna ma przydzieloną pamięć "gdzie indziej" (zwykle w innym module)

Inkrementacja i dekrementacja

- Operator ++ oznacza zwiększenie zmiennej o 1, zaś -- zmniejszenie o 1.
- Wartością wyrażenia jest wartość zmiennej, ale "przed", czy "po"?
- Jeśli operator został umieszczony przed zmienną, to najpierw wykona się operacja, czyli dostaniemy wartość "po"
- Jeśli operator został umieszczony za zmienną, to dostaniemy wartość "przed" (a operacja "potem")

Modyfikator `static`

- `static` w odniesieniu do zmiennej lokalnej nakazuje jej alokację statyczną; jej wartość będzie pamiętana pomiędzy wywołaniami funkcji.
- W odniesieniu do funkcji lub zmiennej globalnej, `static` oznacza "ta nazwa ma nie być widoczna poza tym plikiem"

```
void f() {
  static int i = 0 ;
  cout << ++i << endl;
}
int main() {
  for(x=0;x<10;x++)
    f(); // wydrukuje 1..10
}
```

Stałe

- Stałą możemy zadeklarować używając modyfikatora `const`, np:

  `const int i = 10 ;`
- Stałe obowiązują te same reguły zasięgu co zmienne (tj. możemy mieć stałe lokalne.
- Możemy pobrać adres stałej, ale nie możemy zmodyfikować komórki pamięci o tym adresie.
- Zwykle zadba o to kompilator, ale jeśli stosujemy "brudne sztuczki", dowiemy się o tym dopiero w czasie wykonania...
- Specyfikatora `const` możemy też użyć do argumentu funkcji, np:

  `int f(const int i)`

Modyfikator volatile

- `volatile` jest poniekąd przeciwieństwem `const`; informuje kompilator “nigdy nie wiadomo kiedy ta wartość się zmieni”
- Używamy go wobec wartości poza kontrolą programu, np. rejestrów sprzętowych.
- Ma on również zastosowanie w programach wielowątkowych.

Specyfikatory rozmiaru i znaku

- Do typów `int` oraz `double` można dodawać specyfikator rozmiaru `short` lub `long`. Oznacza to sugestię użycia reprezentacji o mniejszym lub większym od standardowego zakresie (ew. precyzji).
- Dokładne znaczenie jest zależne od implementacji.
- Do typów całkowitych można dodawać specyfikatory znaku `unsigned` lub `signed` oznaczającej użycie liczby bez znaku lub ze znakiem.
- `int` jest domyślnie ze znakiem
- `char`, `signed char` i `unsigned char` to trzy różne typy.